Hans Tompits · Salvador Abreu
Johannes Oetsch · Jörg Pührer
Dietmar Seipel · Masanobu Umeda
Armin Wolf (Eds.)

LNAI 7773

Applications of Declarative Programming and Knowledge Management

19th International Conference, INAP 2011
and 25th Workshop on Logic Programming, WLP 2011
Vienna, Austria, September 28–30, 2011
Revised Selected Papers

Springer

Hans Tompits · Salvador Abreu
Johannes Oetsch · Jörg Pührer
Dietmar Seipel · Masanobu Umeda
Armin Wolf (Eds.)

Applications of Declarative Programming and Knowledge Management

19th International Conference, INAP 2011
and 25th Workshop
on Logic Programming, WLP 2011
Vienna, Austria, September 28–30, 2011
Revised Selected Papers

Springer

Editors
Hans Tompits
Johannes Oetsch
Jörg Pührer
Vienna University of Technology
Vienna
Austria

Salvador Abreu
Universidade de Évora
Evora
Portugal

Dietmar Seipel
Universität Würzburg
Würzburg
Germany

Masanobu Umeda
Kyushu Institute of Technology
Iizuka
Japan

Armin Wolf
Fraunhofer FIRST
Berlin
Germany

ISSN 0302-9743 ISSN 1611-3349 (electronic)
ISBN 978-3-642-41523-4 ISBN 978-3-642-41524-1 (eBook)
DOI 10.1007/978-3-642-41524-1
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013953626

CR Subject Classification I.2, D.1.6

Preface

This volume consists of revised and selected papers presented at the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011), which were held at Hotel Castle Wilheminenberg, Vienna, Austria, during September 28–30, 2011.

INAP is a communicative and dense conference for intensive discussion of applications of important technologies around logic programming, constraint problem solving, and closely related computing paradigms. It comprehensively covers the impact of programmable logic solvers in the Internet society, its underlying technologies, and leading-edge applications in industry, commerce, government, and societal services.

The series of workshops on (constraint) logic programming brings together researchers interested in logic programming, constraint programming, and related areas such as databases and artificial intelligence. Previous workshops have been held in Germany, Austria, Switzerland, and Egypt, serving as the annual meeting of the Society of Logic Programming (GLP, Gesellschaft fur Logische Programmierung e.V.).

Following the success of previous occasions, INAP and WLP were in 2011 again jointly organized in order to promote the cross-fertilization of ideas and experiences among researches and students from the different communities interested in the foundations, applications, and combinations of high-level, declarative programming languages and related areas.

Both events received a total of 35 submissions from authors of 16 countries (Austria, Belgium, Canada, Czech Republic, Egypt, Finland, France, Germany, India, Italy, Japan, Lebanon, Portugal, Slovakia, Tunisia, and the USA). Each submission was assigned to three members of the PC for reviewing, and 27 submissions were accepted for presentation. Among these papers, 19 were selected for this proceedings volume by means of an additional review process. The program also included two invited talks, given by Stefan Szeider and Michael Fink (both from the Vienna University of Technology, Austria).

In concluding, I would like to thank all authors for their submissions and all members of the Program Committee, as well as all additional referees, for the time and effort spent on the careful reviewing of the papers. Furthermore, special thanks go to the members of the Organizing Committee, Johannes Oetsch, Jörg Pührer, and Eva Nedoma, who were indispensable in the realization of the event. Excelsior!

September 2013 Hans Tompits

Organization

Conference Chair INAP 2011

Hans Tompits	Vienna University of Technology, Austria

Track Chairs INAP 2011

Nonmonotonic Reasoning Track

Hans Tompits	Vienna University of Technology, Austria

Applications Track

Masanobu Umeda	Kyushu Institute of Technology, Japan

Extensions of Logic Programming Track

Salvador Abreu	Universidade de Évora, Portugal

Constraint Programming Track

Armin Wolf	Fraunhofer FIRST, Germany

Databases and Data Mining Track

Dietmar Seipel	University of Würzburg, Germany

Program Committee INAP 2011

Salvador Abreu	Universidade de Évora, Portugal
José Alferes	Universidade Nova de Lisboa, Portugal
Sergio Alvarez	Boston College, USA
Grigoris Antoniou	University of Crete, Greece
Marcello Balduccini	Kodak Research Labs, USA

Chitta Baral	Arizona State University, USA
Christoph Beierle	FernUniversität in Hagen, Germany
Philippe Besnard	Université Paul Sabatier, France
Stefan Brass	University of Halle, Germany
Gerd Brewka	University of Leipzig, Germany
Philippe Codognet	University of Tokyo, Japan
Vitor Santos Costa	Universidade do Porto, Portugal
James P. Delgrande	Simon Fraser University, Canada
Marc Denecker	Katholieke Universiteit Leuven, Belgium
Marina De Vos	University of Bath, UK
Daniel Diaz	University of Paris 1, France
Jürgen Dix	Clausthal University of Technology, Germany
Esra Erdem	Sabanci University, Turkey
Gerhard Friedrich	Alpen-Adria-Universität Klagenfurt, Austria
Michael Fink	Vienna University of Technology, Austria
Thom Frühwirth	University of Ulm, Germany
Johannes Fürnkranz	Technische Universität Darmstadt, Germany
Michael Gelfond	Texas Tech University, USA
Carmen Gervet	German University in Cairo, Egypt
Ulrich Geske	University of Potsdam, Germany
Gopal Gupta	University of Texas at Dallas, USA
Petra Hofstedt	Brandenburg University of Technology Cottbus, Germany
Anthony Hunter	University College London, UK
Katsumi Inoue	National Institute of Informatics, Japan
Tomi Janhunen	Aalto University, Finland
Gabriele Kern-Isberner	University of Dortmund, Germany
Nicola Leone	University of Calabria, Italy
Vladimir Lifschitz	University of Texas at Austin, USA
Alessandra Mileo	National University of Ireland
Ulrich Neumerkel	Vienna University of Technology, Austria
Ilkka Niemelä	Aalto University, Finland
Vitor Nogueira	Universidade de Évora, Portugal
David Pearce	Universidad Politécnica de Madrid, Spain
Reinhard Pichler	Vienna University of Technology, Austria
Axel Polleres	National University of Ireland
Enrico Pontelli	New Mexico State University, USA
Irene Rodrigues	Universidade de Évora, Portugal
Carolina Ruiz	Worcester Polytechnic Institute, UK
Torsten Schaub	University of Potsdam, Germany
Dietmar Seipel	University of Würzburg, Germany
V.S. Subrahmanian	University of Maryland, USA
Terrance Swift	Universidade Nova de Lisboa, Portugal
Hans Tompits	Vienna University of Technology, Austria
Masanobu Umeda	Kyushu Institute of Technology, Japan
Kewen Wang	Griffith University, Australia

Emil Weydert	University of Luxembourg, Luxembourg
Armin Wolf	Fraunhofer FIRST, Germany
Osamu Yoshie	Waseda University, Japan

Workshop Chair WLP 2011

Hans Tompits	Vienna University of Technology, Austria

Program Committee WLP 2011

Slim Abdennadher	German University in Cairo, Egypt
Gerd Brewka	University of Leipzig, Germany
Christoph Beierle	FernUniversität in Hagen, Germany
François Bry	University of Munich, Germany
Marc Denecker	Katholieke Universiteit Leuven, Belgium
Marina De Vos	University of Bath, UK
Jürgen Dix	Clausthal University of Technology, Germany
Esra Erdem	Sabanci University, Turkey
Wolfgang Faber	University of Calabria, Italy
Michael Fink	Vienna University of Technology, Austria
Thom Frühwirth	University of Ulm, Germany
Carmen Gervet	German University in Cairo, Egypt
Ulrich Geske	University of Potsdam, Germany
Michael Hanus	Christian Albrechts University of Kiel, Germany
Petra Hofstedt	Brandenburg University of Technology Cottbus, Germany
Steffen Hölldobler	Dresden University of Technology, Germany
Tomi Janhunen	Aalto University, Finland
Ulrich John	SIR Dr. John UG, Germany
Gabriele Kern-Isberner	University of Dortmund, Germany
Alessandra Mileo	National University of Ireland
Axel Polleres	National University of Ireland
Torsten Schaub	University of Potsdam, Germany
Jan Sefranek	Comenius University, Slovakia
Dietmar Seipel	University of Würzburg, Germany
Hans Tompits	Vienna University of Technology, Austria
Armin Wolf	Fraunhofer FIRST, Germany

Local Organization

Hans Tompits	Vienna University of Technology, Austria
Johannes Oetsch	Vienna University of Technology, Austria
Jörg Pührer	Vienna University of Technology, Austria
Eva Nedoma	Vienna University of Technology, Austria

Additional Reviewers

Bogaerts, Bart
De Cat, Broes
Kaminski, Roland
Kaufmann, Benjamin
Krennwallner, Thomas
Mazo, Raul
Oetsch, Johannes
Paulheim, Heiko
Peña, Raúl Mazo
Perri, Simona
Piazza, Carla
Pührer, Jörg
Sneyers, Jon

Contents

Invited Talks

The IMPL Policy Language for Managing Inconsistency in Multi-Context Systems

Thomas Eiter[1], Michael Fink[1(✉)], Giovambattista Ianni[2], and Peter Schüller[1]

[1] Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter, fink, ps}@kr.tuwien.ac.at

[2] Dipartimento di Matematica, Cubo 30B, Università della Calabria, 87036 Rende, CS, Italy
ianni@mat.unical.it

Abstract. Multi-context systems are a declarative formalism for interlinking knowledge-based systems (contexts) that interact via (possibly nonmonotonic) bridge rules. Interlinking knowledge provides ample opportunity for unexpected inconsistencies. These are undesired and come in different categories: some may simply be repaired automatically, while others are more serious and must be inspected by a human operator. In general, no one-fits-all solution exists, since these categories depend on the application scenario. To nevertheless tackle inconsistencies in a general and principled way, we thus propose a declarative policy language for inconsistency management in multi-context systems. We define its syntax and semantics, discuss methodologies for applying the language in real world applications, and outline an implementation by rewriting to acthex, a formalism extending Answer Set Programs.

1 Introduction

The trend to interlink data and information through networked infrastructures, which started out by the spread of the Internet, continues and more recently extends to richer entities of knowledge and knowledge processing. This challenges knowledge management systems that aim at powerful knowledge based applications, in particular when they are built by interlinking smaller existing such systems, and this integration shall be done in a principled way beyond ad-hoc approaches.

Declarative programming methods, and in particular logic programming based approaches, provide rigorous means for developing knowledge based systems through formal representation and model-theoretic evaluation of the knowledge at hand. Extending this technology to advanced scenarios of interlinked information sources is a highly relevant topic of research in declarative knowledge

This research has been supported by the Vienna Science and Technology Fund (WWTF) project ICT08-020. G. Ianni has been partially supported by Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l., and by MIUR under the PRIN project LoDeN.

representation and reasoning. For instance, Multi-context systems (MCSs) [5], based on [7,22], are a generic formalism that captures heterogeneous knowledge bases (contexts) which are interlinked using (possibly nonmonotonic) bridge rules.

The advantages of modular systems, i.e., of building a system from smaller parts, however, poses the problem of unexpected inconsistencies due to unintended interaction of system parts. Such inconsistencies are undesired in general, since inference becomes trivial (under common principles; reminiscent of *ex falso sequitur quodlibet*). The problem of explaining reasons for inconsistency in MCSs has been addressed in [16]: several independent inconsistencies can exist in a MCS, and each inconsistency usually can be repaired in more than one possible way.

For example, imagine a hospital information system which links several databases in order to suggest treatments for patients. A simple inconsistency which can be automatically ignored would be if a patient states her birth date correctly at the front desk, but swaps two digits filling in a form at the X-ray department. An entirely different type of inconsistency is (at least as far as the health of the patient is concerned), if the patient needs treatment, but all options are in conflict with some allergy of the patient. Attempting an automatic repair may not be a viable option in this case: a doctor should inspect the situation and make a decision.

In view of such scenarios, tackling inconsistency requires individual strategies and targeted (re)actions, depending on the type of inconsistency and on the application. In this work, we thus propose the declarative *Inconsistency Management Policy Language* (IMPL), which provides a means to specify inconsistency management strategies for MCSs. Our contributions are briefly summarized as follows.

- We define the syntax of IMPL, inspired by Answer Set Programming (ASP) [21] following the syntax of ASP programs. In particular, we specify the *input for policy reasoning*, as being provided by dedicated reserved predicates. These predicates encode inconsistency analysis results in terms of the respective structures in [16]. Furthermore, we specify *action predicates that can be derived by rules.* Actions provide a means to counteract inconsistency by modifying the MCS, and may involve interaction with a human operator.
- We define the semantics of IMPL in a three-step process. In a first step, models of a given policy are calculated. Then, in a second step, the effects of actions which are present in such a model are determined (this possibly involves user interaction). Finally, in a third step, these effects are applied to the MCS.
- On the basis of the above, we provide methodologies for utilizing IMPL in application scenarios, and briefly discuss useful language extensions.
- Finally, we give the necessary details of a concrete realization of IMPL by rewriting it to the acthex formalism [2] which extends ASP programs with external computations and actions.

The remainder of the paper is organized as follows: we first introduce MCS and notions for explaining inconsistency in MCSs in Sect. 2. We then define syntax and semantics of the IMPL policy language in Sect. 3, describe methodologies for applying IMPL in practice in Sect. 4, provide a possibility for realizing IMPL by rewriting to acthex in Sect. 5, and conclude the paper in Sect. 6.

2 Preliminaries

Multi-context systems (MCSs). A heterogeneous nonmonotonic MCS [5] consists of *contexts*, each composed of a knowledge base with an underlying *logic*, and a set of *bridge rules* which control the information flow between contexts.

A logic $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$ is an abstraction which captures many monotonic and nonmonotonic logics, e.g., classical logic, description logics, or default logics. It consists of the following components, the first two intuitively define the logic's syntax, the third its semantics:

- $\mathbf{KB}_L$ is the set of well-formed knowledge bases of L. We assume each element of $\mathbf{KB}_L$ is a set of "formulas".
- $\mathbf{BS}_L$ is the set of possible belief sets, where a belief set is a set of "beliefs".
- $\mathbf{ACC}_L : \mathbf{KB}_L \rightarrow 2^{\mathbf{BS}_L}$ assigns to each KB a set of acceptable belief sets.

Since contexts may have different logics, this allows to model heterogeneous systems.

Example 1. For *propositional logic* L_{prop} under the closed world assumption over signature Σ, $\mathbf{KB}$ is the set of propositional formulas over Σ; $\mathbf{BS}$ is the set of deductively closed sets of propositional Σ-literals; and $\mathbf{ACC}(kb)$ returns for each kb a singleton set, containing the set of literal consequences of kb under the closed world assumption. □

A *bridge rule* models information flow between contexts: it can add information to a context, depending on the belief sets accepted at other contexts. Let $L = (L_1, \ldots, L_n)$ be a tuple of logics. An L_k-bridge rule r over L is of the form

$$(k : s) \leftarrow (c_1 : p_1), \ldots, (c_j : p_j), \mathbf{not}\ (c_{j+1} : p_{j+1}), \ldots, \mathbf{not}\ (c_m : p_m). \quad (1)$$

where k and c_i are context identifiers, i.e., integers in the range $1, \ldots, n$, p_i is an element of some belief set of L_{c_i}, and s is a formula of L_k. We denote by $h_b(r)$ the formula s in the head of r and by $B(r) = \{(c_1 : p_1), \ldots, \mathbf{not}\ (c_{j+1} : p_{j+1}), \ldots\}$ the set of body literals (including negation) of r.

A multi-context system $M = (C_1, \ldots, C_n)$ is a collection of contexts $C_i = (L_i, kb_i, br_i)$, $1 \leq i \leq n$, where $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ is a logic, $kb_i \in \mathbf{KB}_i$ a knowledge base, and br_i is a set of L_i-bridge rules over $(L_1, \ldots, L_n)$. By $IN_i = \{h_b(r) \mid r \in br_i\}$ we denote the set of possible *inputs* of context C_i added by bridge rules. For each $H \subseteq IN_i$ it is required that $kb_i \cup H \in \mathbf{KB}_{L_i}$. By $br_M = \bigcup_{i=1}^{n} br_i$ we denote the set of all bridge rules of M.

The following running example will be used throughout the paper.

Example 2 (*generalized from* [16]). Consider a MCS M_1 in a hospital which comprises the following contexts: a patient database C_{db}, a blood and X-Ray analysis database C_{lab}, a disease ontology C_{onto}, and an expert system C_{dss} which suggests proper treatments. Knowledge bases are given below; initial uppercase letters are used for variables and description logic concepts.

$$\begin{aligned}
kb_{db} &= \{\, person(sue, 02/03/1985), allergy(sue, ab1)\},\\
kb_{lab} &= \{\, customer(sue, 02/03/1985), test(sue, xray, pneumonia),\\
&\quad\; test(Id, X, Y) \rightarrow \exists D : customer(Id, D)),\\
&\quad\; customer(Id, X) \land customer(Id, Y) \rightarrow X = Y\},\\
kb_{onto} &= \{\, Pneumonia \sqcap Marker \sqsubseteq AtypPneumonia\},\\
kb_{dss} &= \{\, give(Id, ab1) \lor give(Id, ab2) \leftarrow need(Id, ab).\\
&\quad\; give(Id, ab1) \leftarrow need(Id, ab1).\\
&\quad\; \neg give(Id, ab1) \leftarrow not\ allow(Id, ab1), need(Id, Med).\}.
\end{aligned}$$

Context C_{db} uses propositional logic (see Example 1) and provides information that Sue is allergic to antibiotics '*ab1*'. Context C_{lab} is a database with constraints which stores laboratory results connected to Sue: *pneumonia* was detected in an X-ray. Constraints enforce, that each test result must be linked to a *customer* record, and that each customer has only one birth date. C_{onto} specifies that presence of a blood marker in combination with pneumonia indicates atypical pneumonia. This context is based on $\mathcal{AL}$, a basic description logic [1]: $\mathbf{KB}_{onto}$ is the set of all well-formed theories within that description logic, $\mathbf{BS}_{onto}$ is the powerset of the set of all assertions $C(o)$ where C is a concept name and o an individual name, and $\mathbf{ACC}_{onto}$ returns the set of all concept assertions entailed by a given theory. C_{dss} is an ASP program that suggests a medication using the *give* predicate.

Schemas for bridge rules of M_1 are as follows:

$$\begin{aligned}
r_1 &= (lab : customer(Id, Birthday)) &&\leftarrow (db : person(Id, Birthday)).\\
r_2 &= (onto : Pneumonia(Id)) &&\leftarrow (lab : test(Id, xray, pneumonia)).\\
r_3 &= (onto : Marker(Id)) &&\leftarrow (lab : test(Id, bloodtest, m1)).\\
r_4 &= (dss : need(Id, ab)) &&\leftarrow (onto : Pneumonia(Id)).\\
r_5 &= (dss : need(Id, ab1)) &&\leftarrow (onto : AtypPneumonia(Id)).\\
r_6 &= (dss : allow(Id, ab1)) &&\leftarrow \mathbf{not}\ (db : allergy(Id, ab1).
\end{aligned}$$

Rule r_1 links the patient records with the lab database (so patients do not need to enter their data twice). Rules r_2 and r_3 provide test results from the lab to the ontology. Rules r_4 and r_5 link disease information with medication requirements, and r_6 associates acceptance of the particular antibiotic '*ab1*' with a negative allergy check on the patient database. □

Equilibrium semantics [5] selects certain belief states of a MCS $M = (C_1, \dots, C_n)$ as acceptable. A *belief state* is a list $S = (S_1, \dots, S_n)$, s.t. $S_i \in \mathbf{BS}_i$. A bridge rule (1) is *applicable* in S iff for $1 \leq i \leq j$: $p_i \in S_{c_i}$ and for $j < l \leq m$: $p_l \notin S_{c_l}$. Let $app(R, S)$ denote the set of bridge rules in R that are applicable in belief state S. Then a belief state $S = (S_1, \dots, S_n)$ of M is an *equilibrium* iff, for $1 \leq i \leq n$, the following condition holds: $S_i \in \mathbf{ACC}_i(kb_i \cup \{hd(r) \mid r \in app(br_i, S)\})$.

For simplicity we will disregard the issue of grounding bridge rules (see [20]), and only consider ground instances of bridge rules. In the following, with $r_1, \ldots, r_6$ we refer to the ground instances of the respective bridge rules in Example 2, where variables are replaced by $Id \mapsto sue$ and $Birthday \mapsto 02/03/1985$ (all other instances are irrelevant).

Example 3 (*ctd*). MCS M_1 has one equilibrium $S = (S_{db}, S_{lab}, S_{onto}, S_{dss})$, where $S_{db} = kb_{db}, S_{lab} = \{customer(sue, 02/03/1985), test(sue, xray, pneumonia)\}$, $S_{onto} = \{Pneumonia(sue)\}$, and$S_{dss} = \{need(sue, ab), give(sue, ab2), \neg give(sue, ab1)\}$. Moreover, bridge rules r_1, r_2, and r_4 are applicable under S. □

Explaining Inconsistency in MCSs. *Inconsistency* in a MCS is the lack of an equilibrium [16]. Note that no equilibrium may exist even if all contexts are 'paraconsistent' in the sense that for all $kb \in \mathbf{KB}, \mathbf{ACC}(kb)$ is nonempty. No information can be obtained from an inconsistent MCS, e.g., inference tasks like brave or cautious reasoning on equilibria become trivial. To analyze, and eventually repair, inconsistency in a MCS, we use the notions of consistency-based *diagnosis* and entailment-based *inconsistency explanation* [16], which characterize inconsistency by sets of involved bridge rules.

Intuitively, a diagnosis is a pair (D_1, D_2) of sets of bridge rules which represents a concrete system repair in terms of removing rules D_1 and making rules D_2 unconditional. The intuition for considering rules D_2 as unconditional is that the corresponding rules should become applicable to obtain an equilibrium. One could consider more fine-grained changes of rules such that only some body atoms are removed instead of all. However, this increases the search space while there is little information gain: every diagnosis (D_1, D_2) as above, together with a witnessing equilibrium S, can be refined to such a generalized diagnosis. Dual to that, inconsistency explanations (short: explanations) separate independent inconsistencies. An explanation is a pair (E_1, E_2) of sets of bridge rules, such that the presence of rules E_1 and the absence of heads of rules E_2 necessarily makes the MCS inconsistent. In other words, bridge rules in E_1 cause an inconsistency in M which cannot be resolved by considering additional rules already present in M or by modifying rules in E_2 (in particular making them unconditional). See [16] for formal definitions of these notions, relationships between them, and more background discussion.

Example 4 (*ctd*). Consider a MCS M_2 obtained from M_1 by modifying kb_{lab}: we replace $customer(sue, 02/03/1985)$ by the two facts $customer(sue, 03/02/1985)$ and $test(sue, bloodtest, m1)$, i.e., we change the birth date, and add a blood test result. M_2 is inconsistent with two minimal inconsistency explanations $e_1 = (\{r_1\}, \emptyset)$ and $e_2 = (\{r_2, r_3, r_5\}, \{r_6\})$: e_1 characterizes the problem, that C_{lab} does not accept any belief set because constraint $customer(Id, X) \wedge customer(Id, Y) \rightarrow X = Y$ is violated. Another independent inconsistency is pointed out by e_2: if e_1 is repaired, then C_{onto} accepts $AtypPneumonia(sue)$, therefore r_5 imports the need for $ab1$ into C_{dss} which makes C_{dss} inconsistent due to Sue's allergy. Moreover, the following minimal diagnoses exist for M_2: $(\{r_1, r_2\}, \emptyset)$, $(\{r_1, r_3\}, \emptyset)$, $(\{r_1, r_5\}, \emptyset)$, and $(\{r_1\}, \{r_6\})\}$. For instance, diagnosis $(\{r_1\}, \{r_6\})$

removes bridge rule r_1 from M_2 and adds r_6 unconditionally to M_2, which yields a consistent MCS. □

3 Policy Language IMPL

Dealing with inconsistency in an application scenario is difficult, because even if inconsistency analysis provides information how to restore consistency, it is not obvious which choice of system repair is rational. It may not even be clear whether it is wise at all to repair the system by changing bridge rules.

Example 5 (*ctd*). Repairing e_1 by removing r_1 and thereby ignoring the birth date (which differs at the granularity of months) may be the desired reaction and could very well be done automatically. On the contrary, repairing e_2 by ignoring either the allergy or the illness is a decision that should be left to a doctor, as every possible repair could cause serious harm to Sue. □

Therefore, managing inconsistency in a controlled way is crucial. To address these issues, we propose the declarative *Inconsistency Management Policy Language* IMPL, which provides a means to create policies for dealing with inconsistency in MCSs. Intuitively, an IMPL policy specifies (i) which inconsistencies are repaired automatically and how this shall be done, and (ii) which inconsistencies require further external input, e.g., by a human operator, to make a decision on how and whether to repair the system. Note that we do not rule out automatic repairs, but — contrary to previous approaches — automatic repairs are done only if a given policy specifies to do so, and only to the extent specified by the policy.

Since a major point of MCSs is to abstract away context internals, IMPL treats inconsistency by modifying bridge rules. For the scope of this work we delegate any potential repair by modifying the *kb* of a context to the user. The effect of applying an IMPL policy to an inconsistent MCS M is a *modification* (A, R), which is a pair of sets of bridge rules which are syntactically compatible with M. Intuitively, a modification specifies bridge rules A to be added to M and bridge rules R to be removed from M, similar as for diagnoses without restriction to the original rules of M.

An IMPL policy P for a MCS M is intended to be evaluated on a set of *system and inconsistency analysis* facts, denoted EDB_M, which represents information about M, in particular EDB_M contains atoms which describe bridge rules, minimal diagnoses, and minimal explanations of M.

The evaluation of P yields certain actions to be taken, which potentially interact with a human operator, and modify the MCS at hand. This modification has the potential to restore consistency of M.

In the following we formally define syntax and semantics of IMPL.

3.1 Syntax

We assume disjoint sets C, V, *Built*, and *Act*, of constants, variables, built-in predicate names, and action names, respectively, and a set of ordinary predicate names $\mathit{Ord} \subseteq C$. Constants start with lowercase letters, variables with uppercase letters, built-in predicate names with #, and action names with @. The set of *terms* T is defined as $T = C \cup V$.

An *atom* is of the form $p(t_1, \dots, t_k)$, $0 \leq k$, $t_i \in T$, where $p \in \mathit{Ord} \cup \mathit{Built} \cup \mathit{Act}$ is an ordinary predicate name, builtin predicate name, or action name. An atom is ground if $t_i \in C$ for $0 \leq i \leq k$. The sets A_{Act}, A_{Ord}, and A_{Built}, called sets of *action atoms*, *ordinary atoms*, and *builtin atoms*, consist of all atoms over T with $p \in \mathit{Act}$, $p \in \mathit{Ord}$, respectively $p \in \mathit{Built}$.

Definition 1. *An* IMPL policy *is a finite set of rules of the form*

$$h \leftarrow a_1, \dots, a_j, \mathit{not}\, a_{j+1}, \dots, \mathit{not}\, a_k. \tag{2}$$

where h is an atom from $A_{\mathit{Ord}} \cup A_{\mathit{Act}}$, every a_i, $1 \leq i \leq k$, is from $A_{\mathit{Ord}} \cup A_{\mathit{Built}}$, and 'not' is negation as failure.

Given a rule r, we denote by $H(r)$ its head, by $B^+(r) = \{a_1, \dots, a_j\}$ its positive body atoms, and by $B^-(r) = \{a_{j+1}, \dots, a_k\}$ its negative body atoms. A rule is ground if it contains ground atoms only. A ground rule with $k = 0$ is a *fact*. As in ASP, a rule must be safe, i.e., variables in $H(r)$ or in $B^-(r)$ must also occur in $B^+(r)$. For a set of rules R, we use $\mathit{cons}(R)$ to denote the set of constants from C appearing in R, and $\mathit{pred}(R)$ for the set of ordinary predicate names and action names (elements from $\mathit{Ord} \cup \mathit{Act}$) in R.

We next describe how a policy represents information about the MCS M under consideration.

System and Inconsistency Analysis Predicates. Entities, diagnoses, and explanations of the MCS M at hand are represented by a suitable finite set $C_M \subseteq C$ of constants which uniquely identify contexts, bridge rules, beliefs, rule heads, diagnoses, and explanations. For convenience, when referring to an element represented by a constant c we identify it with the constant, e.g., we write 'bridge rule r' instead of 'bridge rule of M represented by constant r'.

Reserved atoms use predicates from the set $C_{\mathit{res}} \subseteq \mathit{Ord}$ of *reserved predicates*, with $C_{\mathit{res}} = \{\mathit{ruleHead}, \mathit{ruleBody}^+, \mathit{ruleBody}^-, \mathit{context}, \mathit{modAdd}, \mathit{modDel}, \mathit{diag}, \mathit{explNeed}, \mathit{explForbid}\}$. They represent the following information.

- $\mathit{context}(c)$ denotes that c is a context.
- $\mathit{ruleHead}(r, c, s)$ denotes that bridge rule r is at context c with head formula s.
- $\mathit{ruleBody}^+(r, c, b)$ (resp., $\mathit{ruleBody}^-(r, c, b)$) denotes that bridge rule r contains body literal '$(c\,{:}\,b)$' (resp., '**not** $(c\,{:}\,b)$').
- $\mathit{modAdd}(m, r)$ (resp., $\mathit{modDel}(m, r)$) denotes that modification m adds (resp., deletes) bridge rule r. Note that r is represented using *ruleHead* and *ruleBody*.
- $\mathit{diag}(m)$ denotes that modification m is a minimal diagnosis in M.
- $\mathit{explNeed}(e, r)$ (resp., $\mathit{explForbid}(e, r)$) denotes that the minimal explanation (E_1, E_2) identified by constant e contains bridge rule $r \in E_1$ (resp., $r \in E_2$).

- $\mathit{modset}(\mathit{ms}, m)$ denotes that modification m belongs to the set of modifications identified by ms.

Example 6 (*ctd*). We can represent r_1, r_5, and the diagnosis $(\{r_1, r_5\}, \emptyset)$ as the set of reserved atoms $I_{ex} = \{\mathit{ruleHead}(r_1, c_{lab}, `\mathit{customer}(\mathit{sue}, \mathit{02/03/1985})'),$ $\mathit{ruleBody}^+(r_1, c_{db}, `\mathit{person}(\mathit{sue}, \mathit{02/03/1985})')$, $\mathit{ruleHead}(r_5, c_{dss}, `\mathit{need}(\mathit{sue}, \mathit{ab1})')$, $\mathit{ruleBody}^+(r_5, c_{onto}, `\mathit{AtypPneumonia}(\mathit{sue})')$, $\mathit{modDel}(d, r_1)$, $\mathit{modDel}(d, r_5)$, $\mathit{diag}(d)\}$ where constant d identifies the diagnosis. □

Further knowledge used as input for policy reasoning can easily be defined using additional (supplementary) predicates. Note that predicates over all explanations or bridge rules can easily be defined by projecting from reserved atoms. Moreover, to encode preference relations (e.g., as in [17]) between system parts, diagnoses, or explanations, an atom $\mathit{preferredContext}(c_1, c_2)$ could denote that context c_1 is considered more reliable than context c_2. The extensions of such auxiliary predicates need to be defined by the rules of the policy or as additional input facts (ordinary predicates), or they are provided by the implementation (built-in predicates), i.e., the 'solver' used to evaluate the policy. The rewriting to acthex given in Sect. 5.2 provides a good foundation for adding supplementary predicates as built-ins, because the acthex language has generic support for calls to external computational sources. A possible application would be to use a preference relation between bridge rules that is defined by an external predicate and can be used for reasoning in the policy.

Towards a more formal definition of a policy input, we distinguish the set B_M of ground atoms built from reserved predicates C_{res} and terms from C_M, called *MCS input base*, and the *auxiliary input base* B_{Aux} given by predicates over $\mathit{Ord} \setminus C_{res}$ and terms from C. Then, the *policy input base* $B_{Aux,M}$ is defined as $B_{Aux} \cup B_M$. For a set $I \subseteq B_{Aux,M}$, $I|_{B_M}$ and $I|_{B_{Aux}}$ denote the restriction of I to predicates from the respective bases.

Now, given an MCS M, we say that a set $S \subseteq B_M$ is a *faithful representation* of M wrt. a reserved predicate $p \in C_{res} \setminus \{\mathit{modset}\}$ iff the extension of p in S exactly characterizes the respective entity or property of M (according to a unique naming assignment associated with C_M as mentioned). For instance, $\mathit{context}(c) \in S$ iff c is a context of M, and correspondingly for the other predicates. Consequently, S is a faithful representation of M iff it is a faithful representation wrt. all p in $C_{res} \setminus \{\mathit{modset}\}$ and the extension of modset in S is empty.

A finite set of facts $I \subseteq B_{Aux,M}$ containing a faithful representation of all relevant entities and properties of an MCS qualifies as input of a policy, as defined next.

Definition 2. *A* policy input *I wrt. MCS M is a finite subset of the policy input base $B_{Aux,M}$, such that $I|_{B_M}$ is a faithful representation of M.*

In the following, we denote by EDB_M a policy input wrt. a MCS M. Note that reserved predicate modset has an empty extension in a policy input (but corresponding atoms will be of use later on in combination with actions). Given a set of reserved atoms I, let c be a constant that appears as a bridge rule identifier in I. Then $\mathit{rule}_I(c)$ denotes the corresponding bridge rule represented by reserved atoms *ruleHead*, $\mathit{ruleBody}^+$, and $\mathit{ruleBody}^-$ in I with c as their first argument. Similarly we denote by $\mathit{mod}_I(m) = (A, R)$ (resp., by $\mathit{modset}_I(m) = \{(A_1, R_1), \ldots\}$) the modification (resp., set of modifications) represented in I by the respective predicates and identified by constant m.

Subsequently, we call a modification m that is projected to rules located at a certain context c the *projection* of m to context c (and similarly for sets of modifications). Formally we denote by $\mathit{mod}_I(m)|_c$ (resp., $\mathit{modset}_I(m)|_c$) the projection of modification (resp., set of modifications) m in I to context c.

Example 7 (*ctd*). In the previous example I_{ex}, $\mathit{rule}_{I_{ex}}(r_1)$ refers to rule r_1; moreover $\mathit{mod}_{I_{ex}}(d) = (\{r_1, r_5\}, \emptyset)$ and the projection of modification d to c_{dss} is $(\{r_5\}, \emptyset)$. □

A policy can create representations of new rules, modifications, and sets of modifications, because reserved atoms are allowed to occur in heads of policy rules. However such new entities require new constants identifying them. To tackle this issue, we next introduce a facility for value invention.

Value Invention via Builtin Predicates '$\#id_k$'. Whenever a policy specifies a new rule and uses it in some action, the rule must be identified with a constant. The same is true for modifications and sets of modifications. Therefore, IMPL contains a family of special builtin predicates which provide policy writers a means to comfortably create new constants from existing ones.

For this purpose, builtin predicates of the form $\#id_k(c', c_1, \ldots, c_k)$ may occur in rule bodies (only). Their intended usage is to uniquely (and thus reproducibly) associate a new constant c' with a tuple $c_1, \ldots, c_k$ of constants (for a formal semantics see the definitions for action determination in Sect. 3.2).

Note that this value invention feature is not strictly necessary, as new constants can be obtained via defining an order relation over all constants, a pool of unused constants, and auxiliary rules that use the next unused constant for each new constant that is required by the program. However, a dedicated value invention builtin simplifies policy writing and improves policy readability.

Example 8. Assume one wants to consider projections of modifications to contexts as specified by the extension of an auxiliary predicate $\mathit{projectMod}(M, C)$. The following policy fragment achieves this using a value invention builtin to assign a unique identifier with every projection (recorded in the extension of another auxiliary predicate $\mathit{projectedModId}(M', M, C)$).

$$\left\{\begin{array}{rl} \mathit{projectedModId}(M', M, C) \leftarrow & \mathit{projectMod}(M, C), \\ & \#id_3(M', pm_{id}, M, C); \\ \mathit{modAdd}(M', R) \leftarrow & \mathit{modAdd}(M, R), \mathit{ruleHead}(R, C, S), \\ & \mathit{projectedModId}(M', M, C); \\ \mathit{modDel}(M', R) \leftarrow & \mathit{modDel}(M, R), \mathit{ruleHead}(R, C, S), \\ & \mathit{projectedModId}(M', M, C) \end{array}\right\} \tag{3}$$

Intuitively, we identify new modifications by new ids $c_{pm_{id},M,C}$ obtained from M and C via $\#id_3$ and an auxiliary constant $pm_{id} \notin C_M$. The latter simply serves the purpose of disambiguating constants used for projections of modifications. □

Besides representing modifications of a MCS aiming at resolving inconsistency, an important feature of IMPL is to actually apply them. Actions serve this purpose.

Actions. Actions alter the MCS at hand and may interact with a human operator. According to the change that an action performs, we distinguish *system actions* which modify the MCS in terms of entire bridge rules that are added and/or deleted, and *rule actions* which modify a single bridge rule. Moreover, the changes can depend on external input, e.g., obtained by user interaction. In the latter case, the action is termed *interactive*. Accumulating the changes of all actions yields an overall modification of the MCS. We formally define this intuition when addressing the semantics in Sect. 3.2.

Syntactically, we use @ to prefix action names from Act, and those of the predefined actions listed below are reserved action names. Let M be the MCS under consideration, then the following predefined actions are (non-interactive) system actions:

- $@delRule(r)$ removes bridge rule r from M.
- $@addRule(r)$ adds bridge rule r to M.
- $@applyMod(m)$ applies modification m to M.
- $@applyModAtContext(m, c)$ applies those changes in m to the MCS that add or delete bridge rules at context c (i.e., applies the projection of m to c).

Note that a policy might specify conflicting effects, i.e., the removal and the addition of a bridge rule at the same time. In this case the semantics gives preference to addition.

The predefined actions listed next are rule actions:

- $@addRuleCondition^+(r, c, b)$ (resp., $@addRuleCondition^-(r, c, b)$) adds body literal $(c\,{:}\,b)$ (resp., **not** $(c\,{:}\,b)$) to bridge rule r.
- $@delRuleCondition^+(r, c, b)$ (resp., $@delRuleCondition^-(r, c, b)$) removes body literal $(c\,{:}\,b)$ (resp., **not** $(c\,{:}\,b)$) from bridge rule r.
- $@makeRuleUnconditional(r)$ makes bridge rule r unconditional.

Since these actions can modify the same rule, this may also result in conflicting effects, where again addition is given preference over removal by the semantics. (Moreover, rule modifications are given preference over addition or removal of the entire rule.)

Eventually, the subsequent predefined actions are interactive (system) actions, i.e., they involve a human operator:

- $@guiSelectMod(ms)$ displays a GUI for choosing from the set of modifications ms. The modification chosen by the user is applied to M.

- $@\mathit{guiEditMod}(m)$ displays a GUI for editing modification m. The resulting modification is applied to M.[1]
- $@\mathit{guiSelectModAtContext}(ms, c)$ projects modifications in ms to c, displays a GUI for choosing among them and applies the chosen modification to M.
- $@\mathit{guiEditModAtContext}(m, c)$ projects modification m to context c, displays a GUI for editing it, and applies the resulting modification to M.

As we define formally in Sect. 3.2, changes of individual actions are not applied directly, but collected into an overall modification which is then applied to M (respecting preferences in case of conflicts as stated above). Before turning to a formal definition of the semantics, we give example policies.

Example 9 (*ctd*). Figure 1 shows three policies that can be useful for managing inconsistency in our running example. Their intended behavior is as follows. P_1 deals with inconsistencies at C_{lab}: if an explanation concerns only bridge rules at C_{lab}, an arbitrary diagnosis is applied at C_{lab}, other inconsistencies are not handled. Applying P_1 to M_2 removes r_1 at C_{lab}, the resulting MCS is still inconsistent with inconsistency explanation e_2, as only e_1 has been automatically fixed. P_2 extends P_1 by adding an 'inconsistency alert formula' to C_{lab} if an

Policies (sets of IMPL rules)	Intuitive meaning of rules in each set
$P_1 = \{\mathit{expl}(E) \leftarrow \mathit{explNeed}(E, R);$	Define domain predicate
$\mathit{expl}(E) \leftarrow \mathit{explForbid}(E, R);$	for explanations.
$\mathit{incNotLab}(E) \leftarrow \mathit{explNeed}(E, R),$	Find out whether one explanation
$\mathit{ruleHead}(R, C, F), C \neq c_{lab};$	only concerns bridge rules at c_{lab}.
$\mathit{incNotLab}(E) \leftarrow \mathit{explForbid}(E, R),$	
$\mathit{ruleHead}(R, C, F), C \neq c_{lab};$	
$\mathit{incLab} \leftarrow \mathit{expl}(E), \mathit{not}\ \mathit{incNotLab}(E);$	
$\mathit{in}(D) \leftarrow \mathit{not}\ \mathit{out}(D), \mathit{diag}(D), \mathit{incLab};$	Guess a diagnosis.
$\mathit{out}(D) \leftarrow \mathit{not}\ \mathit{in}(D), \mathit{diag}(D), \mathit{incLab};$	
$\bot \leftarrow \mathit{in}(A), \mathit{in}(B), A \neq B;$	Ensure that we guess exactly one
$\mathit{useOne} \leftarrow \mathit{in}(D);$	diagnosis if there is a local
$\bot \leftarrow \mathit{not}\ \mathit{useOne}, \mathit{incLab};$	inconsistency at c_{lab}.
$@\mathit{applyModAtContext}(D, c_{lab}) \leftarrow$	Apply the guessed diagnosis after
$\mathit{useDiag}(D)\}$	projecting it to context c_{lab}.
$P_2 = \{\mathit{ruleHead}(r_{alert}, c_{lab}, \mathit{alert}) \leftarrow;$	Define new inconsistency alert rule r_{alert}.
$@\mathit{addRule}(r_{alert}) \leftarrow \mathit{incLab}\}$	Add that new rule to c_{lab}.
$\cup\ P_1$	Reuse policy P_1.
$P_3 = \{\mathit{modset}(md, X) \leftarrow \mathit{diag}(X);$	Create modification set with all diagnoses.
$@\mathit{guiSelectMod}(md) \leftarrow\}$	Let the user choose from that set.

Fig. 1. Sample IMPL policies for our running example.

[1] It is suggestive to also give the operator a possibility to abort, causing no modification at all to be made, however we do not specify this here because a useful design choice depends on the concrete application scenario.

inconsistency was automatically repaired at that context. Finally, P_3 is a fully manual approach which displays a choice of all minimal diagnoses to the user and applies the user's choice. Note, that we did not combine automatic actions and user-interactions here since this would result in more involved policies (and/or require an iterative methodology; cf. Sect. 4). $\square$

We refer to the predefined IMPL actions $@\mathit{delRule}$, $@\mathit{addRule}$, $@\mathit{guiSelectMod}$, and $@\mathit{guiEditMod}$ as *core* actions, and to the remaining ones as *comfort* actions. Comfort actions exist for convenience of use, providing means for projection and for rule modifications. They can be rewritten to core actions as sketched in the following example.

Example 10. To realize $@\mathit{applyMod}(M)$ and $@\mathit{applyModAtContext}(M, C)$ using the core language, we replace them by $\mathit{applyMod}(M)$ and $\mathit{applyModAtContext}$ (M, C), respectively, use rules (3) from Example 8, and add the following set of rules.

$$\left\{\begin{array}{rl} @\mathit{addRule}(R) \leftarrow & \mathit{applyMod}(M),\ \ \mathit{modAdd}(M, R); \\ @\mathit{delRule}(R) \leftarrow & \mathit{applyMod}(M),\ \ \mathit{modDel}(M, R); \\ \mathit{projectMod}(M, C) \leftarrow & \mathit{applyModAtContext}(M, C); \\ \mathit{applyMod}(M') \leftarrow & \mathit{applyModAtContext}(M, C), \\ & \mathit{projectedModId}(M', M, C) \end{array}\right\} \tag{4}$$

$\square$

This concludes our introduction of the syntax of IMPL, and we move on to a formal development of its semantics which so far has only been conveyed by accompanying intuitive explanations.

3.2 Semantics

The semantics of applying an IMPL policy P to a MCS M is defined in three steps:

– *Actions* to be executed are determined by computing a *policy answer set* of P wrt. policy input EDB_M.
– *Effects of actions* are determined by executing actions. This yields modifications (A, R) of M for each action. Action effects can be nondeterministic and thus only be determined by executing respective actions (which is particularly true for user interactions).
– Effects of actions are *materialized* by building the componentwise union over individual action effects and applying the resulting modification to M.

In the remainder of this section, we introduce the necessary definitions for a precise formal account of these steps.

Action Determination. We define IMPL policy answer sets similar to the stable model semantics [21]. Given a policy P and a policy input EDB_M, let id_k be a fixed (built-in) family of one-to-one mappings from k-tuples $c_1, \ldots, c_k$, where

$c_i \in \mathit{cons}(P \cup \mathit{EDB}_M)$ for $1 \leq i \leq k$, to a set $C_{id} \subset C$ of 'fresh' constants, i.e., disjoint from $\mathit{cons}(P \cup \mathit{EDB}_M)$.[2] Then the *policy base* $B_{P,M}$ of P wrt. EDB_M is the set of ground IMPL atoms and actions, that can be built using predicate symbols from $\mathit{pred}(P \cup \mathit{EDB}_M)$ and terms from $U_{P,M} = \mathit{cons}(P \cup \mathit{EDB}_M) \cup C_{id}$, called policy universe.

The *grounding of* P, *denoted by* $\mathit{grnd}(P)$, is given by grounding its rules wrt. $U_{P,M}$ as usual. Note that, since $\mathit{cons}(P \cup \mathit{EDB}_M)$ is finite, only a finite amount of mapping functions id_k is used in P. Hence only a finite amount of constants C_{id} is required, and therefore $U_{P,M}$, $B_{P,M}$, and $\mathit{grnd}(P)$ are finite as well.

An *interpretation* is a set of ground atoms $I \subseteq B_{P,M}$. We say that I *models* an atom $a \in B_{P,M}$, denoted $I \models a$ iff (i) a is not a built-in atom and $a \in I$, or (ii) a is a built-in atom of the form $\#id_k(c, c_1, \ldots, c_k)$ and $c = id_k(c_1, \ldots, c_k)$. I models a set of atoms $A \subseteq B_{P,M}$, denoted $I \models A$, iff $I \models a$ for all $a \in A$. I models the body of rule r, denoted as $I \models B(r)$, iff $I \models a$ for every $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$; and for a ground rule r, $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$. Eventually, I *is a model of* P, denoted $I \models P$, iff $I \models r$ for all $r \in \mathit{grnd}(P)$. The *FLP-reduct* [19] of P wrt. an interpretation I, denoted fP^I, is the set of all $r \in \mathit{grnd}(P)$ such that $I \models B(r)$.[3]

Definition 3 (Policy Answer Set). *Given an MCS M, let P be an* IMPL *policy, and let EDB_M be a policy input wrt. M. An interpretation $I \subseteq B_{P,M}$ is a* policy answer set *of P for EDB_M iff I is a $\subseteq$-minimal model of $fP^I \cup \mathit{EDB}_M$.*

We denote by $\mathcal{AS}(P \cup \mathit{EDB}_M)$ the set of all policy answer sets of P for EDB_M.

Effect Determination. We define the effects of action predicates $@a \in \mathit{Act}$ by nondeterministic functions $f_{@a}$. Nondeterminism is required for interactive actions. An action is evaluated wrt. an interpretation of the policy and yields an effect according to its type: the effect of a system action is a modification (A, R) of the MCS, in the following sometimes denoted *system modification*, while the effect of a rule action is a *rule modification* $(A, R)_r$ wrt. a bridge rule r of M, i.e., in this case A is a set of bridge rule body literals to be added to r, and R is a set of bridge rule body literals to be removed from r.

Definition 4. *Given an interpretation I, and a ground action α of form $@a(t_1, \ldots, t_k)$, the* effect *of α wrt. I is given by $\mathit{eff}_I(\alpha) = f_{@a}(I, t_1, \ldots, t_k)$, where $\mathit{eff}_I(\alpha)$ is a system modification if α is a system action, and a rule modification if α is a rule action.*

Action predicates of the IMPL core fragment have the following semantic functions.

[2] Disjointness ensures finite groundings; without this restriction, e.g., the program $\{p(C') \leftarrow \#id_1(C', C);\ p(C)\}$ would not have finite grounding.

[3] We use the FLP reduct for compliance with acthex (used for realization in Sect. 5), but for the language considered, the Gelfond-Lifschitz reduct would yield an equivalent definition.

- $f_{@delRule}(I, r) = (\emptyset, \{rule_I(r)\})$.
- $f_{@addRule}(I, r) = (\{rule_I(r)\}, \emptyset)$.
- $f_{@guiSelectMod}(I, ms) = (A, R)$ where (A, R) is the user's selection after being displayed a choice among all modifications in $\{(A_1, R_1), \ldots\} = modset_I(ms)$.
- $f_{@guiEditMod}(I, m) = (A', R')$, where (A', R') is the result of user interaction with a modification editor that is preloaded with modification $(A, R) = mod_I(m)$.

Note that the effect of any core action in I can be determined independently from the presence of other core actions in I, and rule modifications are not required to define the semantics of core actions. However, rule modifications are needed to capture the effect of *comfort* actions. Moreover, adding and deleting rule conditions, and making a rule unconditional can modify the same rule, therefore such action effects yield accumulated rule modifications.

More specifically, the semantics of IMPL comfort actions is defined as follows:

- $f_{@delRuleCondition^+}(I, r, c, b) = (\emptyset, \{(c : b)\})_r$.
- $f_{@delRuleCondition^-}(I, r, c, b) = (\emptyset, \{\mathbf{not}\ (c : b)\})_r$.
- $f_{@addRuleCondition^+}(I, r, c, b) = (\{(c : b)\}, \emptyset)_r$.
- $f_{@addRuleCondition^-}(I, r, c, b) = (\{\mathbf{not}\ (c : b)\}, \emptyset)_r$.
- $f_{@makeRuleUnconditional}(I, r) = (\emptyset, \{(c_1 : p_1), \ldots, (c_j : p_j), \mathbf{not}\ (c_{j+1} : p_{j+1}), \ldots, \mathbf{not}\ (c_m : p_m)\})_r$ for r of the form (1).
- $f_{@applyMod}(I, m) = mod_I(m)$.
- $f_{@applyModAtContext}(I, m, c) = mod_I(m)|_c$.
- $f_{@guiSelectModAtContext}(I, ms, c) = (A', R')$ where (A', R') is the user's selection after being displayed a choice among all modifications in $\{(A'_1, R'_1), \ldots\} = modset_I(ms)|_c$.
- $f_{@guiEditModAtContext}(I, m, c) = (A', R')$, where (A', R') is the result of user interaction with a modification editor that is preloaded with modification $mod_I(m)_c$.

In practice, however, it is not necessary to implement action functions on the level of rule modifications, since a policy in the comfort fragment can equivalently be rewritten to the core fragment (which does not rely on rule modifications). Example 10 already sketched a rewriting for @*applyMod* and @*applyModAtContext*. For a complete rewriting from the comfort to the core fragment, we refer to the extended version of this paper [15].

The effects of user-defined actions have to comply to Definition 4.

Effect Materialization. Once the effects of all actions in a selected policy answer set have been determined, an overall modification is computed by the componentwise union over all individual modifications. This overall modification is then materialized in the MCS.

Given a MCS M and a policy answer set I (for a policy P and a corresponding policy input EDB_M), let I_M, respectively I_R, denote the set of ground system actions, respectively rule actions, in I. Then, $M_{eff} = \{eff_I(\alpha) | \alpha \in I_M\}$ is the set of effects of system action atoms in I, and $R_{eff} = \{eff_I(\alpha) | \alpha \in I_R\}$ is the set of effects of rule actions in I. Furthermore, $\mathit{Rules} = \{r \mid (A,R)_r \in R_{\mathit{eff}}\}$ is the set of bridge rules modified by R_{eff}, and for every $r \in \mathit{Rules}$, let $\mathcal{R}_r = \bigcup_{(A,R)_r \in R_{\mathit{eff}}} R$, respectively $\mathcal{A}_r = \bigcup_{(A,R)_r \in R_{\mathit{eff}}} A$, denote the union of rule body removals, respectively additions, wrt. r in R_{eff}.

Definition 5. *Given a MCS M, and an* IMPL *policy P, let I be a policy answer set of P for a policy input EDB_M wrt. M. Then, the* materialization of I *in M is the MCS M' obtained from M by replacing its set of bridge rules br_M by the set*

$$(br_M \setminus \mathcal{R} \cup \mathcal{A}) \setminus \mathit{Rules} \cup \mathcal{M},$$

where $\mathcal{R} = \bigcup_{(A,R) \in M_{\mathit{eff}}} R$, $\mathcal{A} = \bigcup_{(A,R) \in M_{\mathit{eff}}} A$, and $\mathcal{M} = \{(k{:}s) \leftarrow \mathit{Body} \mid r \in \mathit{Rules},\ r \in br_k,\ h_b(r) = s,\ \mathit{Body} = B(r) \setminus \mathcal{R}_r \cup \mathcal{A}_r\}$.

Note that, by definition, the addition of bridge rules has precedence over removal, and the addition of body literals similarly has precedence over removal. There is no particular reason for this choice; one just has to be aware of it when specifying a policy. Apart from that, no order for evaluating individual actions is specified or required.

Eventually, we can define modifications of a MCS that are accepted by a corresponding IMPL policy.

Definition 6. *Given a MCS M, an* IMPL *policy P, and a policy input EDB_M wrt. M, a modified MCS M'* is an admissible modification of *M wrt. P and EDB_M iff M' is the materialization of some policy answer set $I \in \mathcal{AS}(P \cup \mathit{EDB}_M)$.*

Example 11 (*ctd*). For brevity we here do not give a full account of a proper EDB_{M_2} of our running example. Intuitively EDB_{M_2} represents all bridge rules, minimal diagnoses and minimal explanations, in a similar fashion as already shown in Ex. 6. We assume, that the two explanations and four diagnoses given in Ex. 4 are identified by constants e_1, e_2, d_1, ..., d_4, respectively.

Evaluating $P_2 \cup \mathit{EDB}_{M_2}$ yields four policy answer sets, one is $I_1 = \mathit{EDB}_{M_2} \cup \{\mathit{expl}(e_1),\ \mathit{expl}(e_2),\ \mathit{incNotLab}(e_2),\ \mathit{incLab},\ \mathit{in}(d_1),\ \mathit{out}(d_2),\ \mathit{out}(d_3),\ \mathit{out}(d_4),$ $\mathit{useOne},\ \mathit{ruleHead}(r_{\mathit{alert}}, c_{\mathit{lab}}, \mathit{alert}),\ @\mathit{addRule}(r_{\mathit{alert}}),\ @\mathit{applyModAtContext}(d_1,$ $c_{\mathit{lab}})\}$. From I_1 we obtain a single admissible modification of M_2 wrt. P_2: add bridge rule r_{alert} and remove r_1.

Evaluating $P_3 \cup \mathit{EDB}_{M_2}$ yields one policy answer set, which is $I_2 = \mathit{EDB}_{M_2} \cup \{\mathit{modset}(\mathit{md}, d_1),\ \mathit{modset}(\mathit{md}, d_2),\ \mathit{modset}(\mathit{md}, d_3),\ \mathit{modset}(\mathit{md}, d_4),\ @\mathit{guiSelect}$-$\mathit{Mod}(\mathit{md})\}$. Determining the effect of I_2 involves user interaction; thus multiple materializations of I_2 exist. For instance, if the user chooses to ignore Sue's allergy and birth date (and probably imposes additional monitoring on Sue), then we obtain an admissible modification of M: it adds the unconditional version of r_6 and removes r_1. □

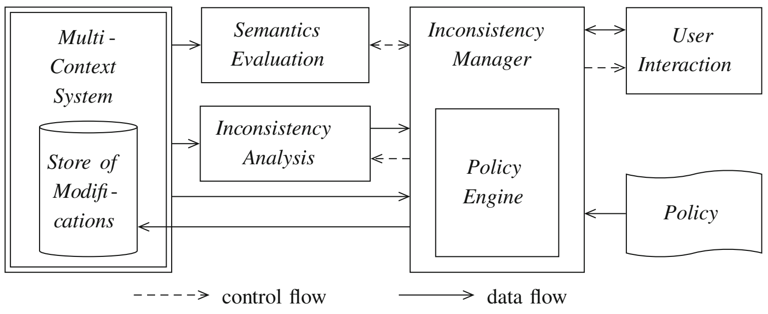


Fig. 2. Policy integration data flow and control flow block diagram.

4 Methodologies of Applying IMPL and Realization

Based on the simple system design shown in Fig. 2, we next briefly discuss elementary methodologies of applying IMPL for the purpose of integrating MCS reasoning with potential user interaction in case of inconsistency. Due to space constraints, we restrict ourselves to an informal discussion.

We maintain a representation of the MCS together with a *store of modifications*. The *semantics evaluation* component performs reasoning tasks on the MCS and invokes the *inconsistency manager* in case of an inconsistency. This inconsistency manager uses the *inconsistency analysis* component[4] to provide input for the *policy engine* which computes policy answer sets of a given IMPL *policy* wrt. the MCS and its inconsistency analysis result. This policy evaluation step results in action executions potentially involving user interactions and causes changes to the store of modifications, which are subsequently materialized. Finally the inconsistency manager hands control back to the semantics evaluation component. Principal modes of operation, and their merits, are the following.

Reason and Manage once. This mode of operation evaluates the policy once, if the effect materialization does not repair inconsistency in the MCS, no further attempts are made and the MCS stays inconsistent. While simple, this mode may not be satisfying in practice.

However, one can improve on the approach by extending actions with priority: the result of a single policy evaluation step then may be a sequence of sets of actions (of equal priority), corresponding to successive *attempts* (of increasing priority) for repairing the MCS. This can be exploited for writing policies that ensure repairs, by first attempting a 'sophisticated' repair possibly involving user interaction, and — if this fails — to simply apply some diagnosis to ensure consistency while the problem may be further investigated.

[4] For realizations of this component we refer to [3,16].

Reason and Manage iteratively. Another way to deal with failure to restore consistency is to simply invoke policy evaluation again on the modified but still inconsistent system. This is useful if user interaction may involve trial-and-error, especially if multiple inconsistencies occur: some might be more difficult to counteract than others.

Another positive aspect of iterative policy evaluation is, that it allows for policies to be structured, e.g., as follows: (a) classify inconsistencies into automatically versus manually repairable; (b) apply actions to repair one of the automatically repairable inconsistencies; (c) if such inconsistencies do not exist: apply user interaction actions to repair one (or all) of the manually repairable inconsistencies. Such policy structuring follows a divide-and-conquer approach, trying to focus on individual sources of inconsistency and to disregard interactions between inconsistencies as much as possible. If user interaction consists of trial-and-error bugfixing, fewer components of the system are changed in each iteration, and the user starts from a situation where only critical (i.e. not automatically repairable) inconsistencies are present in the MCS. Moreover, such policies may be easier to write and maintain. On the other hand, termination of iterative methodologies is not guaranteed. However, one can enforce termination by limiting the number of iterations, possibly by extending IMPL with a *control action* that configures this limit.

In iterative mode, passing information from one iteration to the next may be useful. This can be accomplished by considering additional user-defined add and delete actions which modify an iteration-persistent *knowledge base*, provided to the policy as further input (by means of dedicated auxiliary predicates). For more details we refer to [15].

5 Realizing IMPL in acthex

In this section, we demonstrate how IMPL can be realized using acthex. First we give preliminaries about acthex which is a logic programming formalism that extends HEX programs with executable actions. We then show how to implement the core IMPL fragment by rewriting it to acthex in Sect. 5.2. A rewriting from the comfort to the core fragment of IMPL is given in the extended version of this paper [15].

5.1 Preliminaries on acthex

The acthex formalism [2] generalizes HEX programs [18] by adding dedicated action atoms to heads of rules. An acthex program operates on an *environment*; this environment can influence external sources in acthex, and it can be modified by the execution of actions.

Syntax. By $\mathcal{C}$, $\mathcal{X}$, $\mathcal{G}$, and $\mathcal{A}$ we denote mutually disjoint sets whose elements are called constant names, variable names, external predicate names, and action predicate names, respectively. Elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ (resp., $\mathcal{A}$) are prefixed with "&" (resp. "#"). Names in $\mathcal{C}$ serve both as constant and predicate names, and we assume that $\mathcal{C}$ contains a finite subset of consecutive integers $\{0, \ldots, n_{max}\}$.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots Y_n)$, where $Y_0, Y_1, \ldots Y_n$ are terms, and $n \geq 0$ is the arity of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1 \ldots Y_n)$. An atom is ordinary if Y_0 is a constant. An external atom is of the form $\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m)$ with $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ being lists of terms. An action atom is of the form $\#g\,[Y_1, \ldots, Y_n]\,\{o, r\}\,[w : l]$ where $\#g$ is an action predicate name, $Y_1, \ldots, Y_n$ is a list of terms (called input list), and each action predicate $\#g$ has fixed length $in(\#g) = n$ for its input list. Attribute $o \in \{b, c, c_p\}$ is called the *action option*; depending on o the action atom is called *brave, cautious, and preferred cautious*, respectively. Attributes r, w, and l are called *precedence*, *weight*, and *level* of $\#g$, denoted by $prec(a)$, $weight(a)$, and $level(a)$, respectively. They are optional and range over variables and positive integers.

A *rule* r is of the form $\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\,\beta_{n+1}, \ldots, not\,\beta_m$, where m, n, $k \geq 0$, $m \geq n$, $\alpha_1, \ldots, \alpha_k$ are atoms or action atoms, and $\beta_1, \ldots \beta_m$ are atoms or external atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. An acthex *program* is a finite set P of rules.

Example 12. The acthex program $\{\#robot[goto, charger]\{b, 1\} \leftarrow \&sensor[bat](low);\ \#robot[clean, kitchen]\{c, 2\} \leftarrow night;\ \#robot[clean, bedroom]\{c, 2\} \leftarrow day;\ night \vee day \leftarrow\}$ uses action atom $\#robot$ to command a robot, and an external atom $\&sensor$ to obtain sensor information. Precedence 1 of action atom $\#robot[goto, charger]\{b, 1\}$ makes the robot recharge its battery before executing cleaning actions, if necessary. □

Semantics. Intuitively, an acthex program P is evaluated wrt. an *external environment* E using the following steps: (i) *answer sets* of P are determined wrt. E, the set of *best models* is a subset of the answer sets determined by an objective function; (ii) one best model is selected, and one *execution schedule* S is generated for that model (although a model may give rise to multiple execution schedules); (iii) the *effects of action atoms* in S are applied to E in the order defined by S, yielding an updated environment E'; and finally (iv) the process may be iterated starting at (i), unless no actions were executed in (iii) which terminates an iterative evaluation process. Formally the acthex semantics is defined as follows.

Given an acthex program P, the *Herbrand base* HB_P of P is the set of all possible ground versions of atoms, external atoms, and action atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. Given a rule $r \in P$, the grounding $grnd(r)$ of r is defined accordingly, the grounding of program P is given as the grounding of all its rules. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$, $\mathcal{G}$, and $\mathcal{A}$ are implicitly given by P.

An *interpretation I relative to P* is any subset $I \subseteq HB_P$ containing ordinary atoms and action atoms. We say that I is a *model* of atom (or action atom) $a \in HB_P$, denoted by $I \models a$, iff $a \in I$. With every external predicate name $\&g \in \mathcal{G}$, we associate an $(n{+}m{+}2)$-ary Boolean function $f_{\&g}$, assigning each tuple $(E, I, y_1, \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $x_i, y_j \in \mathcal{C}$, $I \subseteq HB_P$, and E is an environment. Note that this slightly generalizes the external atom semantics such that they may take E into account, which was left implicit in [2]. We say that an interpretation I relative to P is a *model* of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$ wrt. environment E, denoted as $I, E \models a$, iff $f_{\&g}(E, Iy_1 \ldots, y_n x_1, \ldots, x_m) = 1$. Let r be a ground rule. We define (i) $I, E \models H(r)$ iff there is some $a \in H(r)$ such that $I, E \models a$, (ii) $I, E \models B(r)$ iff $I, E \models a$ for all $a \in B^+(r)$ and $I, E \not\models a$ for all $a \in B^-(r)$, moreover (iii) $I, E \models r$ iff $I, E \models H(r)$ or $I, E \not\models B(r)$. We say that I is a *model* of P wrt. E, denoted by $I, E \models P$, iff $I, E \models r$ for all $r \in grnd(P)$. The *FLP-reduct* of P wrt. I and E, denoted as $fP^{I,E}$, is the set of all $r \in grnd(P)$ such that $I, E \models B(r)$. Eventually, I is an *answer set* of P wrt. E iff I is a $\subseteq$-minimal model of $fP^{I,E}$. We denote by $\mathcal{AS}(P, E)$ the collection of all answer sets of P wrt. E.

The set of *best models* of P, denoted $\mathcal{BM}(P, E)$, contains those $I \in \mathcal{AS}(P, E)$ that minimize the objective function $H_P(I) = \Sigma_{a \in A}(\omega \cdot level(a) + weight(a))$, where $A \subseteq I$ is the set of action atoms in I, and ω is the first limit ordinal. (This definition using ordinal numbers is equivalent to the definition of weak constraint semantics in [8].)

An action $a = \#g[y_1, \ldots, y_n]\{o, r\}[w : l]$ with option o and precedence r is *executable in I wrt. P and E* iff (i) a is brave and $a \in I$, or (ii) a is cautious and $a \in B$ for every $B \in \mathcal{AS}(P, E)$, or a is preferred cautious and $a \in B$ for every $B \in \mathcal{BM}(P, E)$. An *execution schedule* of a best model I is a sequence of all actions executable in I, such that for all action atoms a, $b \in I$, if $prec(a) < prec(b)$ then a has a lower index in the sequence than b. We denote by $\mathcal{ES}_{P,E}(I)$ the set of all execution schedules of a best model I wrt. acthex program P and environment E; formally, let A_e be the set of action atoms that are executable in I wrt. P and E, then $\mathcal{ES}_{P,E}(I) = \big\{[a_1, \ldots, a_n] \mid prec(a_i) \leq prec(a_j)$, for all $1 \leq i < j \leq n$, and $\{a_1, \ldots, a_n\} = A_e\big\}$.

Example 13. In Example 12, if the robot has low battery, then $\mathcal{AS}(P, E) = \mathcal{BM}(P, E)$ contains models $I_1 = \{night, \#robot[clean, kitchen]\{c, 2\}, \#robot[goto, charger]\{b, 1\}\}$ and $I_2 = \{day, \#robot[clean, bedroom]\{c, 2\}, \#robot[goto, charger]b, 1\}$. We have $\mathcal{ES}_{P,E}(I_1) = \{\#robot[goto, charger]\{b, 1\}, \#robot[clean, bedroom]c, 2\}$. □

Given a model I, the *effect of executing a ground action* $\#g[y_1, \ldots, y_m]\{o, p\}[w : l]$ on an environment E wrt. I is defined for each action predicate name $\#g$ by an associated $(m{+}2)$-ary function $f_{\#g}$ which returns an updated environment $E' = f_{\#g}(E, I, y_1, \ldots, y_m)$. Correspondingly, given an execution schedule $S = [a_1, \ldots, a_n]$ of a model I, the *execution outcome* of S in environment E wrt. I is defined as $EX(S, I, E) = E_n$, where $E_0 = E$, and $E_{i+1} =$

$f_{\#g}(E_i, I, y_1, \ldots, y_m)$, given that a_i is of the form $\#g[y_1, \ldots, y_m]\{o, p\}[w : l]$. Intuitively the initial environment $E_0 = E$ is updated by each action in S in the given order. The set of possible execution outcomes of a program P on an environment E is denoted as $\mathcal{EX}(P, E)$, and formally defined by $\mathcal{EX}(P, E) = \{EX(S, I, E) \mid S \in \mathcal{ES}_{P,E}(I)$ where $I \in \mathcal{BM}(P, E)\}$.

In practice, one usually wants to consider a single execution schedule. This requires the following decisions during evaluation: (i) to select one best model $I \in \mathcal{BM}(P, E)$, and (ii) to select one execution schedule $S \in \mathcal{ES}_{P,E}(I)$. Finally, one can then execute S and obtain the new environment $E' = EX(S, I, E)$.

5.2 Rewriting IMPL to acthex

Using acthex for realizing IMPL is a natural and reasonable choice because acthex already natively provides several features necessary for IMPL: external atoms can be used to access information from a MCS, and acthex actions come with weights for creating ordered execution schedules for actions occurring within the same answer set of an acthex program. Based on this, IMPL can be implemented by a rewriting to acthex, with acthex actions implementing IMPL actions, acthex external predicates providing information about the MCS to the IMPL policy, and acthex external predicates realizing the value invention builtin predicates.

We next describe a rewriting from the IMPL core language fragment to acthex. We assume that the environment E contains a pair $(\mathcal{A}, \mathcal{R})$ of sets of bridge rules, and an encoding of the MCS M (suitable for an implementation of the external atoms introduced below, e.g., in the syntax used by the MCS-IE system [3], which provide the corresponding policy input). A given IMPL policy P wrt. the MCS M is then rewritten to an acthex program P^{act} as follows.

1. Each core IMPL action $@a(t)$ in the head of a rule of P is replaced by a brave acthex action $\#a[t]\{b, 2\}$ with precedence 2. These acthex actions implement semantics of the respective IMPL actions according to Def. 4: interpretation I and the original action's argument t are used as input, the effects are accumulated as $(\mathcal{A}, \mathcal{R})$ in E.
2. Each IMPL builtin $\#id_k(C, c_1, \ldots, c_k)$ in P is replaced by an acthex external atom $\&id_k[c_1, \ldots, c_k](C)$. The family of external atoms $\&id_k[c_1, \ldots, c_k](C)$ realizes value invention and has as semantics function $f_{\&id_k}(E, I, c_1, \ldots, c_k, C) = 1$ for one constant $C = auxc\_c_1\_\ldots\_c_k$ created from the constants in tuple $c_1, \ldots, c_k$.
3. We add to P^{act} a set P_{in} of acthex rules containing (i) rules that use, for every $p \in C_{res} \setminus \{modset\}$, a corresponding external atom to 'import' a faithful representation of M, and (ii) a preparatory action $\#reset$ with precedence 1, and a final action $\#materialize$ with precedence 3: $P_{in} = \{p(\mathbf{t}) \leftarrow \&p[](\mathbf{t}) \mid p \in C_{res} \setminus \{modset\}\} \cup \{\#reset[]\{b, 1\}; \#materialize[]\{b, 3\}\}$, where $\mathbf{t}$ is a vector of different variables of length equal to the arity of p (i.e., one, two, or three).

The first two steps transform IMPL actions into acthex actions, and $\#id_k$-value invention into external atom calls. The third step essentially creates policy input facts from acthex external sources. External atoms in P_{in} return a representation of M and analyze inconsistency in M, providing minimal diagnoses and minimal explanations. Thus, the respective rules in P_{in} yield an extension of the corresponding reserved predicates which is a faithful representation of M. Moreover, action $\#reset$ resets the modification $(\mathcal{A}, \mathcal{R})$ stored in E to $(\emptyset, \emptyset)$.[5] Action $\#materialize$ materializes the modification $(\mathcal{A}, \mathcal{R})$ (as accumulated by actions of precedence 2) in the MCS M (which is part of E).

Example 14 (*ctd*). Policy P_3 from Ex. 9 translated to acthex contains the following rules $P_3^{act} = P_{in} \cup \{modset(md, X) \leftarrow diag(X); \#guiSelectMod[md]\{b, 2\}\}$. □

Note, that actions in the rewriting have no weights, therefore all answer sets are best models. For obtaining an admissible modification, any policy answer set can be chosen, and any execution schedule can be used.

Proposition 1. *Given a MCS M, a core* IMPL *policy P, and a policy input EDB_M wrt. M, let P^{act} be as above, and consider an environment E containing M and $(\emptyset, \emptyset)$. Then, every execution outcome $E' \in \mathcal{EX}(P^{act} \cup EDB_M|_{B_A}, E)$ contains instead of M an admissible modification M' of M wrt. P and EDB_M.*

The proof of this correctness result can be found in the extended version [15].

6 Conclusion

Related to IMPL is the action language *IMPACT* [26], which is a declarative formalism for actions in distributed and heterogeneous multi-agent systems. *IMPACT* is a very rich general purpose formalism, which however is more difficult to manage compared to the special purpose language IMPL. Furthermore, user interaction as in IMPL is not directly supported in *IMPACT*; nevertheless most parts of IMPL could be embedded in *IMPACT*.

In the fields of access control, e.g., surveyed in [4], and privacy restrictions [13], policy languages have also been studied in detail. As a notable example, $\mathcal{PDL}$ [12] is a declarative policy language based on logic programming which maps events in a system to actions. $\mathcal{PDL}$ is richer than IMPL concerning action interdependencies, whereas actions in IMPL have a richer internal structure than $\mathcal{PDL}$ actions. Moreover, actions in IMPL depend on the content of a policy answer set. Similarly, inconsistency analysis input in IMPL has a deeper structure than events in $\mathcal{PDL}$.

In the context of relational databases, logic programs have been used for specifying repairs for databases that are inconsistent wrt. a set of integrity constraints [14,23,24]. These approaches may be considered fixed policies without user interaction, like an IMPL policy simply applying diagnoses in a homogeneous MCS. Note however, that an important motivation for developing IMPL is the

[5] This reset is necessary if a policy is applied repeatedly.

fact that automatic repair approaches are not always a viable option for dealing with inconsistency in a MCS.

Active integrity constraints (AICs) [9–11] and *inconsistency management policies (IMPs)* [25] have been proposed for specifying repair strategies for inconsistent databases in a more flexible way. AICs extend integrity constraints by introducing update actions, for inserting and deleting tuples, to be performed if the constraint is not satisfied. On the other hand, an IMP is a function which is defined wrt. a set of functional dependencies mapping a given relation R to a 'modified' relation R' obeying some basic axioms.

Although suitable IMPL policy encodings can mimic database repair programs—AICs and (certain) IMPs—for specific classes of integrity constraints, there are fundamental conceptual differences between IMPL and the above approaches to database repair. Most notably, IMPL policies aim at restoring consistency by modifying bridge rules leaving the knowledge bases unchanged rather than considering a set of constraints as fixed and repairing the database. Additionally, IMPL policies operate on heterogeneous knowledge bases and may involve user interaction.

Ongoing and Future Work. Regarding an actual prototype implementation of IMPL, we are currently working on improvements of acthex which are necessary for realizing IMPL using the rewriting technique described in Sect. 5.2. In particular, this includes the generalization of taking into account the environment in external atom evaluation. Other improvements concern the support for implementing model and execution schedule selection functions.

An important feature of IMPL is the user interface for selecting or editing modifications. There the number of displayed modifications might be reduced considerably by grouping modifications according to nonground bridge rules. This would lead to a considerable improvement of usability in practice.

Also, we currently just consider bridge rule modifications for system repairs, therefore an interesting issue for further research is to drop this convention. A promising way to proceed in this direction is to integrate IMPL with recent work on managed MCSs [6], where bridge rules are extended such that they can arbitrarily modify the knowledge base of a context and even its semantics. Accordingly, IMPL could be extended with the possibility of using management operations on contexts in system modifications.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory Implementation and Applications. Cambridge University Press, Cambridge (2003)
2. Basol, S., Erdem, O., Fink, M., Ianni, G.: HEX programs with action atoms. In: ICLP, pp. 24–33 (2010)
3. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The MCS-IE system for explaining inconsistency in multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 356–359. Springer, Heidelberg (2010)

4. Bonatti, P.A., De Coi, J.L., Olmedilla, D., Sauro, L.: Rule-based policy pepresentations and reasoning. In: Bry, F., Małuszyński, J. (eds.) Semantic Techniques for the Web. LNCS, vol. 5500, pp. 201–232. Springer, Heidelberg (2009)
5. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI Conference on Artificial Intelligence (AAAI), pp. 385–390 (2007)
6. Brewka, G., Eiter, T., Fink, M., Weinzierl, A.: Managed multi-context systems. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 786–791 (2011)
7. Brewka, G., Roelofsen, F., Serafini, L.: Contextual default reasoning. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 268–273 (2007)
8. Buccafurri, F., Leone, N., Rullo, P.: Strong and weak constraints in disjunctive datalog. In: Dix, J., Furbach, U., Nerode, A. (eds.) Logic Programming and Nonmonotonic Reasoning. LNCS, vol. 1265, pp. 2–17. Springer, Heidelberg (1997)
9. Caroprese, L., Greco, S., Zumpano, E.: Active integrity constraints for database consistency maintenance. IEEE Trans. Knowl. Data Eng. **21**(7), 1042–1058 (2009)
10. Caroprese, L., Truszczyński, M.: Declarative semantics for active integrity constraints. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 269–283. Springer, Heidelberg (2008)
11. Caroprese, L., Truszczyński, M.: Declarative semantics for revision programming and connections to active integrity constraints. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 100–112. Springer, Heidelberg (2008)
12. Chomicki, J., Lobo, J., Naqvi, S.A.: A logic programming approach to conflict resolution in policy management. In: KR, pp. 121–132 (2000)
13. Duma, C., Herzog, A., Shahmehri, N.: Privacy in the semantic web: what policy languages have to offer. In. POLICY, pp. 109–118 (2007)
14. Eiter, T., Fink, M., Greco, G., Lembo, D.: Repair localization for query answering from inconsistent databases. ACM Trans. Database Syst. **33**(2), 10:01–10:51 (2008)
15. Eiter, T., Fink, M., Ianni, G., Schüller, P.: Managing inconsistency in multi-context systems using the IMPL policy language. Tech. Rep. INFSYS RR-1843-12-05, Vienna University of Technology, Institute for, Information Systems (2012)
16. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in nonmonotonic multi-context systems. In: KR, pp. 329–339 (2010)
17. Eiter, T., Fink, M., Weinzierl, A.: Preference-based inconsistency assessment in multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 143–155. Springer, Heidelberg (2010)
18. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI, pp. 90–96. Pofessional Book Center, Denver (2005)
19. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. **175**(1), 278–298 (2011)
20. Fink, M., Ghionna, L., Weinzierl, A.: Relational information exchange and aggregation in multi-context systems. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 120–133. Springer, Heidelberg (2011)
21. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991)
22. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: how we can do without modal logics. Artif. Intell. **65**(1), 29–70 (1994)
23. Greco, G., Greco, S., Zumpano, E.: A logical framework for querying and repairing inconsistent databases. IEEE Trans. Knowl. Data Eng. **15**(6), 1389–1408 (2003)

24. Marileo, M.C., Bertossi, L.E.: The consistency extractor system: answer set programs for consistent query answering in databases. Data Knowl. Eng. **69**(6), 545–572 (2010)
25. Martinez, M.V., Parisi, F., Pugliese, A., Simari, G.I., Subrahmanian, V.S.: Inconsistency management policies. In: KR, pp. 367–377 (2008)
26. Subrahmanian, V., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Ozcan, F., Ross, R.: Heterogeneous Agent Systems: Theory and Implementation. MIT Press, Cambridge (2000)

The Parameterized Complexity of Constraint Satisfaction and Reasoning

Stefan Szeider(✉)

Vienna University of Technology, A-1040 Vienna, Austria
stefan@szeider.net

Abstract. Parameterized Complexity is a new and increasingly popular theoretical framework for the rigorous analysis of NP-hard problems and the development of algorithms for their solution. The framework provides adequate concepts for taking structural aspects of problem instances into account. We outline the basic concepts of Parameterized Complexity and survey some recent parameterized complexity results on problems arising in Constraint Satisfaction and Reasoning.

1 Introduction

Computer science has been quite successful in devising fast algorithms for important computational tasks, for instance, to sort a list of items or to match workers to machines. By means of a theoretical analysis one can guarantee that the algorithm will always find a solution quickly. Such a worst-case performance guarantee is the ultimate aim of algorithm design. The traditional theory of algorithms and complexity as developed in the 1960s and 1970s aims at performance guarantees in terms of one dimension only, the input size of the problem. However, for many important computational problems that arise from real-world applications, the traditional theory cannot give reasonable (i.e., polynomial) performance guarantees. The traditional theory considers such problems as *intractable*. Nevertheless, heuristics-based algorithms and solvers work surprisingly well on real-world instances of such problems. Take for example the satisfiability problem (Sat) of propositional reasoning. No algorithm is known that can solve a Sat instance on n variables in $2^{o(n)}$ steps (by the widely believed Exponential Time Hypothesis such an algorithm is impossible [29]). On the other hand, state-of-the-art Sat solvers solve routinely instances with hundreds of thousands of variables in a reasonable amount of time (see e.g., [23]). Hence there is an enormous gap between theoretical performance guarantees and the empirically observed performance of solvers. This gap separates theory-oriented and applications-oriented research communities. This theory-practice gap has been

Research supported by the European Research Council, grant reference 239962 (COMPLEX REASON). This survey was written in 2011, some references where updated in 2013.

observed by many researchers, including Moshe Vardi, who closed an editorial letter [49] as follows:

> [...] an important role of theory is to shed light on practice, and there we have large gaps. We need, I believe, a richer and broader complexity theory, a theory that would explain both the difficulty and the easiness of problems like SAT. More theory, please!

Parameterized Complexity is a new theoretical framework that offers a great potential for reducing the theory-practice gap. The key idea is to consider—in addition to the input size—a secondary dimension, the parameter, and to design and analyse algorithms in this two-dimensional setting. Virtually in every conceivable context we know more about the input data than just its size in bytes. The second dimension (the parameter) can represent this additional information. This two-dimensional setting gives raise to a foundational theory of algorithms and complexity that can be closer to the problems as they appear in the real world. Parameterized Complexity has been introduced and pioneered by R. Downey and M. R. Fellows [8] and is receiving growing interest as reflected by hundreds of research papers (see the references in [8,15,37]). In more and more research areas such as Computational Biology, Computational Geometry, and Computational Social Choice the merits of Parameterized Complexity become apparent (see, e.g., [2,22,26]).

2 Parameterized Complexity: Basic Concepts and Definitions

In the following we outline the central concepts of Parameterized Complexity.

An instance of a parameterized problem is a pair (I, k) where I is the *main part* and k is the *parameter*; the latter is usually a non-negative integer. The central notion of the field is *fixed-parameter tractability* (FPT) which refers to solvability in time $f(k)n^c$, where f is some (possibly exponential) function of the parameter k, c is a constant, and n is the size of the instance with respect to some reasonable encoding. An algorithm that runs in time $f(k)n^c$ is called an *fpt-algorithm*. As a consequence of this definition, a fixed-parameter tractable problem can be solved in polynomial time for any fixed value of the parameter, and, importantly, the order of the polynomial does not depend on the parameter. This is significantly different from problems that can be solved in, say, time n^k, which also gives polynomial-time solvability for each fixed value of k, but since the order of the polynomial depends on k it does not scale well in k and quickly becomes inefficient for small values of k.

Take for example the Vertex Cover problem: Given a graph G and an integer k, the question is whether there is a set of k vertices such that each edge of G has at least one of its ends in this set. The problem is NP-complete, but fixed-parameter tractable for parameter k. Currently the best known fpt-algorithm for this problem runs in time of order $1.2738^k + kn$ [6]. This algorithm is practical for huge instances as long as the parameter k is below 100. The situation is dramatically different for the Independent Set problem, where for a given graph G and an integer k it is asked whether there is a set of k vertices of G such that no edge joints two vertices in the set. Also this problem is NP-complete, and indeed for traditional complexity the problems Vertex Cover and Independent Set are essentially the same, as there is a trivial polynomial-time transformation from one problem to the other (the complement set of a vertex cover is an independent set). However, no fixed-parameter algorithm for Independent Set is known and the Parameterized Complexity of this problem appears to be very different from the complexity of Vertex Cover. Theoretical evidence suggests that Independent Set cannot be solved significantly faster than by trying all subsets of size k, which gives a running time of order n^k.

The subject of Parameterized Complexity splits into two complementary questions, each with its own mathematical toolkit and methods:

1. How to design and improve fixed-parameter algorithms for parameterized problems. For this question there exists a rich *toolkit of algorithmic techniques* (see, e.g., [46]).
2. How to gather evidence that a parameterized problem is not fixed-parameter tractable. For this question a *completeness theory* has been developed which is similar to the theory of NP-completeness (see, e.g., [7]) and supports the accumulation of strong theoretical evidence that a parameterized problem is *not* fixed-parameter tractable.

Every completeness theory requires a suitable notion of reduction. The classical polynomial-time reductions are not suitable for Parameterized Complexity, as they do not differentiate between problems that are fixed-parameter tractable and problems that are believed to be not (such as Vertex Cover and Independent Set, respectively, as discussed above). A reduction that preserves fixed-parameter tractability must ensure that the parameter of one problem maps to the parameter of the other problem. This is the case for *fpt-reductions*, the standard reductions in Parameterized Complexity. An fpt-reduction between parameterized decision problems P and Q is an fpt-algorithm that maps a problem instance (x, k) of P to a problem instance (x', k') of Q such that (i) (x, k) is a yes-instance of P if and only if (x', k') is a yes-instance of Q, and (ii) there is a computable function g such that $k' \leq g(k)$. It is easy to see that if we have an fpt-reduction from P to Q, and Q is fixed-parameter tractable, then so is P.

3 How to Parameterize?

Most research in Parameterized Complexity considers optimization problems, where the parameter is a bound on the objective function, also called solution size. For instance, the standard parameter for Vertex Cover is the size of the vertex cover we are looking for. However, many problems that arise in Constraint Satisfaction and Reasoning are not optimization problems, and it seems more natural to consider parameters that indicate the presence of a “hidden structure” in the problem instance. It is a widely accepted view that efficient solvers exploit the hidden structure of real-world problems. Hence such a parameter can be used to capture the presence of a hidden structure. There are several approaches to making the vague notion of a hidden structure mathematically precise in terms of a parameter.

3.1 Backdoors

If a computational problem is intractable in general, it is natural to ask for subproblems for which the problem is solvable in polynomial-time, and indeed much research has been devoted to this question. Such tractable subproblems are sometimes called "islands of tractability" or "tractable fragments." It seems unlikely that a problem instance originating from a real-world application belongs to one of the known tractable fragments, but it might be close to one. The concept of *backdoor sets* offers a generic way to gradually enlarge and extend an island of tractability and thus to solve problem instances efficiently if they are close to a tractable fragment. The size of a smallest backdoor set indicates the distance between an instance and a tractable fragment. Backdoor sets were introduced in the context of propositional and constraint-based reasoning [50] but similar notions can be defined for other reasoning problems. Roughly speaking, after eliminating the variables of a backdoor set one is left with an instance that belongs to the tractable subproblem under consideration. The "backdoor approach" to reasoning problems involves two tasks. The first task, called *backdoor detection*, is to detect a small backdoor set by an fpt-algorithm, parameterized by the size of the backdoor set. The second task, called *backdoor evaluation*, is to solve the reasoning problem efficiently using the information provided by the backdoor set.

There are several Parameterized Complexity results on backdoor sets for the SAT problem as described in a recent survey [21]. Backdoors have also been applied to problems beyond NP such as Model Counting and QBF-Satisfiability [38,42], and to the main reasoning problems of propositional disjunctive answer set programming (deciding whether an atom belongs to some stable model or whether it belongs to all stable models). The latter problems are located at the second level of the Polynomial Hierarchy [11] but can be solved in polynomial time for normal (disjunction-free) programs that have certain acyclicity properties. Several of these tractable classes admit a backdoor approach, with fixed-parameter tractable backdoor detection and backdoor evaluation, thus rendering the answer-set programming problems fixed-parameter tractable [13]. A similar backdoor approach has also been developed for problems of abstract argumentation [10] whose unparameterized versions are also located at the second level of the Polynomial Hierarchy.

3.2 Decompositions

A key technique for coping with hard computational problems is to decompose the problem instance into small tractable parts, and to reassemble the solutions of the parts to a solution of the entire instance. One aims at decompositions for which the overall complexity depends on how much the parts overlap, the "width" of the decomposition. The most popular and widest studied decomposition method is *tree decomposition* with the associated parameter *treewidth.* A recent survey by Hlinený et al. covers several decomposition methods with particular focus on fixed-parameter tractability [28].

Recent results on the Parameterized Complexity of reasoning problems with respect to decomposition width include results on disjunctive logic programming and answer-set programming with weight constraints [24,41], abductive reasoning [25], satisfiability and propositional model counting [39,44], constraint satisfaction and global constraints [43,45], and abstract and value-based argumentation [9,30].

3.3 Locality

Practical algorithms for hard reasoning problems are often based on *local search* techniques. The basic idea is to start with an arbitrary candidate solution and to try to improve it step by step, at each step moving from one candidate solution to a better "neighbor" candidate solution. It would provide an enormous speed-up if one could perform k elementary steps of local search efficiently in one "giant" k-step. Such a giant k-step also decreases the probability of getting stuck at a poor local optimum. However, the obvious strategy for performing one giant k-step requires time of order N^k (assuming a candidate solution has N neighbour solutions), which is impractical already for very small values of k since typically N is related to the input size. A challenging objective is the design of fpt-algorithms (with respect to parameter k) that compute a giant k-step. Recent work on parameterized local search includes the problem of minimizing the Hamming weight of satisfying assignments for Boolean CSP [31] and for the Max Sat problem [48]. It turns out that there are interesting cases for which k-step local search is fixed-parameter tractable.

Local consistency is a further form of locality that plays an important role in constraint satisfaction and is one of the oldest and most fundamental concepts of in this area. It can be traced back to Montanari's famous 1974 paper [36]. If a constraint network is locally consistent, then consistent instantiations to a small number of variables can be consistently extended to any further variable. Hence local consistency avoids certain dead-ends in the search tree, in some cases it even guarantees backtrack-free search [1,18]. The simplest and most widely used form of local consistency is arc-consistency, introduced by Mackworth [33], and later generalized to k-consistency by Freuder [17]. A constraint network is k*-consistent* if each consistent assignment to $k-1$ variables can be consistently extended to any further k-th variable. It is a natural question to ask for the Parameterized Complexity of checking whether a constraint network is k-consistent, taking k as the parameter. This question has been subject to a recent study [20]. It turned out that in general, deciding whether a constraint network is k-consistent is complete for the parameterized complexity class co-W[2] and thus unlikely to be fixed-parameter tractable. However, if we include as secondary parameters the maximum domain size and the maximum number of constraints in which a variable occurs, then the problem becomes fixed-parameter tractable.

3.4 Above or Below Guaranteed Bounds

For some optimization problems that arise in constraint satisfaction and reasoning, the standard parameter (solution size) is not a very useful one. Take for instance the problem Max Sat. The standard parameter is the number of satisfied clauses. However, it is well-known that one can always satisfy at least half of the clauses. Hence, if we are given m clauses, and if we want to satisfy at least k of them, then the answer is clearly *yes* if $k \leq m/2$. On the other hand, if $k > m/2$ then $m < 2k$, hence the size of the given formula is bounded in terms of the parameter k, and thus can be trivially solved by brute force in time that only depends on k. Less trivial is the question of whether we can satisfy at least $m/2 + k$ clauses, where k is the parameter. Such a problem is called *parameterized above a guaranteed value* [34,35]. Over the last few years, several variants of Max Sat but also optimization problems regarding ordering constraints have been studied, parameterized above a guaranteed value. A recent survey by Gutin and Yeo covers these results [27].

4 Kernelization: Preprocessing with Guarantee

Preprocessing and data reduction are powerful ingredients of virtually every practical solver. Before performing a computationally expensive case distinction, it seems always better to seek for a "safe step" that simplifies the instance, and to preprocess. Indeed, the success of practical solvers relies often on powerful preprocessing techniques. However, preprocessing has been neglected by traditional complexity theory: if we measure the complexity of a problem just in terms of the input size n, then reducing the size from n to $n-1$ in polynomial time yields a polynomial-time algorithm for the problem as we can iterate the reduction [12]. Hence it does not make much sense to study preprocessing for NP-hard problems in the traditional one-dimensional framework. However, the notion of "kernelization", a key concept of Parameterized Complexity provides the means for studying preprocessing, since the impact of preprocessing can measured in terms of the parameter, not the size of the input. When a problem is fixed-parameter tractable then each instance (I, k) can be reduced in polynomial time to an equivalent instance (I', k'), the *problem kernel*, where $k' \leq k$ and the size of I' is bounded by a function of k. The smaller the kernel, the more efficient the fixed-parameter algorithm. For a parameterized problem it is therefore interesting to know whether it admits a *polynomial kernel* or not. Over the last few years, this question has received a lot of attention in the Parameterized Complexity community [32].

Several optimization problems, such as Vertex Cover and Feedback Vertex Set admit polynomial kernels with respect to the standard parameter solution size [5,6]. However, it turns out that many fixed-parameter tractable problems in the areas of constraint satisfaction, global constraints, satisfiability, nonmonotonic and Bayesian reasoning do not have polynomial kernels unless the Polynomial Hierarchy collapses to its third level [47]. Such super-polynomial kernel lower bounds can be obtained by means of recent tools [4,16]. A positive exception is the consistency problem for certain global constraint, which admits a polynomial kernel for an interesting parameter [19].

5 Breaking Complexity Barriers with FPT-Reductions

Many important problems in constraint satisfaction and reasoning are located above the first level of the Polynomial Hierarchy or are even PSpace-complete, thus considered "harder" than the Sat problem. Above we have discussed some results that establish fixed-parameter tractability for such problems (including ASP problems and QBF satisfiability, parameterized by backdoor size). However, for such hard problems, asking for fixed-parameter tractability is asking for a lot and requires the parameters to be quite restrictive. Therefore, it seems to be an even more interesting approach to exploit some structural properties of the instance in terms of a parameter, not to solve the instance, but to reduce it to an equivalent instance of a problem of lower classical complexity. The parameter can thus be less restrictive and can therefore be small for larger classes of inputs. The reduction cannot run in polynomial time, unless the Polynomial Hierarchy collapses, but the enhanced power of *fpt-reductions* (see Sect. 2) can break the barriers between classical complexity classes. The Sat problem is well-suited as a target problem (say, with the constant parameter 0), since by means of fpt-reductions to Sat we can make today's extremely powerful Sat solvers applicable to problems on higher levels of the Polynomial Hierarchy. In fact, there are some known reductions that, in retrospect, can be seen as fpt-reductions to SAT. A prominent example is *Bounded Model Checking* [3], a technique of immense practical significance for hardware and software verification, which can be seen as an fpt-reduction from the PSpace-complete model checking problem for linear temporal logic to Sat. The parameter is an upper bound on the size of a counterexample (or the diameter of the instance).

In recent work [14] we have developed fpt-reductions that break complexity barriers for the main reasoning problems of disjunctive answer-set programming. These problems are located at the second level of the Polynomial Hierarchy in general, but drop back to the first level if restricted to normal (i.e., disjunction-free) programs. Thus, it is natural to consider as a parameter the *distance of a disjunctive program form being normal*; the backdoor size with respect to the base class of normal programs provides such a distance measure. And indeed, there is an fpt-reduction with respect to this parameter, that takes as input a disjunctive program P and an atom x, and outputs a CNF formula that is satisfiable if and only if x is in some answer set of P (a similar fpt-reduction outputs a CNF formula that is unsatisfiable if and only if x is in all answer sets of P). In terms of parameterized complexity, this shows that the brave reasoning problem for disjunctive ASP is paraNP-complete; paraNP is the class of all parameterized decision problems that can be solved in time $f(k)n^c$ by a *nondeterministic* algorithm [15]. For parameterizations of NP-problems, a paraNP-completeness result is considered as very negative. For a problem that is harder than NP, however, a paraNP-completeness result is a positive one, as it shows that the structure represented by the parameter can be exploited to break the complexity barrier. Very recently, we developed similar fpt-reductions for problems arising in propositional abductive reasoning which are also located on the second level of the Polynomial Hierarchy, taking as parameters the distance of the input theory form being Horn or being Krom [40].

6 Conclusion

Over the last decade, Parameterized Complexity has become an important field of research in Algorithms and Complexity. It provides a more fine-grained complexity analysis than the traditional theory taking structural aspects of problem instances into account. In this brief survey we have outlined the basic concepts of Parameterized Complexity and indicated some recent results on the Parameterized Complexity of problems arising in Constraint Satisfaction an Reasoning.

References

1. Atserias, A., Bulatov, A., Dalmau, V.: On the power of k-consistency. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 279–290. Springer, Heidelberg (2007)
2. Betzler, N., Bredereck, R., Chen, J., Niedermeier, R.: Studies in computational aspects of voting- a parameterized complexity perspective. In: Bodlaender, H.L., Downey, R., Fomin, F.V., Marx, D. (eds.) Fellows Festschrift 2012. LNCS, vol. 7370, pp. 318–363. Springer, Heidelberg (2012)
3. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
4. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On problems without polynomial kernels (extended abstract). In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 563–574. Springer, Heidelberg (2008)
5. Cao, Y., Chen, J., Liu, Y.: On feedback vertex set new measure and new structures. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 93–104. Springer, Heidelberg (2010)
6. Chen, J., Kanj, I.A., Xia, G.: Improved upper bounds for vertex cover. Theoret. Comput. Sci. **411**(40–42), 3736–3756 (2010)
7. Chen, J., Meng, J.: On parameterized intractability: hardness and completeness. Comput. J. **51**(1), 39–59 (2008)
8. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science. Springer, New York (1999)
9. Dunne, P.E.: Computational properties of argument systems satisfying graph-theoretic constraints. Artif. Intell. **171**(10–15), 701–729 (2007)
10. Dvorák, W., Ordyniak, S., Szeider, S.: Augmenting tractable fragments of abstract argumentation. Artif. Intell. **186**, 157–173 (2012)

11. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: propositional case. Ann. Math. Artif. Intell. **15**(3–4), 289–323 (1995)
12. Fellows, M.R.: The lost continent of polynomial time: preprocessing and kernelization. In: Bodlaender, H.L., Langston, M.A. (eds.) IWPEC 2006. LNCS, vol. 4169, pp. 276–277. Springer, Heidelberg (2006)
13. Fichte, J.K., Szeider, S.: Backdoors to tractable answer-set programming. Technical Report 1104.2788, Arxiv.org. Extended and updated version of a paper that appeared in the proceedings of IJCAI 2011, The 22nd International Joint Conference on Artificial Intelligence (2012)
14. Fichte, J.K., Szeider, S.: Backdoors to normality for disjunctive logic programs. In: des Jardins, M., Littman, M.L. (eds.) Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013), Bellevue, Washington, USA, 14–18 July 2013, pp. 320–337. AAAI Press, California (2013)
15. Flum, J., Grohe, M.: Parameterized Complexity Theory, vol. XIV. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2006)
16. Fortnow, L., Santhanam, R.: Infeasibility of instance compression and succinct PCPs for NP. In: Dwork, C. (ed.) Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, 17–20 May 2008, pp. 133–142. ACM, New York (2008)
17. Freuder, E.C.: Synthesizing constraint expressions. Commun. ACM **21**(11), 958–966 (1978)
18. Freuder, E.C.: A sufficient condition for backtrack-bounded search. J. ACM **32**(4), 755–761 (1985)
19. Gaspers, S., Szeider, S.: Kernels for global constraints. In: Walsh, T. (ed.) Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011, pp. 540–545. AAAI Press, California (2011)
20. Gaspers, S., Szeider, S.: The parameterized complexity of local consistency. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 302–316. Springer, Heidelberg (2011)
21. Gaspers, S., Szeider, S.: Backdoors to satisfaction. In: Bodlaender, H.L., Downey, R., Fomin, F.V., Marx, D. (eds.) Fellows Festschrift 2012. LNCS, vol. 7370, pp. 287–317. Springer, Heidelberg (2012)
22. Giannopoulos, P., Knauer, C., Whitesides, S.: Parameterized complexity of geometric problems. Comput. J. **51**(3), 372–384 (2008)
23. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability solvers. In: van Harmelen, F., Lifschitz, V. (eds.) Handbook of Knowledge Representation, vol. 3, Foundations of Artificial Intelligence, pp. 89–134. Elsevier, Amsterdam (2008)
24. Gottlob, G., Pichler, R., Wei, F.: Bounded treewidth as a key to tractability of knowledge representation and reasoning. In: 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference. AAAI Press (2006)
25. Gottlob, G., Pichler, R., Wei, F.: Abduction with bounded treewidth: from theoretical tractability to practically efficient computation. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), Chicago, Illinois, USA, 13–17 July 2008, pp. 1541–1546. AAAI Press, California (2008)
26. Gramm, J., Nickelsen, A., Tantau, T.: Fixed-parameter algorithms in phylogenetics. Comput. J. **51**(1), 79–101 (2008)
27. Gutin, G., Yeo, A.: Constraint satisfaction problems parameterized above or below tight bounds: a survey. In: Bodlaender, H.L., Downey, R., Fomin, F.V., Marx, D. (eds.) Fellows Festschrift 2012. LNCS, vol. 7370, pp. 257–286. Springer, Heidelberg (2012)

28. Hlinený, P., Oum, S., Seese, D., Gottlob, G.: Width parameters beyond tree-width and their applications. Comput. J. **51**(3), 326–362 (2008)
29. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? J. Comput. Syst. Sci. **63**(4), 512–530 (2001)
30. Kim, E.J., Ordyniak, S., Szeider, S.: Algorithms and complexity results for persuasive argumentation. Artif. Intell. **175**, 1722–1736 (2011)
31. Krokhin, A., Marx, D.: On the hardness of losing weight. ACM Trans. Algorithm **8**(2), 19 (2012)
32. Lokshtanov, D., Misra, N., Saurabh, S.: Kernelization – preprocessing with a guarantee. In: Bodlaender, H.L., Downey, R., Fomin, F.V., Marx, D. (eds.) Fellows Festschrift 2012. LNCS, vol. 7370, pp. 129–161. Springer, Heidelberg (2012)
33. Mackworth, A.K.: Consistency in networks of relations. Artif. Intell. **8**, 99–118 (1977)
34. Mahajan, M., Raman, V.: Parameterizing above guaranteed values: MaxSat and MaxCut. J. Algorithms **31**(2), 335–354 (1999)
35. Mahajan, M., Raman, V., Sikdar, S.: Parameterizing MAX SNP problems above guaranteed values. In: Bodlaender, H.L., Langston, M.A. (eds.) IWPEC 2006. LNCS, vol. 4169, pp. 38–49. Springer, Heidelberg (2006)
36. Montanari, U.: Networks of constraints: fundamental properties and applications to picture processing. Inf. Sci. **7**, 95–132 (1974)
37. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, Oxford (2006)
38. Nishimura, N., Ragde, P., Szeider, S.: Solving #SAT using vertex covers. Acta Informatica **44**(7–8), 509–523 (2007)
39. Ordyniak, S., Paulusma, D., Szeider, S.: Satisfiability of acyclic and almost acyclic cnf formulas. Theor. Comput. Sci. **481**, 85–99 (2013)
40. Pfandler, A., Rümmele, S., Szeider, S.: Backdoors to absuction. In: Proceedings of the 23th International Joint Conference on Artificial Intelligence (IJCAI 2013), Beijing, China, 3–9 August 2013 (2013) (to appear)
41. Pichler, R., Rümmele, S., Szeider, S., Woltran, S.: Tractable answer-set programming with weight constraints: bounded treewidth is not enough. In: Lin, F., Sattler, U., Truszczynski, M. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference (KR 2010), Toronto, Ontario, Canada, 9–13 May 2010. AAAI Press, California (2010)
42. Samer, M., Szeider, S.: Backdoor sets of quantified Boolean formulas. J. Autom. Reason. **42**(1), 77–97 (2009)
43. Samer, M., Szeider, S.: Tractable cases of the extended global cardinality constraint. Constraints **16**(1), 1–24 (2009)
44. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discret. Algorithms **8**(1), 50–64 (2010)
45. Samer, M., Szeider, S.: Constraint satisfaction with bounded treewidth revisited. J. Comput. Syst. Sci. **76**(2), 103–114 (2010)
46. Sloper, C., Telle, J.A.: An overview of techniques for designing parameterized algorithms. Comput. J. **51**(1), 122–136 (2008)
47. Szeider, S.: Limits of preprocessing. Proceedings of the twenty-fifth conference on artificial intelligence, AAAI 2011, pp. 93–98. AAAI Press, California (2011)
48. Szeider, S.: The parameterized complexity of k-flip local search for SAT and MAX SAT. Discrete Optim. **8**(1), 139–145 (2011)
49. Vardi, M.Y.: On P, NP, and computational complexity. Commun. ACM **53**(11), 5 (Nov. 2010)

50. Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. In: Gottlob, G., Walsh, T. (eds.) Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003, pp. 1173–1178. Morgan Kaufmann, San Francisco (2003)

INAP Technical Papers I: Languages

Translating Nondeterministic Functional Language Based on Attribute Grammars into Java

Masanobu Umeda[1(✉)], Ryoto Naruse[2], Hiroaki Sone[2], and Keiichi Katamine[1]

[1] Kyushu Institute of Technology, 680-4 Kawazu, Iizuka 820-8502, Japan
umerin@ci.kyutech.ac.jp, katamine@ci.kyutech.ac.jp
[2] NaU Data Institute Inc., 680-41 Kawazu, Iizuka 820-8502, Japan
naruse@nau.co.jp, sone@nau.co.jp

Abstract. Knowledge-based systems are suitable for realizing advanced functions that require domain-specific expert knowledge, while knowledge representation languages and their supporting environments are essential for realizing such systems. Although Prolog is both useful and effective in this regard, the language interoperability with other implementation languages, such as Java, is often critical in practical application development. This paper describes the techniques for translating a knowledge representation language, a nondeterministic functional language based on attribute grammars, into Java. Translation is based on binarization and current techniques for Prolog to Java translation although the semantics are different from those of Prolog. A continuation unit is introduced to handle continuation efficiently, while variable and register management for backtracking is simplified by exploiting the single and unidirectional assignment features of variables. Experimental results for several benchmarks show that the code generated by the prototype translator is more than 25 times and 2 times faster than that of Prolog Cafe for nondeterministic programs and deterministic programs, respectively, and more than 2 times faster than B-Prolog for nondeterministic programs.

Keywords: Knowledge representation language · Language translation · Prolog · Java

1 Introduction

As information and communication technology penetrates more deeply into our society, demand for advanced information services in various application domains such as medical services and supply-chain management is growing. Clinical decision support [1,2] to prevent medical errors and order placement support for optimal inventory management [3] are two typical examples of such systems. It is, however, not prudent to implement these advanced functions as part of a traditional information system using conventional programming languages. This is because expert knowledge is often large-scale and complicated, and each application domain typically has its own specific structures and semantics of knowledge. Therefore, not only the analysis, but also the description, audit, and maintenance of such knowledge are often difficult without expertise in the application domain. It is thus essential to realize such advanced functions to allow domain experts themselves to describe, audit, and maintain their knowledge. A knowledge-based system approach is suitable for this purpose because a suitable framework for representing and managing expert knowledge is provided.

Previously, Nagasawa et al. proposed the knowledge representation language DSP [4,5] and its supporting environment. DSP is a nondeterministic functional language based on attribute grammars [6,7] and is suitable for representing complex search problems without relying on any side-effects. The supporting environment has been developed on top of an integrated development environment called Inside Prolog [8]. Inside Prolog provides standard Prolog functionality, conforming to ISO/IEC 13211-1 [9], as well as a large variety of application programming interfaces that are essential for practical application development, and multi-thread capability for enterprise use [10].

These features allow the consistent development of knowledge-based systems from prototypes to practical systems for both stand-alone and enterprise use [11]. Such systems have been applied to practical applications such as clinical decision support and customer management support, and the effectiveness thereof has been verified. However, several issues have arisen from these experiences. One is the complexity of combining a Prolog-based system with one written in a normal procedural language, such as Java. Another issue is the adaptability to new computing environments, such as mobile devices.

This paper describes the implementation techniques required to translate a nondeterministic functional language based on attribute grammars into a procedural language such as Java. The proposed techniques are based on techniques for Prolog to Java translation. Section 2 gives an overview of the knowledge representation language DSP, and explains how it differs from Prolog. In Sect. 3, the translation techniques for logic programming languages are briefly reviewed, and basic ideas useful for the translation of DSP identified. Section 4 discusses the program representations of DSP in Java, while Sect. 5 evaluates the performance using an experimental translator.

2 Overview of Knowledge Representation Language DSP

2.1 Background

In the development of a knowledge-based system, it is essential to formally analyze, systematize, and describe the knowledge of an application domain. The description of knowledge is conceptually possible in any conventional programming language. Nevertheless, it is difficult to describe, audit, and maintain a knowledge base using a procedural language such as Java. This is because the knowledge of an application domain is often large-scale and complicated, and each application domain has its own specific structures and semantics of knowledge. In particular, the audit and maintenance of written knowledge is a major issue in an information system involving expert knowledge, because such a system cannot easily be changed and the transfer of expert knowledge to succeeding generations is difficult [12]. Therefore, it is very important to provide a framework to enable domain experts themselves to describe, audit, and maintain their knowledge included in an information system [13]. It is perceived that a description language that is specific to an application domain and designed to be described by domain experts is superior in terms of the minimality, constructability, comprehensibility, extensibility, and formality of the language [14]. For this reason, Prolog cannot be considered as a candidate for a knowledge representation language even though it can be an implementation language.

DSP is a knowledge representation language based on nondeterministic attribute grammars. It is a functional language with a search capability using the generate and test method. Because the language is capable of representing trial and error without any side-effects or loop constructs, and the knowledge descriptions can be declaratively read and understood, it is suitable for representing domain-specific expert knowledge involving search problems.

2.2 Syntax and Semantics of DSP

The program unit to represent knowledge in DSP is called a "module", and it represents a nondeterministic function involving no side-effects. Inherited attributes, synthesized attributes, and tentative variables for the convenience of program description, all of which are called variables, follow the single assignment rule, and the assignment is unidirectional. Therefore, the computation process of a module can be represented as non-cyclic dependencies between variables.

Table 1 gives some typical statements in the language. In this table, the types, namely, generator, calculator, and tester, are functional classifications in the generate and test method. Generators `for(B,E,S)` and `select(L)` are provided as primitives for the convenience of knowledge representation although they can be defined as modules using the nondeterministic features of the language.

Table 1. Typical statements in the DSP language

Type	Statement	Function
Generator	`for(B,E,S)`	Assume a numeric value from B to E with step S
Generator	`select(L)`	Assume one of the elements of list L
Generator	`call(M,I,O)`	Call a module M nondeterministically with inputs I and outputs O
Calculator	`dcall(M,I,O)`	Call a module M deterministically with inputs I and outputs O
Calculator	`find(M,I,OL)`	Create a list OL of all outputs of module M with inputs I
Tester	`when(C)`	Specify the domain C of a method
Tester	`test(C)`	Specify the constraint C of a method
Tester	`verify(C)`	Specify the verification condition C

```
pointInQuarterCircle({R : real},              --(a)
                     {X : real, Y : real}) --(b)
  method
    X : real = for(0.0, R, 1.0);            --(c)
    Y : real = for(0.0, R, 1.0);            --(d)
    D : real = sqrt(X^2 + Y^2);             --(e)
    test(D =< R);                           --(f)
  end method;
end module;
```

Fig. 1. Module `pointInQuarterCircle`, which enumerates all points in a quarter circle

Both `call(M,I,O)` and `dcall(M,I,O)` are used for module decomposition, with the latter restricting the first solution of a module call like `once/1` in Prolog,[1] while the former calls a module nondeterministically. Calculator `find(M,I,OL)` collects all outputs of a module and returns a list thereof. Testers `when(C)` and `test(C)` are used to represent decomposition conditions. Both behave in the same way in normal execution mode,[2] although the former is intended to describe a guard of a method, while the latter describes a constraint. Tester `verify(C)`, which describes a verification condition, does not affect the execution of a module despite being classified as a tester. Solutions in which a verification condition is not satisfied are indicated as such, and this verification status is used to evaluate the inference results.

Figure 1 gives the code for module `pointInQuarterCircle`, which enumerates all points in a quarter circle with radius `R`. Statements (a) and (b) in Fig. 1 define the input and output variables of module `pointInQuarterCircle`, respectively. Statements (c) and (d) define the values of variables `X` and `Y` as being between `0.0` and `R` with an incremental step `1.0`. Statement (e) calculates the distance `D` between point `(0,0)` and point `(X,Y)`. Statement (f) checks if point `(X,Y)` is within the circle of radius `R`. Module `pointInQuarterCircle` runs nondeterministically for a given `R`, and returns one of all possible {`X,Y`} values.[3] Therefore, this module also behaves like a generator. Statements (c) to (f) can be listed in any order, and are executed according to the dependencies between variables. Therefore, the computation process can be described as a non-cyclic data flow. Figure 2 shows the data flow diagram for module `pointInQuarterCircle`. Because no module includes any side-effects, the set of points returned by the module is always the same for the same input.

Figure 3 shows an example of module `for`, which implements the generator primitive `for`. If multiple methods are defined in a module with some overlap in their domains specified by `when`, the module works nondeterministically, and

[1] `dcall` stands for deterministic call.

[2] Failures of `when(C)` and `test(C)` are treated differently in debugging mode owing to their semantic differences.

[3] {`X,Y`} represents a vector of two elements `X` and `Y`.

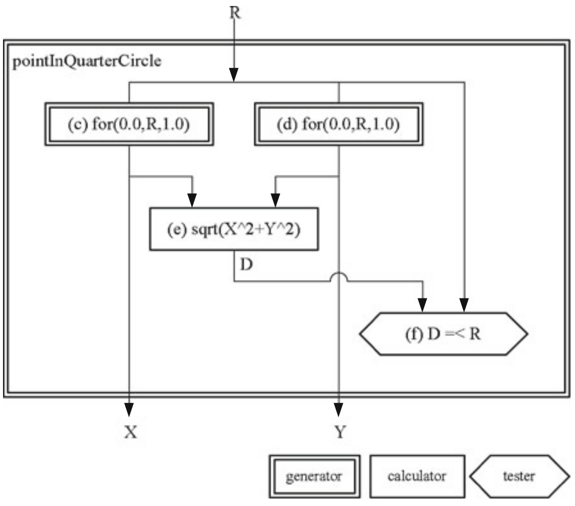


Fig. 2. Data flow diagram of module `pointInQuarterCircle`

```
for({B : real, E : real, S : real},{N : real})
  method                        --The fist method
    when(B =< E);               --(a)
    N : real = B;               --(b)
  end method;
  method                        --The second method
    when(B+S =< E);             --(c)
    B1 : real = B+S;            --(d)
    call(for, {B1, E, S}, {N}); --(e)
  end method;
end;
```

Fig. 3. Module `for`, which implements the generator primitive `for`

thus a module can also be a generator. In this example, there is overlap between the domains specified by statements `(a)` and `(c)`.

2.3 Execution Model for DSP

Since variables follow the single assignment rule and the assignment is unidirectional, the statements are partially ordered according to the dependencies between variables. During execution, the statements must be totally ordered and evaluated in this order. Although the method used to totally order the partially ordered statements does not affect the set of solutions, the order of the generators does affect the order of the solutions returned from a nondeterministic module.

The execution model for DSP can be represented in Prolog. Figure 4 gives an example of a simplified DSP interpreter in Prolog. In this interpreter, statements are represented as terms concatenated by “;” and it is assumed that the

```
solve((A ; B)) :-
     solve(A),
     solve(B).
solve(call(M, In, Out)) :-
     reduce(call(M, In, Out), Body),
     solve(Body).
solve(dcall(M, In, Out)) :-
     reduce(call(M, In, Out), Body),
     solve(Body),!.
solve(find(M, In, OutList)) :-
     findall(Out, solve(M, In, Out), OutList).
solve(when(Exp)) :-
     call(Exp),!.
solve(test(Exp)) :-
     call(Exp),!.
solve(V := for(B, E, S)) :- !,
     for(B, E, S, V).
solve(V := select(L)) :- !,
     member(V, L).
solve(V := Exp) :-
     V is Exp.
```

Fig. 4. Simplified DSP interpreter in Prolog

statements are totally ordered. Variables are represented using logical variables in Prolog. In an actual development environment on top of Inside Prolog, DSP modules are translated into Prolog code by the compiler, and the generated Prolog code is then translated into bytecode by the Prolog compiler.

3 Translation Techniques for Logic Programming Languages

Prolog is a logic programming language that offers both declarative features and practical applicability to various application domains. Many implementation techniques for Prolog and its family have been proposed, with abstract machines represented by the WAM (Warren's Abstract Machine) [15] proven effective as practical implementation techniques. Nevertheless, few Prolog implementations provide practical functionality applicable to both stand-alone systems and enterprise-mission-critical information systems without the use of other languages. Practically, Prolog is often combined with a conventional procedural language, such as Java, C, or C#, for use in practical applications. In such cases, language interoperability is an important issue.

Language translation is one possible solution for improving the interoperability between Prolog and other combined languages. jProlog [16] and Prolog Cafe [17] are Prolog to Java translators based on binarization [18], while P# [19] is a Prolog to C# translator based on Prolog Cafe with concurrent extensions. The binarization with continuation passing is a useful idea for handling nondeterminism simply in procedural languages. For example, the following clauses

```
p(X) :- q(X, Y), r(Y).
q(X, X).
r(X).
```

can be represented by semantically equivalent clauses that take a continuation goal `Cont` as the last parameter:

```
p(X, Cont) :- q(X, Y, r(Y, Cont)).
q(X, X, Cont) :- call(Cont).
r(X, Cont) :- call(Cont).
```

Once the clauses have been converted into this form, those clauses composing a predicate can be translated into Java classes. Figure 5 gives an example of code generated by Prolog Cafe. Predicate `p/2` after binarization is represented as a Java class called `PRED_p_1`, which is a subclass of class `Predicate`. The parameters of a predicate call are passed as arguments of the constructor of the class, while the right-hand side of the clause is expanded as method `exec`.

If a predicate consists of multiple clauses as in the following predicate `p/1`, it may have choice points.

```
p(X) :- q(X, Y), r(Y).
p(X) :- r(X).
```

```
public class PRED_p_1 extends Predicate {
    public Term arg1;

    public PRED_p_1(Term a1, Predicate cont) {
        arg1 = a1;
        this.cont = cont; /* this.cont is inherited. */
    }
    ...
    public Predicate exec(Prolog engine) {
        engine.setB0();
        Term a1, a2;
        Predicate p1;
        a1 = arg1;
        a2 = new VariableTerm(engine);
        p1 = new PRED_r_1(a2, cont);
        return new PRED_q_2(a1, a2, p1);
    }
}
```

Fig. 5. Java code generated by Prolog Cafe

```
public class PRED_p_1 extends Predicate {
    static Predicate _p_1_sub_1 = new PRED_p_1_sub_1();
    static Predicate _p_1_1 = new PRED_p_1_1();
    static Predicate _p_1_2 = new PRED_p_1_2();
    public Term arg1;

    ...
    public Predicate exec(Prolog engine) {
        engine.aregs[1] = arg1;
        engine.cont = cont;
        engine.setB0();
        return engine.jtry(_p_1_1, _p_1_sub_1);
    }
}

class PRED_p_1_sub_1 extends PRED_p_1 {
    public Predicate exec(Prolog engine) {
        return engine.trust(_p_1_2);
    }
}

class PRED_p_1_1 extends PRED_p_1 {
    public Predicate exec(Prolog engine) {
        Term a1, a2;
        Predicate p1;
        Predicate cont;
        a1 = engine.aregs[1];
        cont = engine.cont;
        a2 = new VariableTerm(engine);
        p1 = new PRED_r_1(a2, cont);
        return new PRED_q_2(a1, a2, p1);
    }
}

class PRED_p_1_2 extends PRED_p_1 {
    public Predicate exec(Prolog engine) {
        Term a1;
        Predicate cont;
        a1 = engine.aregs[1];
        cont = engine.cont;
        return new PRED_r_1(a1, cont);
    }
}
```

Fig. 6. Java code with choice points generated by Prolog Cafe

In this case, the generated code is more complex than before because the choice points of `p/1` must be dealt with for backtracking. Figure 6 gives an example of the generated code for predicate `p/1` above. Each clause of a predicate is mapped to a subclass of a class representing the predicate. In this example, classes `PRED_p_1_1` and `PRED_p_1_2` correspond to the two clauses of predicate `p/1`. Methods `jtry` and `trust` in the Prolog engine correspond to WAM instructions that manipulate stacks and choice points for backtracking. The key ideas in Prolog Cafe are that continuation is represented as an instance of a Java class representing a predicate, and the execution control including backtracking follows the WAM. The translation is straightforward through the WAM, and interoperability with Java-based systems is somewhat improved. On the other hand, a disadvantage is the performance of the generated code.

4 Program Representation in Java and Inference Engine

This section describes the translation techniques for the nondeterministic functional language DSP into Java based on the translation techniques for Prolog. Current implementations of the compiler and inference engine for DSP have been developed on top of Inside Prolog with the compiler generating Prolog code. Therefore, it is possible to translate this generated Prolog code into Java using Prolog Cafe. However, there are several differences between DSP and Prolog in terms of the semantics of variables and the determinism of statements. These differences allow several optimizations in performance, and the generated code can run faster than the code generated by Prolog Cafe for compatible Prolog programs. The fundamental idea behind our translation techniques is to take advantage of the single and unidirectional assignment features of variables and the deterministic features of certain statements.

The overall structure of the Java code translated from DSP provides for one module to be mapped to a single Java class, and each method in the module to be mapped to a single inner class of this class. Figure 7 shows an example of Java code for module `pointInQuarterCircle` given in Fig. 1. Inner classes are used to represent an execution context of a predicate as an internal state of a class instance. Therefore, the instances of an inner class are not declared as static unlike classes in Fig. 6.

An overview of the translation process follows. First, the data flow of a module is analyzed for each method based on the dependencies between variables, and the statements are reordered according to the analysis results. Next, the statements are grouped into translation units called continuation units, and Java code is generated for each method according to the continuation units.

4.1 Data Flow Analysis

As described in Sect. 2, it is necessary to reorder and evaluate statements so as to fulfill variable dependencies since statements can be listed in any order. Therefore, partially ordered statements must first be totally ordered. In the ordering process, the order of the generators should be kept as long as the variable dependencies are satisfied, because the order of generators affects the order of the solutions as described in Sect. 2. On the other hand, calculators or testers can be moved forward for the least commitment as long as partial orders are kept.

```
public class PointInQuarterCircle implements Executable {
  private Double r;
  private Variable x;
  private Variable y;
  private Executable cont;
  public PointInQuarterCircle(Double r,
                              Variable x, Variable y, Executable cont)
  {
    this.r = r;
    this.x = x;
    this.y = y;
    this.cont = cont;
  }

  public Executable exec(VM vm) {
    return (new Method_1()).exec(vm);
  }

  public class Method_1 implements Executable {
    private Variable d = new Variable();
    private Executable method_1_cu1 = new Method_1_cu1();
    private Executable method_1_cu2 = new Method_1_cu2();
    private Executable method_1_cu3 = new Method_1_cu3();

    public Executable exec(VM vm) {
      return method_1_cu1.exec(vm);
    }

    class Method_1_cu1 implements Executable {
      public Executable exec(VM vm) {
        return new ForDouble(0.0, r.doubleValue(), 1.0, x, method_1_cu2);
      }
    }

    class Method_1_cu2 implements Executable {
      public Executable exec(VM vm) {
        return new ForDouble(0.0, r.doubleValue(), 1.0, y, method_1_cu3);
      }
    }

    class Method_1_cu3 implements Executable {
      public Executable exec(VM vm) {
        d.setValue(Math.sqrt(x.doubleValue()*x.doubleValue() +
                             y.doubleValue()*y.doubleValue()));
        if(!(d.doubleValue() <= r.doubleValue())){
          return Executable.failure;
        }
        return cont;
      }
    }
  }
}
```

Fig. 7. Java code generated for module `pointInQuarterCircle`

4.2 Continuation Unit

If statements of a method are totally ordered, they can be divided into several groups of statements. Each group is called a continuation unit and consists of a series of deterministic statements, such as calculators and testers, followed by a single generator. It should be noted that a continuation unit may not contain a generator if it is the last one in a method. In the translation, a continuation unit is treated as a unit to translate, and is mapped to a Java class representing a continuation.

In the example in Fig. 7, module `pointInQuarterCircle` has one method, with three continuation units in the method. Inner class `Method_1` corresponds to this method of the module, while classes `Method_1_cu1`, `Method_1_cu2`, and `Method_1_cu3` correspond to the continuation units for statements (c), (d), and (e) and (f), respectively.

4.3 Variable and Parameter Passing

Although variables follow the single assignment rule as in Prolog, the binding of a variable is unidirectional, which is not the case in Prolog. Therefore, it is not necessary to introduce logical variables and unification, unlike in Prolog Cafe. This also means that the trail stack and variable unbinding using the stack are unnecessary on backtracking. Therefore, a class representing the variables is only necessary as a place holder for the output values of a module. Class `Variable` is introduced to represent such variables.

Prolog Cafe uses the registers of the Prolog VM (virtual machine) to manage the arguments of a goal. This approach is consistent with the WAM, but is sometimes inefficient since it requires arguments to be copied from/to registers to/from the stack on calls and backtracking. On the other hand, because the direction of variable binding is clearly defined in DSP, it is unnecessary to restore variable bindings on backtracking as described previously. Instead, variables can always be overwritten when a goal is re-executed after backtracking. Therefore, input and output parameters can be passed as arguments of the class constructor. This simplifies the management of variables and arguments. In addition, as shown in Fig. 7, basic Java types, such as `int` and `double`, can be passed directly as inputs in some cases. This contributes to the performance improvement.

4.4 Inference Engine

An inference engine for the translated code is very simple because management of variables and registers on backtracking is unnecessary. Figure 8 gives an example of the inference engine called `VM`, which uses a stack represented as an array of interface `Executable` to store choice points. Method `call()` is an entry point to call the module to find an initial solution, while method `redo()` is used to find the next solution. A typical call procedure of a client program in Java is given below.

```
public class VM {
  private Executable[] choicepoint;
  private int ccp = -1; // Current choice point.
  ...

  public VM(int initSize) {
    choicepoint = new Executable[initSize];
  }
  ...
  public boolean call(Executable goal) {
    while (goal != null) {
      goal = goal.exec(this);
      if (goal == Executable.success) {
        return true;
      } else if (goal == Executable.failure) {
        goal = getChoicePoint();
      }
    }
    return false;
  }

  public boolean redo() {
    return call(getChoicePoint());
  }
}
```

Fig. 8. Inference engine for DSP

```
VM vm = new VM();
Double r = new Double(10.0);
Variable x = new Variable();
Variable y = new Variable();
Executable m = new PointInQuarterCircle(r, x, y,
                                        Executable.success);
for (boolean s = vm.call(m); s == true; s = vm.redo()) {
  System.out.println("X=" + x.doubleValue() +
                     ", Y=" + y.doubleValue());
}
```

This client program creates an inference engine, prepares output variables to receive the values of a solution, creates an instance of class `PointInQuarterCircle` with inputs and outputs, and invokes `call()` to find an initial solution. It then invokes `redo()` to find the next one until there are no more solutions.

Because the implementation of the inference engine is simple and multi-thread safe, and the generated classes of a module are also multi-thread safe, it is easy to deploy several instances of the engine in a multi-thread environment.

5 Implementation and Performance Evaluation

We have implemented a translator from DSP to Java based on the techniques proposed in Sect. 4. The translator is written in DSP itself and generates Java code.

The generated code was evaluated under Windows Vista on an Intel Core2Duo 2.53 GHz processor with 3.0 GB memory. Java 1.6, Prolog Cafe 1.2.5, and B-Prolog 7.4 [20] were used for comparison in the experiments. In the case of Dsp on top of Inside Prolog, the programs written in DSP were first compiled into Prolog and then into bytecode. In the case of B-Prolog, standard Prolog features are only used though it is a CLP system.

Program `plan` is a simple architectural design program for a parking structure. It can enumerate all possible column layouts for the given design conditions, such as free land space and the number of stories. Programs `nqueens`, `ack`, and `tarai` are well-known benchmarks, with `ack` and `tarai` using green cuts for guards in Prolog, while `ack w/o cuts` and `tarai w/o cuts` do not use cuts for guards. In the case of DSP, `ack` and `tarai` use `dcall` for self-recursive calls so as not to leave choice points, while `ack w/o cuts` and `tarai w/o cuts` use `call`. The programs were forced to backtrack in each iteration to enumerate all solutions, with the execution times in milliseconds expressed as the averages over 10 trials.

5.1 Execution Times of Benchmarks

Table 2 gives the performance results of the six benchmark programs. Because the Java garbage collector affects the performance, 512 MB memory was statically allocated for the heap in all but one case.[4]

These results show that the proposed translator generates code that is more than 25 times faster than Prolog Cafe, more than 2 times faster than B-Prolog, and more than 5 times faster than DSP on top of Inside Prolog for `plan` and `nqueens`. On the other hand, for `ack` and `tarai` the translator generates code that is about 2 to 3 times faster than Prolog Cafe, but about 5 to 15 times slower than B-Prolog. The translator also generates code that is about 8 to 13 times faster than Prolog Cafe, but about 4 to 10 times slower than B-Prolog for `ack w/o cuts` and `tarai w/o cuts`. Here, `plan` and `nqueens` are nondeterministic, while `ack` and `tarai` are

Table 2. Execution times of benchmarks (in ms)

Program	DSP on Prolog	B-Prolog	Prolog Cafe	Translator
plan	685.0	295.1	2519.4	90.5
nqueens	594.9	296.2	3279.2	120.3
ack	1568.2	52.9	990.7	265.0
tarai	1302.7	49.4	1680.1	740.8
ack w/o cuts	2035.1	104.7	3421.3	403.9
tarai w/o cuts	1307.8	49.2	6282.2	489.5

[4] About 1000 MB memory was allocated for the generated code for `tarai w/o cuts`.

deterministic. `ack w/o cuts` and `tarai w/o cuts` are also deterministic, but they involve backtracking owing to the lack of green cuts.

These experiments indicate that the proposed translation techniques can generate faster code than both Prolog Cafe and DSP on top of Inside Prolog for all six programs, and faster code than B-Prolog for nondeterministic programs. In the case of deterministic programs, the advantage of the proposed translation techniques is obvious over Prolog Cafe if green cuts are not used in Prolog. The reason that these distinctive differences are observed seems to be that the simplification of the variable and register management for backtracking contributes to an improvement in performance for nondeterministic programs, but it is not effective for deterministic programs with green cuts.

In the case of B-Prolog, the execution time of `tarai` is almost the same as that of `tarai w/o cuts`. This is because the B-Prolog compiler reduces choice points using matching trees for both `tarai` and `tarai w/o cuts` [21]. Although the DSP language has no explicit cut operator as in Prolog, improving the performance by inserting cut instructions automatically in the case of exclusive `when` conditions is a future issue.

5.2 Impact of Java Heap Memory Size

The number of instances created during an execution has a negative impact on performance because of garbage collection. Figure 9 shows the performance results for `plan`, `nqueen`, and `ack` using the translator and Prolog Cafe with various Java heap memory sizes, while Fig. 10 shows the same for `ack w/o cuts`, `tarai`, and `tarai w/o cuts`. Non-plotted areas in Fig. 10 denote that the programs could not be executed as a result of running out of memory or excessive garbage collection. These results

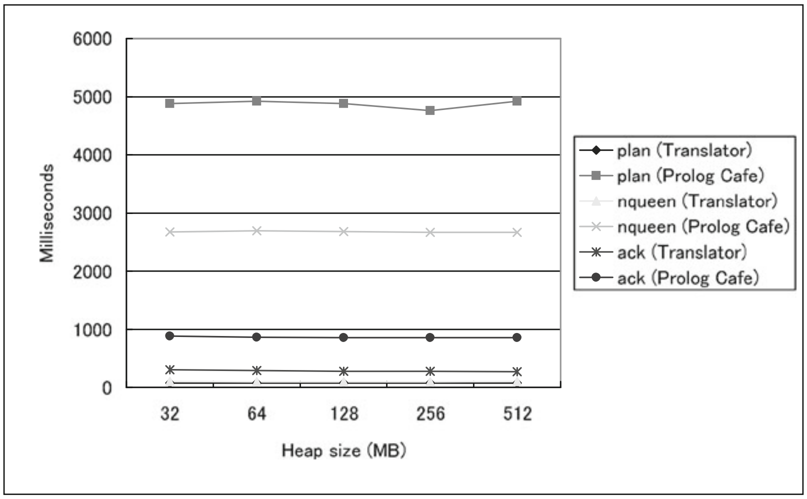


Fig. 9. Impact of Java heap memory size for `plan`, `nqueen`, and `ack`

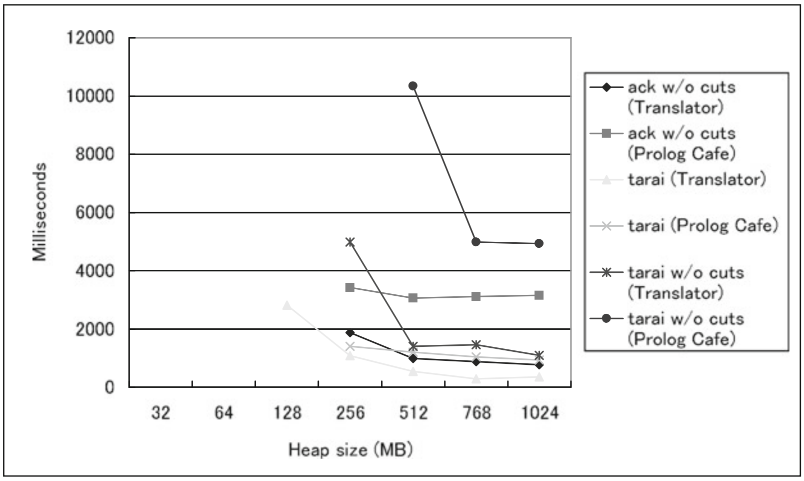


Fig. 10. Impact of Java heap memory size for `ack w/o cuts`, `tarai`, and `tarai w/o cuts`

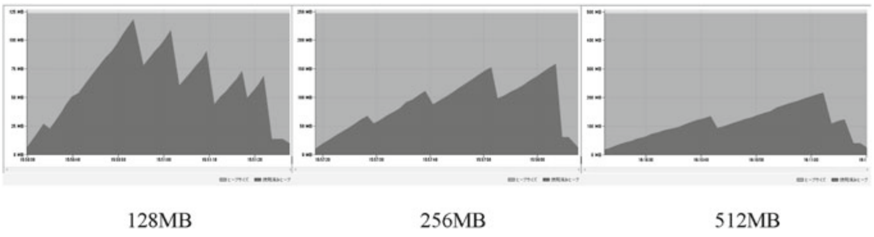


Fig. 11. Consumption trends of Java heap memory for `tarai` using the translator

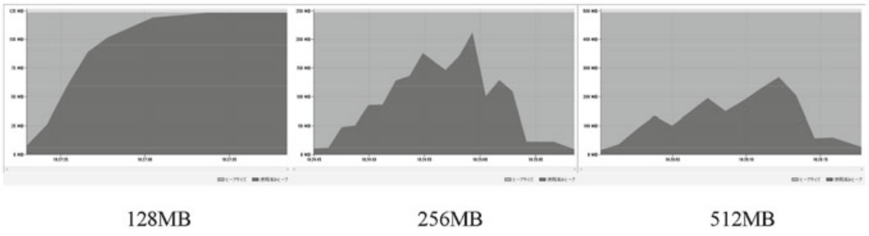


Fig. 12. Consumption trends of Java heap memory for `tarai` using Prolog Cafe

indicate that for both the translator and Prolog Cafe, the performance of `plan`, `nqueen`, and `ack` is not affected by the memory size of the Java heap, whereas the performance of `ack w/o cuts`, `tarai`, and `tarai w/o cuts` is strongly affected by the size.

Figures 11 and 12 show trends in the consumption of memory in the Java heap in the case of 128, 256, and 512 MB memories for `tarai` using the translator and Prolog Cafe, respectively. These results not only confirm that the proposed translation techniques can generate better code than Prolog Cafe for `tarai` in terms of heap consumption, but they also confirm that heap consumption is an important factor in performance improvement.

In the example in Fig. 7, it is clear that the number of instances can be reduced by merging classes `Method_1_cu1` and `Method_1`. Improving the performance by reducing the number of instances created is an important future issue.

6 Conclusions

This paper described the techniques for translating DSP, a nondeterministic functional language based on attribute grammars, into Java. DSP is designed for knowledge representation of large-scale and complicated expert knowledge in application domains. It is capable of representing trial and error without any side-effects or loop constructs using nondeterministic features. Current development and runtime environments are built on top of Inside Prolog, while the runtime environment can be embedded in a Java-based application server. However, issues regarding language interoperability and adaptability to new computing environments are unavoidable when applied to practical application development. Language translation is intended to improve the interoperability and adaptability of DSP.

The proposed translation techniques are based on binarization and the techniques proposed for the translation of Prolog. The performance, however, is improved by introducing a continuation unit and simplifying the management of variables and registers using the semantic differences of variables and explicit determinism of some statements. An experimental translator written in DSP itself generates Java code from DSP descriptions. Experimental results indicate that the generated code is more than 25 times faster than that of Prolog Cafe for nondeterministic programs, and more than 2 times faster for deterministic programs. The generated code is also more than 2 times faster than B-Prolog for nondeterministic programs. However, the generated code is about 3 to 15 times slower than B-Prolog for deterministic programs. Improving the performance of deterministic programs is an important future issue.

References

1. Kaplan, B.: Evaluating informatics applications - clinical decision support systems literature review. Int. J. Med. Inform. **64**, 15–37 (2001)
2. Takada, A., Nagase, K., Ohno, K., Umeda, M., Nagasawa, I.: Clinical decision support system, how do we realize it for hospital information system? Jpn. J. Med. Inform. **27**, 315–320 (2007)
3. Nagasawa, H., Nagasawa, I., Takata, O., Umeda, M., Hashimoto, M., Takizawa, C.: Knowledge modeling for operation management support of a distribution center. In: The Sixth IEEE International Conference on Computer and Information Technology (CIT2006) (2006)
4. Nagasawa, I., Maeda, J., Tegoshi, Y., Makino, M.: A programming technique for some combination problems in a design support system using the method of generate-and-test. J. Struct. Constr. Eng. **417**, 157–166 (1990)

5. Umeda, M., Nagasawa, I., Higuchi, T.: The elements of programming style in design calculations. In: Proceedings of the Ninth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, pp. 77–86 (1996)
6. Katayama, T.: A computation model based on attribute grammar. J. Inf. Process. Soc. Jpn. **24**, 147–155 (1983)
7. Deransart, P., Jourdan, M. (eds.): Attribute Grammars and their Applications. LNCS, vol. 461. Springer, Heidelberg (1990)
8. Katamine, K., Umeda, M., Nagasawa, I., Hashimoto, M.: Integrated development environment for knowledge-based systems and its practical application. IEICE Trans. Inf. Syst. **E87–D**, 877–885 (2004)
9. ISO/IEC: 13211–1 Information technology - Programming Languages - Prolog - Part 1: General core (1995)
10. Umeda, M., Katamine, K., Nagasawa, I., Hashimoto, M., Takata, O., et al.: Multi-threading inside prolog for knowledge-based enterprise applications. In: Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O. (eds.) INAP 2005. LNCS (LNAI), vol. 4369, pp. 200–214. Springer, Heidelberg (2006)
11. Umeda, M., Katamine, K., Nagasawa, I., Hashimoto, M., Takata, O.: The design and implementation of knowledge processing server for enterprise information systems. Trans. Inf. Process. Soc. Jpn. **48**, 1965–1979 (2007)
12. Nagasawa, I.: Feature of design and intelligent CAD. J. Jpn. Soc. Precis. Eng. **54**, 1429–1434 (1988)
13. Umeda, M., Mure, Y.: Knowledge management strategy and tactics for forging die design support. In: Seipel, D. (ed.) INAP 2009. LNCS, vol. 6547, pp. 188–204. Springer, Heidelberg (2011)
14. Hirota, T., Hashimoto, M., Nagasawa, I.: A discussion on conceptual model description language specific for an application domain. Trans. Inf. Process. Soc. Jpn. **36**, 1151–1162 (1995)
15. Ait-Kaci, H.: Warren's Abstract Machine. MIT Press, Cambridge (1991)
16. Demoen, B., Tarau, P.: jprolog home page. http://www.cs.kuleuven.ac.be/bmd/PrologInJava/ (1996)
17. Banbara, M., Tamura, N., Inoue, K.: Prolog Cafe: a prolog to Java translator system. In: Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O. (eds.) INAP 2005. LNCS (LNAI), vol. 4369, pp. 1–11. Springer, Heidelberg (2006)
18. Tarau, P., Boyer, M.: Elementary logic programs. In: Deransart, P., Maluszyński, J. (eds.) PLILP 1990. LNCS, vol. 456, pp. 159–173. Springer, Heidelberg (1990)
19. Cook, J.J.: Language interoperability and logic programming languages. In: Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, Doctor of philosophy (2004)
20. Neng-Fa, Z.: The language features and architecture of b-prolog. Theor. Pract. Logic Program. **11**, 537–553 (2011)
21. Neng-Fa, Z.: Global optimizations in a prolog compiler for the TOAM. J. Logic Program. **15**, 265–294 (1993)

Sensitivity Analysis for Declarative Relational Query Languages with Ordinal Ranks

Radim Belohlavek, Lucie Urbanova, and Vilem Vychodil(✉)

DAMOL (Data Analysis and Modeling Laboratory),
Department of Computer Science, Palacky University,
Olomouc 17. listopadu 12, CZ–77146 Olomouc, Czech Republic
radim.belohlavek@acm.org, lurbanova@acm.org, vychodil@acm.org

Abstract. We present sensitivity analysis for results of query executions in a relational model of data extended by ordinal ranks. The underlying model of data results from the ordinary Codd's model of data in which we consider ordinal ranks of tuples in data tables expressing degrees to which tuples match queries. In this setting, we show that ranks assigned to tuples are insensitive to small changes, i.e., small changes in the input data do not yield large changes in the results of queries.

Keywords: Declarative query languages · Ordinal ranks · Relational databases · Residuated lattices

1 Introduction

Since its inception, the relational model of data introduced by E. Codd [9] has been extensively studied by both computer scientists and database systems developers. The model has become the standard theoretical model of relational data and the formal foundation for relational database management systems. Various reasons for the success and strong position of Codd's model are analyzed in [13], where the author emphasizes that the main virtues of the model like logical and physical data independence, declarative style of data retrieval (database querying), access flexibility and data integrity are consequences of a close connection between the model and the first-order predicate logic.

This paper is a continuation of our previous work [3,4] where we have introduced an extension of Codd's model in which tuples are assigned ordinal ranks. The motivation for the model is that in many situations, it is natural to consider not only the exact matches of queries in which a tuple of values either *does* or *does not* match a query Q but also approximate matches where tuples match queries to degrees. The degrees of approximate matches can usually be described verbally using linguistic modifiers like "not at all (matches)" "almost (matches)", "more or less (matches)", "fully (matches)", etc. From the user's

Supported by grant no. P103/11/1456 of the Czech Science Foundation and internal grant of Palacky University no. PrF_2012_029. DAMOL is supported by project reg. no. CZ.1.07/2.3.00/20.0059 of the European Social Fund in the Czech Republic.

point of view, each data table in our extended relational model consists of (i) an ordinary data table whose meaning is the same as in the Codd's model and (ii) ranks assigned to all tuples in the original data table. This way, we introduce a notion of a ranked data table (shortly, an RDT). The ranks in RDTs are interpreted as "goodness of match" and the interpretation of RDTs is the same as in the Codd's model—they represent answers to queries which are, in addition, equipped with priorities expressed by the ranks. A user who looks at an answer to a query in our model is typically looking for the best match possible represented by a tuple or tuples in the resulting RDT with the highest ranks (i.e., highest priorities).

In order to have a suitable formalization of ranks and to perform operations with ranked data tables, we have to choose a suitable structure for ranks. Since ranks are meant to be compared by users, the set L of all considered ranks should be equipped with a partial order $\leq$, i.e. $\langle L, \leq \rangle$ should be a poset. Moreover, it is convenient to postulate that $\langle L, \leq \rangle$ is a complete lattice [6], i.e., for each subset $A \subseteq L$, its least upper bound (a supremum) and greatest lower bound (an infimum) exist. This way, for any $A \subseteq L$, one can take the least rank in L which represents a higher priority (a better match) than all ranks from A. Such a rank is then the supremum of A (dually for the infimum). Since $\langle L, \leq \rangle$ is a complete lattice, it contains the least element denoted 0 (no match at all) and the greatest element denoted 1 (full match).

The set L of all ranks should also be equipped with additional operations for aggregation of ranks. Indeed, if tuple t with rank a is obtained as one of the results of subquery Q_1 and the same t with another rank b is obtained from answers to subquery Q_2 then we might want to express the rank to which t matches a compound conjunctive query "Q_1 *and* Q_2". A natural way to do so is to take a suitable binary operation $\otimes \colon L \times L \to L$ which acts as a conjunctor and take $a \otimes b$ for the resulting rank. Obviously, not every binary operation on L represents a (reasonable) conjunctor, i.e. we may restrict the choices only to particular binary operations that make "good conjunctors". There are various ways to impose such restrictions. In our model, we follow the approach of using residuated conjunctions that has proved to be useful in logics based on residuated lattices [2,18,19]. Namely, we assume that $\langle L, \otimes, 1 \rangle$ is a commutative monoid (i.e., $\otimes$ is associative, commutative, and neutral with respect to 1) and there is a binary operation $\to$ on L such that for all $a, b, c \in L$:

$$a \otimes b \leq c \quad \text{if and only if} \quad a \leq b \to c. \tag{1}$$

Operations $\otimes$ (a multiplication) and $\to$ (a residuum) satisfying (1) are called *adjoint operations.* Altogether, the structure for ranks we use in this paper is a *complete residuated lattice* $\mathbf{L} = \langle L, \wedge, \vee, \otimes, \to, 0, 1 \rangle$, i.e., a complete lattice in which $\otimes$ and $\to$ are adjoint operations, and $\wedge$ and $\vee$ denote the operations of infimum and supremum, respectively. Considering $\mathbf{L}$ as a basic structure of ranks brings several benefits including a close connection to formal logics based on residuated structures. This is due to the fact that in multiple-valued logics and in particular fuzzy logics [18,19], residuated lattices are interpreted as structures of truth degrees and the relationship (1) between $\otimes$ (a fuzzy conjunction) and $\rightarrow$ (a fuzzy implication) is derived from requirements on graded counterpart of the *modus ponens* deduction rule (currently, there are many strong-complete logics based on residuated lattices, see [18] for an overview).

Remark 1. The graded counterpart of *modus ponens* [19,27] can be seen as a generalized deduction rule saying "from φ valid (at least) to degree $a \in L$ and $\varphi \Rightarrow \psi$ valid (at least) to degree $b \in L$, infer ψ valid (at least) to degree $a \otimes b$". The if-part of (1) ensures that the rule is sound while the only-if part ensures that it is as powerful as possible, i.e., $a \otimes b$ is the highest degree to which we infer ψ valid provided that φ is valid at least to degree a and $\varphi \Rightarrow \psi$ is valid at least to degree $b \in L$. This relationship between $\rightarrow$ (a truth function for logical connective implication $\Rightarrow$) and $\otimes$ (a truth function for conjunction) has been discovered in [17] and later used, e.g., in [16,27]. Interestingly, (1) together with the lattice ordering ensure enough properties of $\rightarrow$ and $\otimes$. For instance, $\rightarrow$ is antitone in the first argument and is monotone in the second one, condition $a \leq b$ iff $a \rightarrow b = 1$ holds for all $a, b \in L$, $a \rightarrow (b \rightarrow c)$ equals $(a \otimes b) \rightarrow c$ for all $a, b, c \in L$, etc. Since complete residuated lattices are in general weaker structures than Boolean algebras, not all laws satisfied by truth functions of the classic conjunction and implication are preserved by all complete residuated lattices. For instance, neither $a \otimes a = a$ (idempotency of $\otimes$) nor $(a \rightarrow 0) \rightarrow 0 = a$ (the law of double negation) nor $a \vee (a \rightarrow 0) = 1$ (the law of the excluded middle) hold in general. Nevertheless, complete residuated lattices are strong enough to provide a formal framework for relational analysis and similarity-based reasoning as it has been shown by previous results [2].

Our extension of the Codd's model results from the model by replacing the two-element Boolean algebra, which is the classic structure of truth values, by a more general structure of truth values represented by a residuated lattice, i.e. we make the following shift in (the semantics of) the underlying logic:

$$\textit{two-element Boolean algebra} \quad \Longmapsto \quad \textit{a complete residuated lattice.}$$

As a consequence, the original Codd's model becomes a special case of our model for $\mathbf{L}$ being the two-element Boolean algebra (only two borderline ranks 1 and 0 are available). As a practical consequence, data tables in the Codd's model can be seen as RDTs where all ranks are either equal to 1 (full match) or 0 (no match; tuples with 0 rank are considered as not present in the result of a query). Using residuated lattices as structures of truth degrees, we obtain a generalization of Codd's model which is based on solid logical foundations and has desirable properties. In addition, its relationship to residuated first-order logics is the same as the relationship of the original Codd's model to the classic first-order logic. The formalization we offer can further be used to provide insight into several isolated approaches that have been provided in the past, see e.g. [7,15,24,28,29,31], and a comparison paper [5].

A typical choice of $\mathbf{L}$ is a structure with $L = [0,1]$ (ranks are taken from the real unit interval), $\wedge$ and $\vee$ being minimum and maximum, $\otimes$ being a left-continuous (or a continuous) t-norm with the corresponding $\rightarrow$, see [2,18,19].

Table 1. Houses for sale at `$200,000` with square footage `1200`

	agent	*id*	*sqft*	*age*	*location*	*price*
0.93	`Brown`	138	1185	48	`Vestal`	$228,500
0.89	`Clark`	140	1120	30	`Endicott`	$235,800
0.86	`Brown`	142	950	50	`Binghamton`	$189,000
0.85	`Brown`	156	1300	85	`Binghamton`	$248,600
0.81	`Clark`	158	1200	25	`Vestal`	$293,500
0.81	`Davis`	189	1250	25	`Binghamton`	$287,300
0.75	`Davis`	166	1040	50	`Vestal`	$286,200
0.37	`Davis`	112	1890	30	`Endicott`	$345,000

For example, an RDT with ranks coming from such **L** is in Table 1. It can be seen as a result of similarity-based query "show all houses which are sold for (approximately) `$200,000` and have (approximately) `1200` square feet". The left-most column contains ranks. The remaining part of the table is a data table in the usual sense containing tuples of values (a relation on a relation scheme in the standard database terminology). At this point, we do not explain in detail how the particular ranks in Table 1 have been obtained (this will be outlined in further sections). One way is by executing a similarity-based query that uses additional information about similarity (proximity) of domain values which is also described using degrees from **L**. Note that the concept of a similarity-based query appears when human perception is involved in rating or comparing close values from domains where not only the exact equalities (matches) are interesting. For instance, a person searching in a database of houses is usually not interested in houses sold for a particular exact price. Instead, the person wishes to look at houses sold approximately at that price, including those which are sold for other prices that are sufficiently close. While the ranks constitute a "visible" part of any RDT, the similarities are not a direct part of RDT and have to be specified for each domain independently. They can be seen as an additional (background) information about domains which is supplied by users of the database system.

Let us stress the meaning of ranks as priorities. As it is usual in fuzzy logics in narrow sense, their meaning is primarily *comparative*, cf. [19, p. 2] and the comments on comparative meaning of truth degrees therein. In our example, it means that tuple $\langle \texttt{Clark}, 140, 1120, 30, \texttt{Endicott}, \texttt{\$235,800} \rangle$ with rank 0.89 is a better match than tuple $\langle \texttt{Brown}, 142, 950, 50, \texttt{Binghamton}, \texttt{\$189,000} \rangle$ whose rank 0.86 is strictly smaller. Thus, for end-users, the numerical values of ranks (if L is a unit interval) are not so important, the important thing is the relative ordering of tuples given by the ranks.

Note that our model which provides theoretical foundations for similarity-based databases [3,4] should not be confused with models for probabilistic databases [30] which have recently been studied, e.g. in [8,11,12,22,23,26], see also [10] for a survey. In particular, numerical ranks used in our model (if $L = [0, 1]$) cannot be interpreted as probabilities, confidence degrees of belief degrees as in case of probabilistic databases where ranks play such roles. In probabilistic databases, the tuples (i.e., the data itself) are uncertain and the ranks express probabilities that tuples appear in data tables. Consequently, a probabilistic database is formalized by a discrete probability space over the possible contents of the database [10]. Nevertheless, the underlying logic of the models is the classical two-valued first-order logic—only yes/no matches are allowed (with uncertain outcome). In our case, the situation is quite different. The data (represented by tuples) is absolutely certain but the tuples are allowed to match queries to degrees. This, translated in terms of logic, means that formulas (encoding queries) are allowed to be evaluated to truth degrees other than 0 and 1. Therefore, the underlying logic in our model is not the classic two-element Boolean logic as we have argued hereinbefore.

In [1], a report written by leading authorities in database systems, the authors say that the current database management systems have no facilities for either approximate data or imprecise queries. According to this report, the management of uncertainty and imprecision is one of the six currently most important research directions in database systems. Nowadays, probabilistic databases (dealing with approximate data) are extensively studied. On the contrary, it seems that similarity-based databases (dealing with imprecise queries) have not yet been paid full attention. This paper is a contribution to theoretical foundations of similarity-based databases.

2 Problem Setting

The issue we address in this paper is the following. In our model, we can get two or more RDTs (as results of queries) which are not exactly the same but which are perceived (by users) as being similar. For instance, one can obtain two RDTs containing the same tuples with numerical values of ranks that are almost the same. A question is whether such similar RDTs, when used in subsequent queries, yield similar results. In this paper, we present a preliminary study of the phenomenon of similarity of RDTs and its relationship to the similarity of query results obtained by applying queries to similar input data tables. We present basic notions and results providing formulas for computing estimations of similarity degrees. The observations we present provide a formal justification for the phenomenon discussed in the previous section—slight changes in ranks do not have a large impact on the results of (complex) queries. The results are obtained for any complete residuated lattice taken as the structure of ranks (truth degrees). Note that the basic query systems in our model are (extensions of) domain relational calculus [4,25] and relational algebra [3,25]. We formulate the results in terms of operations of the relational algebra but due to its equivalence with the domain relational calculus [4], the results pertain to both the query systems. Thus, based on the domain relational calculus, one may design a declarative query language preserving similarity in which execution of queries is based on transformations to expressions of relational algebra in a similar way as in the classic case [25].

The rest of the paper is organized as follows. Section 3 presents a short survey of notions. Section 4 contains results on sensitivity analysis, an illustrative example, and a short outline of future research. Because of the limited scope of the paper, proofs are sketched or omitted.

3 Preliminaries

In this section, we recall basic notions of RDTs and relational operations we need to provide insight into the sensitivity issues of RDTs in Sect. 4. Details can be found in [2,3,5].

3.1 Complete Residuated Lattices

In the rest of the paper, **L** which serves as the structure of ranks always refers to a complete residuated lattice $\mathbf{L} = \langle L, \wedge, \vee, \otimes, \rightarrow, 0, 1\rangle$, i.e. a complete lattice in which $\otimes$ and $\rightarrow$ are adjoint operations, and $\wedge$ and $\vee$ denote the operations of infimum and supremum, respectively, see Sect. 1 for comments on its intended meaning. A typical choice of a complete residuated lattice **L** is a structure with $L = [0, 1]$ (real unit interval), $\wedge$ and $\vee$ being minimum and maximum, $\otimes$ being a left-continuous t-norm with the corresponding residuum $\rightarrow$ satisfying (1) which is uniquely given by $a \rightarrow b = \bigvee\{c \in L \mid a \otimes c \leq b\}$. Three of the most important [19] adjoint pairs $\langle\otimes, \rightarrow\rangle$ where $\otimes$ is a continuous t-norm are:

$$a \otimes b = \max(a + b - 1, 0), \qquad a \rightarrow b = \min(1 - a + b, 1), \qquad \text{Łukasiewicz},$$

$$a \otimes b = \min(a, b), \qquad a \rightarrow b = \begin{cases} 1, & \text{if } a \leq b, \\ b, & \text{else,} \end{cases} \qquad \text{Gödel},$$

$$a \otimes b = a \cdot b, \qquad a \rightarrow b = \begin{cases} 1, & \text{if } a \leq b, \\ \frac{b}{a}, & \text{else,} \end{cases} \qquad \text{Goguen.}$$

For instance, $0.9 \otimes 0.7 = 0.6$ and $0.9 \rightarrow 0.7 = 0.8$ if $\langle\otimes, \rightarrow\rangle$ are the Łukasiewicz operations; $0.9 \otimes 0.7 = 0.7$ and $0.9 \rightarrow 0.7 = 0.7$ if $\langle\otimes, \rightarrow\rangle$ are the Gödel operations; $0.9 \otimes 0.7 = 0.63$ and $0.9 \rightarrow 0.7 = \frac{7}{9}$ if $\langle\otimes, \rightarrow\rangle$ are the Goguen (product) operations. Given **L**, we introduce notions described in the following subsections.

3.2 Basic Structures

An **L**-set A in universe U is a map $A\colon U \rightarrow L$, $A(u)$ being interpreted as "the degree to which u belongs to A". If $L = [0, 1]$, i.e., if the structure of ranks **L** is defined on the real unit interval, **L**-sets are traditionally called fuzzy sets [33]. If **L** is the two-element Boolean algebra, then $A\colon U \rightarrow L$ is an indicator function of a classic subset of U, $A(u) = 1$ ($A(u) = 0$) meaning that u belongs (does not belong) to that subset. In our approach, we tacitly identify sets with their indicator functions. Thus, by a slight abuse of notation, we use both $A(u) = 1$ and $u \in A$ to denote the fact that u belongs to A and analogously for $A(u) = 0$ and $u \notin A$.

A binary **L**-relation B on U is a map $B\colon U \times U \rightarrow L$, $B(u_1, u_2)$ interpreted as "the degree to which u_1 and u_2 are related according to B". Hence, B is an **L**-set in the universe $U \times U$.

3.3 Ranked Data Tables over Domains with Similarities

We make use of the following database terminology [25]. We denote by Y a set of *attributes*, any finite subset $R \subseteq Y$ is called a *relation scheme.* For each attribute $y \in Y$ we consider its *domain* D_y, i.e., a set of all values that are allowed for attribute $y \in Y$. Note that domains are sometimes called types [14]. The role of attributes (as names for columns of data tables) and domains (as sets of possible values appearing in columns inside the data tables) is in our model the same as in the Codd's model.

In addition, we equip each D_y with a binary **L**-relation $\approx_y$ on D_y satisfying reflexivity ($u \approx_y u = 1$) and symmetry $u \approx_y v = v \approx_y u$ (for all $u, v \in D_y$). Each binary **L**-relation $\approx_y$ on D_y satisfying (i) and (ii) shall be called a *similarity.* Pair $\langle D_y, \approx_y \rangle$ is called a *domain with similarity.* Note that $u \approx_y v$ is a general degree from L and it has the same basic comparative interpretation as ranks. For instance, $u_1 \approx_y v_1 > u_2 \approx_y v_2$ means that values u_1 and v_1 are more similar than values u_2 and v_2, etc.

Tuples contained in data tables are considered as usual, i.e., as elements of Cartesian products of domains. Recall that a Cartesian product $\prod_{i \in I} D_i$ of an I-indexed system $\{D_i \,|\, i \in I\}$ of sets D_i $(i \in I)$ is a set of all maps $t\colon I \to \bigcup_{i \in I} D_i$ such that $t(i) \in D_i$ holds for each $i \in I$.

Under this notation, a *tuple* on relation scheme $R \subseteq Y$ over domains D_y is any element from $\prod_{y \in R} D_y$. For brevity, $\prod_{y \in R} D_y$ is denoted by $\mathrm{Tupl}(R)$. Following the example in Table 1, tuple $\langle \texttt{Brown}, \texttt{142}, \texttt{950}, \texttt{50}, \texttt{Binghamton}, \texttt{\$189,000} \rangle$ is a map $r \in \mathrm{Tupl}(R)$ for $R = \{\textit{agent}, \textit{id}, \ldots, \textit{price}\}$ such that $r(\textit{agent}) = \texttt{Brown}$, $r(\textit{id}) = \texttt{142}$, $\ldots$, $r(\textit{price}) = \texttt{\$189,000}$.

A *ranked data table* on $R \subseteq Y$ over $\{\langle D_y, \approx_y \rangle \,|\, y \in R\}$ (shortly, an RDT) is any **L**-set $\mathcal{D}$ in $\mathrm{Tupl}(R)$. The degree $\mathcal{D}(r)$ to which r belongs to $\mathcal{D}$ is called a *rank* of tuple r in $\mathcal{D}$. According to its definition, if $\mathcal{D}$ is an RDT on R over $\{\langle D_y, \approx_y \rangle \,|\, y \in R\}$ then $\mathcal{D}$ is a map $\mathcal{D}\colon \mathrm{Tupl}(R) \to L$. Note that $\mathcal{D}$ is an n-ary **L**-relation between domains D_y $(y \in Y)$ since $\mathcal{D}$ is a map from $\prod_{y \in R} D_y$ to L. This allows us to visualize RDTs as the ordinary data tables with a new "column containing ranks" as it is shown in Table 1. Note however, that conceptually the ranks do not represent data. They should be seen as "metadata" or "annotations of rows in tables", see Sect. 1. In our example, we have e.g. $\mathcal{D}(r) = 0.86$ for r being the tuple with $r(\textit{id}) = \texttt{142}$.

Note also that for practical reasons, we should limit ourselves only to finite RDTs, i.e., to such ranked data tables $\mathcal{D}$ such that there are only finitely many tuples r such that $\mathcal{D}(r) > 0$. In that case, each RDT can be depicted as a table with finitely many rows. There are several possible ways to ensure the finiteness (e.g., considering only finite domains, using active domains [25], constraining all operations by introducing ranges, forced use of $\textit{top}_k$-like operations to include only best matches [21], . . .). In order to keep things conceptually simple, we omit the issue of finiteness in this preliminary study and postpone it to an extended version of the paper (as a result, the query system used in this paper is domain dependent [25]).

3.4 Relational Operations with RDTs

Relational operations we consider in this paper are the following: For RDTs $\mathcal{D}_1$ and $\mathcal{D}_2$ on T, we put $(\mathcal{D}_1 \cup \mathcal{D}_2)(t) = \mathcal{D}_1(t) \vee \mathcal{D}_2(t)$ and $(\mathcal{D}_1 \cap \mathcal{D}_2)(t) = \mathcal{D}_1(t) \wedge \mathcal{D}_2(t)$ for each $t \in \mathrm{Tupl}(T)$; $\mathcal{D}_1 \cup \mathcal{D}_2$ and $\mathcal{D}_1 \cap \mathcal{D}_2$ are called the *union* and the $\wedge$-*intersection* of $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. Analogously, one can define an $\otimes$-*intersection* $\mathcal{D}_1 \otimes \mathcal{D}_2$. Hence, $\cup$, $\cap$, and $\otimes$ are defined componentwise based on the operations of the complete residuated lattice $\mathbf{L}$ and can be seen as counterparts to the ordinary set-theoretic operations with relations on relation schemes.

Let us note that our model admits new operations that are trivial in the classic model. For instance, for $a \in L$, we introduce an *a-shift* $a{\to}\mathcal{D}$ of $\mathcal{D}$ by $(a{\to}\mathcal{D})(t) = a \to \mathcal{D}(t)$ for all $t \in \mathrm{Tupl}(T)$.

Remark 2. Clearly, if $\mathbf{L}$ is the two-element Boolean algebra then a-shifts are trivial operation since $1 \to \mathcal{D} = \mathcal{D}$ and $0 \to \mathcal{D}$ produces a possibly infinite table containing all tuples from $\mathrm{Tupl}(T)$. In our model, an a-shift has the following meaning: If $\mathcal{D}$ is a result of query Q then $(a{\to}\mathcal{D})(t)$ is a "degree to which t matches query Q at least to degree a". This follows from properties of residuum ($a \leq b$ iff $a \to b = 1$), see [2,19]. Hence, a-shifts allow us to emphasize results that match queries at least to a prescribed degree a.

The remaining relational operations we consider represent counterparts to projection, selection, and join in our model. Recall that for $r \in \mathrm{Tupl}(R)$ and $s \in \mathrm{Tupl}(S)$ such that $R \cap S = \emptyset$, rs denotes a concatenation $rs \in \mathrm{Tupl}(R \cup S)$ of tuples r and s so that $(rs)(y) = r(y)$ for $y \in R$ and $(rs)(y) = s(y)$ for $y \in S$. If $\mathcal{D}$ is an RDT on T, the *projection* $\pi_R(\mathcal{D})$ *of* $\mathcal{D}$ onto $R \subseteq T$ is defined by

$$(\pi_R(\mathcal{D}))(r) = \textstyle\bigvee_{s \in \mathrm{Tupl}(T \setminus R)} \mathcal{D}(rs),$$

for each $r \in \mathrm{Tupl}(R)$. Note that the concatenation rs used in the latter formula is correct since both r and s are tuples on disjoint relation schemes. In our example, the result of $\pi_{\{location\}}(\mathcal{D})$ is a ranked data table with single column such that $\pi_{\{location\}}(\mathcal{D})(\langle\texttt{Binghamton}\rangle) = 0.86$, $\pi_{\{location\}}(\mathcal{D})(\langle\texttt{Vestal}\rangle) = 0.93$, and $\pi_{\{location\}}(\mathcal{D})(\langle\texttt{Endicott}\rangle) = 0.89$.

A similarity-based selection is a counterpart to ordinary selection which selects from a data table all tuples which approximately match a given condition: Let $\mathcal{D}$ be an RDT on T and let $y \in T$ and $d \in D_y$. Then, a *similarity-based selection* $\sigma_{y \approx d}(\mathcal{D})$ of tuples in $\mathcal{D}$ matching $y \approx d$ is defined by

$$\big(\sigma_{y \approx d}(\mathcal{D})\big)(t) = \mathcal{D}(t) \otimes t(y) \approx_y d.$$

Considering $\mathcal{D}$ as a result of query Q, the rank of t in $\sigma_{y \approx d}(\mathcal{D})$ can be interpreted as a degree to which "t matches the query Q and the y-value of t is similar to d". In particular, an interesting case is $\sigma_{p \approx q}(\mathcal{D})$ where p and q are both attributes with a common domain with similarity.

Similarity-based joins are considered as derived operations based on Cartesian products and similarity-based selections.

For RDTs $\mathcal{D}_1$ and $\mathcal{D}_2$ on disjoint relation schemes S and T we define a RDT $\mathcal{D}_1 \times \mathcal{D}_2$ on $S \cup T$, called a *Cartesian product* of $\mathcal{D}_1$ and $\mathcal{D}_2$, by $(\mathcal{D}_1 \times \mathcal{D}_2)(st) = \mathcal{D}_1(s) \otimes \mathcal{D}_2(t)$. Using Cartesian products and similarity-based selections, we can introduce *similarity-based θ-joins* such as $\mathcal{D}_1 \bowtie_{p\approx q} \mathcal{D}_2 = \sigma_{p\approx q}(\mathcal{D}_1 \times \mathcal{D}_2)$. Various other types of similarity-based joins can be introduced in our model, see [4].

4 Estimations of Sensitivity of Query Results

4.1 Rank-Based Similarity of Query Results

We now introduce the notion of similarity of RDTs which is based on the idea that RDTs $\mathcal{D}_1$ and $\mathcal{D}_2$ (on the same relation scheme) are similar iff for each tuple t, ranks $\mathcal{D}_1(t)$ and $\mathcal{D}_2(t)$ are similar (degrees from $\mathbf{L}$). Similarity of ranks can be expressed by biresiduum $\leftrightarrow$ (a fuzzy equivalence [2,18,19]) which is a derived operation of $\mathbf{L}$ such that $a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$. Since we are interested in similarity of $\mathcal{D}_1(t)$ and $\mathcal{D}_2(t)$ for all possible tuples t, it is straightforward to define the similarity $E(\mathcal{D}_1, \mathcal{D}_2)$ of $\mathcal{D}_1$ and $\mathcal{D}_2$ by an infimum which goes over all tuples:

$$E(\mathcal{D}_1, \mathcal{D}_2) = \textstyle\bigwedge_{t \in \mathrm{Tupl}(T)} \bigl(\mathcal{D}_1(t) \leftrightarrow \mathcal{D}_2(t)\bigr). \tag{2}$$

An alternative (but equivalent) way is the following: we first formalize a degree $S(\mathcal{D}_1, \mathcal{D}_2)$ to which $\mathcal{D}_1$ is included in $\mathcal{D}_2$. We can say that $\mathcal{D}_1$ is fully included in $\mathcal{D}_2$ iff, for each tuple t, the rank $\mathcal{D}_2(t)$ is at least as high as the rank $\mathcal{D}_1(t)$. Notice that in the classic (two-values) case, this is exactly how one defines the ordinary subsethood relation "$\subseteq$". Considering general degrees of inclusion (subsethood), a degree $S(\mathcal{D}_1, \mathcal{D}_2)$ to which $\mathcal{D}_1$ is included in $\mathcal{D}_2$ can be defined as follows:

$$S(\mathcal{D}_1, \mathcal{D}_2) = \textstyle\bigwedge_{t \in \mathrm{Tupl}(T)} \bigl(\mathcal{D}_1(t) \rightarrow \mathcal{D}_2(t)\bigr). \tag{3}$$

It is easy to prove [2] that (2) and (3) satisfy:

$$E(\mathcal{D}_1, \mathcal{D}_2) = S(\mathcal{D}_1, \mathcal{D}_2) \wedge S(\mathcal{D}_2, \mathcal{D}_1). \tag{4}$$

Note that E and S defined by (2) and (3) are known as degrees of similarity and subsethood from general fuzzy relational systems [2] (in this case, the fuzzy relations are RDTs).

The following assertion shows that $\cup$, $\cap$, $\otimes$, and a-shifts preserve subsethood degrees given by (3). In words, the degree to which $\mathcal{D}_1 \cup \mathcal{D}_2$ is included in $\mathcal{D}'_1 \cup \mathcal{D}'_2$ is at least as high as the degree to which $\mathcal{D}_1$ is included in $\mathcal{D}'_1$ and $\mathcal{D}_2$ is included in $\mathcal{D}'_2$. A similar verbal description can be made for the other operations.

Theorem 1. *For any $\mathcal{D}_1$, $\mathcal{D}'_1$, $\mathcal{D}_2$, and $\mathcal{D}'_2$ on relation scheme T,*

$$S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{D}'_1 \cup \mathcal{D}'_2), \tag{5}$$

$$S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \cap \mathcal{D}_2, \mathcal{D}'_1 \cap \mathcal{D}'_2), \tag{6}$$

$$S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \otimes \mathcal{D}_2, \mathcal{D}'_1 \otimes \mathcal{D}'_2), \tag{7}$$

$$S(\mathcal{D}'_1, \mathcal{D}_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \rightarrow \mathcal{D}_2, \mathcal{D}'_1 \rightarrow \mathcal{D}'_2). \tag{8}$$

Proof. (5): Using adjointness, it suffices to check that $\big(S(\mathcal{D}_1,\mathcal{D}'_1)\wedge S(\mathcal{D}_2,\mathcal{D}'_2)\big)\otimes(\mathcal{D}_1\cup\mathcal{D}_2)(t)\leq(\mathcal{D}'_1\cup\mathcal{D}'_2)(t)$ holds true for any $t\in\mathrm{Tupl}(T)$. Using (3), the monotony of $\otimes$ and $\wedge$ yields

$$\begin{aligned}
&\big(S(\mathcal{D}_1,\mathcal{D}'_1)\wedge S(\mathcal{D}_2,\mathcal{D}'_2)\big)\otimes(\mathcal{D}_1\cup\mathcal{D}_2)(t)\leq\\
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\wedge(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\big)\otimes(\mathcal{D}_1\cup\mathcal{D}_2)(t)=\\
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\wedge(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\big)\otimes(\mathcal{D}_1(t)\vee\mathcal{D}_2(t)).
\end{aligned}$$

Applying $a\otimes(b\vee c)=(a\otimes b)\vee(a\otimes c)$ to the latter expression, we get

$$\begin{aligned}
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\wedge(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\big)\otimes(\mathcal{D}_1(t)\vee\mathcal{D}_2(t))=\\
&\Big(\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\wedge(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\big)\otimes\mathcal{D}_1(t)\Big)\vee\\
&\Big(\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\wedge(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\big)\otimes\mathcal{D}_2(t)\Big)\leq\\
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\otimes\mathcal{D}_1(t)\big)\vee\big((\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\otimes\mathcal{D}_2(t)\big).
\end{aligned}$$

Using $a\otimes(a\rightarrow b)\leq b$ twice, it follows that

$$\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\otimes\mathcal{D}_1(t)\big)\vee\big((\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\otimes\mathcal{D}_2(t)\big)\leq\mathcal{D}'_1(t)\vee\mathcal{D}'_2(t).$$

Putting previous inequalities together, $\big(S(\mathcal{D}_1,\mathcal{D}'_1)\wedge S(\mathcal{D}_2,\mathcal{D}'_2)\big)\otimes(\mathcal{D}_1\cup\mathcal{D}_2)(t)\leq(\mathcal{D}'_1\cup\mathcal{D}'_2)(t)$ which proves (5).

(6) can be proved analogously as (5). Indeed, for any $t\in\mathrm{Tupl}(T)$,

$$\begin{aligned}
&\big(S(\mathcal{D}_1,\mathcal{D}'_1)\wedge S(\mathcal{D}_2,\mathcal{D}'_2)\big)\otimes(\mathcal{D}_1\cap\mathcal{D}_2)(t)\leq\\
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\wedge(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\big)\otimes(\mathcal{D}_1\cap\mathcal{D}_2)(t),
\end{aligned}$$

Now, using $(a\wedge b)\otimes c\leq(a\otimes c)\wedge(b\otimes c)$ and monotony of $\wedge$ and $\otimes$, we get

$$\begin{aligned}
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\wedge(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\big)\otimes(\mathcal{D}_1(t)\wedge\mathcal{D}_2(t))\leq\\
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\otimes(\mathcal{D}_1(t)\wedge\mathcal{D}_2(t))\big)\wedge\big((\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\otimes(\mathcal{D}_1(t)\wedge\mathcal{D}_2(t))\big)\leq\\
&\big((\mathcal{D}_1(t)\rightarrow\mathcal{D}'_1(t))\otimes\mathcal{D}_1(t)\big)\wedge\big((\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\otimes\mathcal{D}_2(t)\big)\leq\\
&\mathcal{D}'_1(t)\wedge\mathcal{D}'_2(t)=(\mathcal{D}'_1\cap\mathcal{D}'_2)(t)
\end{aligned}$$

which proves (6). Furthermore, (7) can be proved in much the same way as (6) using monotony of $\otimes$.

In case of (8), observe that for any $t\in\mathrm{Tupl}(T)$, we have

$$\begin{aligned}
&S(\mathcal{D}'_1,\mathcal{D}_1)\otimes S(\mathcal{D}_2,\mathcal{D}'_2)\otimes(\mathcal{D}_1\rightarrow\mathcal{D}_2)(t)\leq\\
&(\mathcal{D}'_1(t)\rightarrow\mathcal{D}_1(t))\otimes S(\mathcal{D}_2,\mathcal{D}'_2)\otimes(\mathcal{D}_1(t)\rightarrow\mathcal{D}_2(t))\leq\\
&S(\mathcal{D}_2,\mathcal{D}'_2)\otimes(\mathcal{D}'_1(t)\rightarrow\mathcal{D}_2(t))\leq(\mathcal{D}_2(t)\rightarrow\mathcal{D}'_2(t))\otimes(\mathcal{D}'_1(t)\rightarrow\mathcal{D}_2(t))\leq\\
&\mathcal{D}'_1(t)\rightarrow\mathcal{D}'_2(t)=(\mathcal{D}'_1\rightarrow\mathcal{D}'_2)(t)
\end{aligned}$$

using $(a\rightarrow b)\otimes(b\rightarrow c)\leq(a\rightarrow c)$ from which (8) readily follows. □

Remark 3. The lower estimation provided by (8) differs from the previous inequalities (5)–(7) in that the we use the degree to which $\mathcal{D}'_1$ is included in $\mathcal{D}_1$, i.e., $S(\mathcal{D}'_1, \mathcal{D}_1)$ and not *vice versa.* This is a natural consequence of the fact that $\rightarrow$ is antitone in the first argument whereas the other operations $\cap, \cup, \otimes$ are monotone in both their arguments.

Remark 4. Let us note that inclusion estimations like those from Theorem 1 do not have a nontrivial interpretation in the original Codd's model of data. For instance, if $\mathbf{L}$ (the structure of truth degrees) is a two-element Boolean algebra, the left-hand side of (5) is either 0 or 1. Clearly, $S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) = 1$ iff $\mathcal{D}_1$ is a subset of $\mathcal{D}'_1$ (in the usual sense) and $\mathcal{D}_2$ is a subset of $\mathcal{D}'_2$, from which one immediately derives that $\mathcal{D}_1 \cup \mathcal{D}_2$ is a subset of $\mathcal{D}'_1 \cup \mathcal{D}'_2$. A similar situation applies for (6)–(8).

Notice that using (8), we can derive an estimation formula for the operation of an a-shift. Indeed, we have $S(\mathcal{D}_1, \mathcal{D}_2) \leq S(a \rightarrow \mathcal{D}_1, a \rightarrow \mathcal{D}_2)$, because $a \rightarrow \mathcal{D}_i$ can be seen as a result of $\mathcal{D} \rightarrow \mathcal{D}_i$, where $\mathcal{D}(t) = a$ for all $t \in \mathrm{Tupl}(R)$.

The previous assertion showed estimations for inclusions of RDTs. Estimations for similarities of RDTs can be derived from estimations for inclusions:

Corollary 1. *For $\lozenge$ being $\cap$ or $\cup$, and for $\blacklozenge$ being $\otimes$ or $\rightarrow$, we have:*

$$E(\mathcal{D}_1, \mathcal{D}'_1) \wedge E(\mathcal{D}_2, \mathcal{D}'_2) \leq E(\mathcal{D}_1 \lozenge \mathcal{D}_2, \mathcal{D}'_1 \lozenge \mathcal{D}'_2). \tag{9}$$

$$E(\mathcal{D}_1, \mathcal{D}'_1) \otimes E(\mathcal{D}_2, \mathcal{D}'_2) \leq E(\mathcal{D}_1 \blacklozenge \mathcal{D}_2, \mathcal{D}'_1 \blacklozenge \mathcal{D}'_2). \tag{10}$$

Proof. For $\lozenge$ being $\cap$, (6) applied twice yields: $S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \cap \mathcal{D}_2, \mathcal{D}'_1 \cap \mathcal{D}'_2)$ and $S(\mathcal{D}'_1, \mathcal{D}_1) \wedge S(\mathcal{D}'_2, \mathcal{D}_2) \leq S(\mathcal{D}'_1 \cap \mathcal{D}'_2, \mathcal{D}_1 \cap \mathcal{D}_2)$. Hence, (9) for $\cap$ follows using (2). Analogously, for $\lozenge$ being $\cup$ the claim follows from (5) and (2). For $\blacklozenge$ being $\otimes$ or $\rightarrow$, (7) or (8) used twice together with (2) yield

$$(S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2)) \wedge (S(\mathcal{D}'_1, \mathcal{D}_1) \otimes S(\mathcal{D}'_2, \mathcal{D}_2)) \leq E(\mathcal{D}_1 \blacklozenge \mathcal{D}_2, \mathcal{D}'_1 \blacklozenge \mathcal{D}'_2).$$

Applying $(a \wedge c) \otimes (b \wedge d) \leq (a \otimes b) \wedge (c \otimes d)$ to the previous inequality, we get

$$(S(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}'_1, \mathcal{D}_1)) \otimes (S(\mathcal{D}_2, \mathcal{D}'_2) \wedge S(\mathcal{D}'_2, \mathcal{D}_2)) \leq E(\mathcal{D}_1 \blacklozenge \mathcal{D}_2, \mathcal{D}'_1 \blacklozenge \mathcal{D}'_2),$$

which directly gives the desired inequality. □

As a special case of (10), we obtain an estimation formula for the operation of an a-shift: $E(\mathcal{D}_1, \mathcal{D}_2) \leq E(a \rightarrow \mathcal{D}_1, a \rightarrow \mathcal{D}_2)$. Analogously, one can obtain estimation for further derived unary operations with RDTs like a-negations ($\mathcal{D} \rightarrow a$ which in a special case for $a = 0$ becomes a residuated negation $\mathcal{D} \rightarrow 0$, see [2]) and a-multiples ($a \otimes \mathcal{D}$).

Remark 5. Using the idea in the proof of Corollary 1, in order to prove that operation O preserves similarity, it suffices to check that O preserves (graded) subsethood. Thus, from now on, we shall only investigate whether operations preserve subsethood.

In case of Cartesian products, we have:

Theorem 2. *Let $\mathcal{D}_1$ and $\mathcal{D}'_1$ be RDTs on relation scheme S and let $\mathcal{D}_2$ and $\mathcal{D}'_2$ be RDTs on relation scheme T such that $S \cap T = \emptyset$. Then,*

$$S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \times \mathcal{D}_2, \mathcal{D}'_1 \times \mathcal{D}'_2), \tag{11}$$

Proof (sketch). The proof is analogous to that of (7). □

The following assertion shows that projection and similarity-based selection preserve subsethood degrees (and therefore similarities) of RDTs:

Theorem 3. *Let $\mathcal{D}$ and $\mathcal{D}'$ be RDTs on relation scheme T and let $y \in T$, $d \in D_y$, and $R \subseteq T$. Then,*

$$S(\mathcal{D}, \mathcal{D}') \leq S(\pi_R(\mathcal{D}), \pi_R(\mathcal{D}')), \tag{12}$$
$$S(\mathcal{D}, \mathcal{D}') \leq S(\sigma_{y \approx d}(\mathcal{D}), \sigma_{y \approx d}(\mathcal{D}')). \tag{13}$$

Proof (sketch). In oder to prove (12), we check $S(\mathcal{D}, \mathcal{D}') \otimes (\pi_R(\mathcal{D}))(r) \leq (\pi_R(\mathcal{D}'))(r)$ for any $r \in \mathrm{Tupl}(R)$. It means showing that

$$S(\mathcal{D}, \mathcal{D}') \otimes \textstyle\bigvee_{s \in \mathrm{Tupl}(T \setminus R)} \mathcal{D}(rs) \leq (\pi_R(\mathcal{D}'))(r).$$

Thus, is suffices to prove $S(\mathcal{D}, \mathcal{D}') \otimes \mathcal{D}(rs) \leq (\pi_R(\mathcal{D}'))(r)$ for all $s \in \mathrm{Tupl}(T \setminus R)$ which is indeed true. In case of (13), we proceed analogously. □

Theorem 2 and Theorem 3 used together yield

Corollary 2. *Let $\mathcal{D}_1$ and $\mathcal{D}'_1$ be RDTs on relation scheme S and let $\mathcal{D}_2$ and $\mathcal{D}'_2$ be RDTs on relation scheme T such that $S \cap T = \emptyset$. Then,*

$$S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S(\mathcal{D}_1 \bowtie_{p \approx q} \mathcal{D}_2, \mathcal{D}'_1 \bowtie_{p \approx q} \mathcal{D}'_2). \tag{14}$$

for any $p \in S$ and $q \in T$ having the same domain with similarity. □

In [4], we have shown that in order to have a relational algebra whose expressive power is the same as the expressive power of the domain relational calculus, we have to consider an additional operation of *division*: for $\mathcal{D}_1$ being an RDT on R and $\mathcal{D}_2$ being an RDT on $S \subseteq R$, a *division $\mathcal{D}_1 \div \mathcal{D}_2$ of $\mathcal{D}_1$ by $\mathcal{D}_2$ is an RDT on $T = R \setminus S$*, where

$$(\mathcal{D}_1 \div \mathcal{D}_2)(t) = \textstyle\bigwedge_{s \in \mathrm{Tupl}(S)} \big(\mathcal{D}_2(s) \rightarrow \mathcal{D}_1(st)\big),$$

for each $t \in \mathrm{Tupl}(t)$. In words, $(\mathcal{D}_1 \div \mathcal{D}_2)(t)$ is the degree to which the following proposition holds: "For each tuple s in $\mathcal{D}_2$, the concatenation st is in $\mathcal{D}_1$".

Theorem 4. *Let $\mathcal{D}_1$ and $\mathcal{D}'_1$ be RDTs on relation scheme R and let $\mathcal{D}_2$ and $\mathcal{D}'_2$ be RDTs on relation scheme S such that $S \subseteq R$. Then,*

$$S(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}'_2, \mathcal{D}_2) \leq S(\mathcal{D}_1 \div \mathcal{D}_2, \mathcal{D}'_1 \div \mathcal{D}'_2). \tag{15}$$

Proof. The claim can be shown using similar arguments as in the proof of Theorem 1.

We have shown that the similarity is preserved by all queries that can be formulated in the domain relational calculus [4]. Thus, we have provided a formal justification for the (intuitively expected but nontrivial) fact that similar input data yield similar results of queries.

Remark 6. Note that DRC from [4] is domain dependent and allows infinite RDTs as results of queries as do some of our relational operations. Both DRC and relational algebra can be transformed into domain independent languages by introducing ranges and modifying the relational operations. The similarity estimations provided in this paper apply to the domain independent versions of both the query systems.

4.2 Illustrative Example

Consider again the RDT from Table 1. The RDT can be seen as a result of querying a database of houses for sale where one wants to find a house which is sold for (approximately) `$200,000` and has (approximately) `1200` square feet. The attributes in the RDT are: real estate agent name (agent), house ID (id), square footage (sqft), house age (age), house location ($\mathit{location}$), and house price (price), . In this example, the complete residuated lattice $\mathbf{L} = \langle L, \wedge, \vee, \otimes, \rightarrow, 0, 1\rangle$ serving as the structure of ranks will be the so-called Łukasiewicz algebra [2,18,19]. That is, $L = [0,1]$, $\wedge$ and $\vee$ are minimum and maximum, respectively, and the multiplication and residuum are defined as follows: $a \otimes b = \max(a + b - 1, 0)$ and $a \rightarrow b = \min(1 - a + b, 1)$ for all $a, b \in L$, see Sect. 3.1 for details.

Intuitively, it is natural to consider similarity of values in domains of sqft, age, $\mathit{location}$, and price. For instance, similarity of prices can be defined by

$$p_1 \approx_{\mathit{price}} p_2 = s(|p_2 - p_1|)$$

using a non-increasing scaling function $s\colon [0, \infty) \rightarrow [0, 1]$ with $s(0) = 1$ (i.e., identical prices are fully similar). In particular, for $\approx_{\mathit{price}}$, we have used the following piecewise linear scaling function:

$$s(x) = \begin{cases} 1 - \frac{x}{500\,000}, & \text{if } 0 \leq x \leq 500\,000, \\ 0, & \text{if } x > 500\,000. \end{cases}$$

Analogously, a similarity of locations can be defined based on their geographical distance and/or based on their evaluation (safety, school districts, ...) by an expert. In contrast, there is no need to have similarities for id and agents because end-users do not look for houses based on (similarity of) their (internal) IDs which are kept as keys merely because of performance reasons. Obviously, there may be various reasonable similarity relations defined for the above-mentioned domains and their careful choice is an important task. In this paper, we neither

Table 2. Alternative ranks for houses for sale from Table 1

	agent	*id*	*sqft*	*age*	*location*	*price*
0.93	Brown	138	1185	48	Vestal	$228,500
0.91	Clark	140	1120	30	Endicott	$235,800
0.87	Brown	156	1300	85	Binghamton	$248,600
0.85	Brown	142	950	50	Binghamton	$189,000
0.82	Davis	189	1250	25	Binghamton	$287,300
0.79	Clark	158	1200	25	Vestal	$293,500
0.75	Davis	166	1040	50	Vestal	$286,200
0.37	Davis	112	1890	30	Endicott	$345,000

explain nor recommend particular ways to do so because (i) we try to keep a general view of the problem and (ii) similarities on domains are purpose and user dependent.

Consider now the RDT in Table 2 defined over the same relation scheme as the RDT in Table 1. These two RDTs can be seen as two (slightly different) answers to the same query (when e.g., the domain similarities have been slightly changed) or answers to a modified query (e.g., "show all houses which are sold for (approximately) $210,000 and…"). The similarity of both the RDTs given by (2) is 0.98 (very high). The results in the previous section say that if we perform any (arbitrarily complex) query (using the relational operations we consider in this paper) with Table 2 instead of Table 1, the results will be similar at least to degree 0.98.

For illustration, consider an additional RDT of customers over relation scheme containing two attributes: *name* (customer name) and *budget* (price the customer is willing to pay for a house). In particular, let $\langle$Evans, $250,000$\rangle$, $\langle$Finch, $210,000$\rangle$, and $\langle$Grant, $240,000$\rangle$ be the only tuples in the RDT (all with ranks 1). The answer to the following query

$$\pi_{\{agent, id, price, name, budget\}}(\mathcal{D}_1 \bowtie_{price \approx budget} \mathcal{D}_c),$$

Table 3. Join of Table 1 and the table of customers

	agent	*id*	*price*	*name*	*budget*
0.91	Brown	138	$228,500	Grant	$240,000
0.89	Brown	138	$228,500	Evans	$250,000
0.89	Brown	138	$228,500	Finch	$210,000
0.88	Clark	140	$235,800	Grant	$240,000
0.86	Clark	140	$235,800	Evans	$250,000
0.84	Brown	156	$248,600	Evans	$250,000
⋮	⋮	⋮	⋮	⋮	⋮
0.16	Davis	112	$345,000	Grant	$240,000
0.10	Davis	112	$345,000	Finch	$210,000

Table 4. Results of agent-customer matching for Table 1 and Table 2

	agent	*name*
0.91	Brown	Grant
0.89	Brown	Evans
0.89	Brown	Finch
0.88	Clark	Grant
0.86	Clark	Evans
0.84	Clark	Finch
0.74	Davis	Evans
0.72	Davis	Grant
0.66	Davis	Finch

	agent	*name*
0.91	Brown	Grant
0.90	Clark	Grant
0.89	Brown	Evans
0.89	Brown	Finch
0.88	Clark	Evans
0.86	Clark	Finch
0.75	Davis	Evans
0.73	Davis	Grant
0.67	Davis	Finch

where $\mathcal{D}_1$ stands for Table 1 and $\mathcal{D}_c$ stands for the RDT of customers is in Table 3 (for brevity, some records are omitted). The RDT thus represents an answer to query "show deals for houses sold for (approximately) `$200,000` with (approximately) `1200` square feet and customers so that their budget is similar to the house price". Furthermore, we can obtain an RDT of best agent-customer matching if we project the join onto *agent* and *name*:

$$\pi_{\{agent,name\}}(\mathcal{D}_1 \bowtie_{price\approx budget} \mathcal{D}_c).$$

The result of matching is in Table 4 (left). Due to our results, if we perform the same query with Table 2 instead of Table 1, the new result is guaranteed to be similar with the obtained result at least to degree 0.98. The result for Table 2 is shown in Table 4 (right).

4.3 Tuple-Based Similarity and Further Topics

While the rank-based similarity from Sect. 4.1 can be sufficient in many cases, there are situations where one wants to consider a similarity of RDTs based on ranks and (pairwise) similarity of tuples. For instance, if we take the RDT from Table 1 and make a new one by taking all tuples (keeping their ranks) and increasing the prices by one dollar, we will come up with an RDT which is, according to rank-based similarity, very different from the original one. Intuitively, one would expect to have a high degree of similarity of the RDTs because they differ only by a slight change in price. This issue can be solved by considering the following tuple-based degree of inclusion:

$$S^{\approx}(\mathcal{D}_1,\mathcal{D}_2) = \textstyle\bigwedge_{t\in\mathrm{Tupl}(T)}\big(\mathcal{D}_1(t) \rightarrow \bigvee_{t'\in\mathrm{Tupl}(T)}\big(\mathcal{D}_2(t') \otimes t \approx t'\big)\big), \qquad (16)$$

where $t \approx t' = \bigwedge_{y\in T} t(y) \approx_y t'(y)$ is a similarity of tuples t and t' over T, cf. [5]. In a similar way as in (4), we may define $E^{\approx}$ using $S^{\approx}$ instead of S.

Remark 7. By an easy inspection, $S(\mathcal{D}_1,\mathcal{D}_2) \leq S^{\approx}(\mathcal{D}_1,\mathcal{D}_2)$, i.e. (16) yields an estimate which is at least as high as (3) and analogously for E and $E^{\approx}$. Note that (16) has a natural meaning. Indeed, $S^{\approx}(\mathcal{D}_1,\mathcal{D}_2)$ can be understood as a degree to which the following statement is true: "If t belongs to $\mathcal{D}_1$, then there is t' which is similar to t and which belongs to $\mathcal{D}_2$". Hence, $E^{\approx}(\mathcal{D}_1, \mathcal{D}_2)$ is a degree to which for each tuple from $\mathcal{D}_1$ there is a similar tuple in $\mathcal{D}_2$ and *vice versa*. If $\mathbf{L}$ is a two-element Boolean algebra and each $\approx_y$ is an identity, then $E^{\approx}(\mathcal{D}_1, \mathcal{D}_2) = 1$ iff $\mathcal{D}_1$ and $\mathcal{D}_2$ are identical (in the usual sense).

For tuple-based inclusion (similarity) and for certain relational operations, we can prove analogous preservation formulas as in Sect. 4.1. For instance,

$$S^{\approx}(\mathcal{D}_1, \mathcal{D}'_1) \wedge S(\mathcal{D}_2, \mathcal{D}'_2) \leq S^{\approx}(\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{D}'_1 \cup \mathcal{D}'_2), \tag{17}$$

$$S^{\approx}(\mathcal{D}_1, \mathcal{D}'_1) \otimes S(\mathcal{D}_2, \mathcal{D}'_2) \leq S^{\approx}(\mathcal{D}_1 \times \mathcal{D}_2, \mathcal{D}'_1 \times \mathcal{D}'_2), \tag{18}$$

$$S^{\approx}(\mathcal{D}, \mathcal{D}') \leq S^{\approx}(\pi_R(\mathcal{D}), \pi_R(\mathcal{D}')). \tag{19}$$

On the other hand, similarity-based selection $\sigma_{y \approx d}$ (and, as a consequence, similarity-based join $\bowtie_{p \approx q}$) does not preserve $S^{\approx}$ in general which can be seen as a technical complication. This issue can be overcome by introducing a new type of selection $\sigma^{\approx}_{y \approx d}$ which is *compatible* with $S^{\approx}$. Namely, we can define

$$\big(\sigma^{\approx}_{y \approx d}(\mathcal{D})\big)(t) = \textstyle\bigvee_{t' \in \mathrm{Tupl}(T)} \big(\mathcal{D}(t') \otimes t' \approx t \otimes t(y) \approx_y d\big). \tag{20}$$

For this notion, we can prove that $S^{\approx}(\mathcal{D}, \mathcal{D}') \leq S^{\approx}(\sigma^{\approx}_{y \approx d}(\mathcal{D}), \sigma^{\approx}_{y \approx d}(\mathcal{D}'))$. Similar extension can be done for any relational operation which does not preserve $S^{\approx}$ directly. A detailed description of the extension is postponed to a full version of the paper because of the limited scope.

4.4 Unifying Approach to Similarity of RDTs

In this section, we outline a general approach to similarity of RDTs that includes both the approaches from the previous sections. Interestingly, both (3) and (16) have a common generalization using truth-stressing hedges [19,20]. Truth-stressing hedges represent unary operations on complete residuated lattices (denoted by $*$) that serve as interpretations of logical connectives like "very true", see [19]. Two boundary cases of hedges are (i) identity, i.e. $a^* = a$ ($a \in L$); (ii) globalization: $1^* = 1$, and $a^* = 0$ if $a < 1$. The globalization [32] is a hedge which can be interpreted as "fully true".

Let * be truth-stressing hedge on $\mathbf{L}$. For RDTs $\mathcal{D}_1, \mathcal{D}_2$ on T, we define the degree $S^{\approx}_*(\mathcal{D}_1, \mathcal{D}_2)$ of inclusion of $\mathcal{D}_1$ in $\mathcal{D}_2$ (with respect to *) by

$$S^{\approx}_*(\mathcal{D}_i, \mathcal{D}_j) = \textstyle\bigwedge_{t \in \mathrm{Tupl}(T)} \big(\mathcal{D}_i(t) \rightarrow \bigvee_{t' \in \mathrm{Tupl}(T)} \big(\mathcal{D}_j(t') \otimes (t \approx t')^*\big)\big). \tag{21}$$

Now, it is easily seen that for $*$ being the identity, (21) coincides with (16); if $\approx$ is separating (i.e., $t_1 \approx t_2 = 1$ iff t_1 is identical to t_2) and $*$ is the globalization, (21) coincides with (3). Thus, both (3) and (16) are particular instances of (21) resulting by a choice of the hedge. Note that identity and globalization are two borderline cases of hedges. In general, complete residuated lattices admit other nontrivial hedges that can be used in (21). Therefore, the hedge in (21) serves as a parameter that has an influence on how much emphasis we put on the fact that two tuples are similar. In case of globalization, we put full emphasis, i.e., the tuples are required to be equal to degree 1 (exactly the same if $\approx$ is separating).

If we consider properties needed to prove analogous estimation formulas for general $S^{\approx}_{*}$ as we did in case of S and $S^{\approx}$, we come up with the following important property:

$$(r \approx s)^{*} \otimes (s \approx t)^{*} \leq (r \approx t)^{*}, \tag{22}$$

for every $r, s, t \in \mathrm{Tupl}(T)$ which can be seen as transitivity of $\approx$ with respect to $\otimes$ and *. Consider the following two cases in which (22) is satisfied:

Case 1: $*$ is globalization and $\approx$ is separating. If the left hand side of (22) is nonzero, then $r \approx s = 1$ and $s \approx t = 1$. Separability implies $r = s = t$, i.e. $(r \approx t)^{*} = 1^{*} = 1$, verifying (22).

Case 2: $\approx$ is transitive. In this case, since $a^{*} \otimes b^{*} \leq (a \otimes b)^{*}$ (follows from properties of hedges by standard arguments), transitivity of $\approx$ and monotony of * yield $(r \approx s)^{*} \otimes (s \approx t)^{*} \leq ((r \approx s) \otimes (s \approx t))^{*} \leq (r \approx t)^{*}$.

The following lemma shows that $S^{\approx}_{*}$ and consequently $E^{\approx}_{*}$ have properties that are considered natural for (degrees of) inclusion and similarity:

Lemma 1. *If $\approx$ satisfies (22) with respect to * then*

(i) $S^{\approx}_{*}$ is a reflexive and transitive **L**-*relation, i.e. an* **L**-*quasiorder.*
(ii) $E^{\approx}_{*}$ defined by $E^{\approx}_{*}(\mathcal{D}_1, \mathcal{D}_2) = S^{\approx}_{*}(\mathcal{D}_1, \mathcal{D}_2) \wedge S^{\approx}_{*}(\mathcal{D}_2, \mathcal{D}_1)$ is a reflexive, symmetric, and transitive **L**-*relation, i.e. an* **L**-*equivalence.*

Proof. The assertion follows from results in [2, Sect. 4.2] by taking into account that $\approx^{*}$ is reflexive, symmetric, and transitive with respect to $\otimes$. □

5 Conclusion and Future Research

We have shown that an important subset of relational operations in similarity-based databases preserves various types of similarity. As a result, similarity of query results based on these relational operations can be estimated based on similarity of input data tables before the queries are executed. Furthermore, the results of this paper have shown a desirable important property of the underlying similarity-based model of data: slight changes in input data do not produce huge changes in query results. Functions for similarity estimations based on results in this paper will be incorporated into a relational similarity-based query engine which is currently being developed. Future research will focus on the role of particular relational operations called similarity-based closures that play an important role in tuple-based similarities of RDTs.

References

1. Abiteboul, S., et al.: The lowell database research self-assessment. Commun. ACM **48**(5), 111–118 (2005)
2. Belohlavek, R.: Fuzzy Relational Systems: Foundations and Principles. Kluwer, Academic/Plenum Publishers, New York (2002)
3. Belohlavek, R., Vychodil, V.: Logical foundations for similarity-based databases. In: Chen, L., Liu, C., Liu, Q., Deng, K. (eds.) DASFAA 2009 Workshops. LNCS, vol. 5667, pp. 137–151. Springer, Heidelberg (2009)
4. Belohlavek, R., Vychodil, V.: Query systems in similarity-based databases: logical foundations, expressive power, and completeness. In: Proceedings of the ACM SAC 2010, pp. 1648–1655. ACM Press (2010)
5. Belohlavek, R., Vychodil, V.: Codd's relational model from the point of view of fuzzy logic. J. Logic and Comput. **21**(5), 851–862 (2011)
6. Birkhoff, G.: Lattice theory. American Mathematical Society, Providence (1940)
7. Buckles, B.P., Petry, F.E.: Fuzzy databases in the new era. In: Proceedings of the ACM SAC 1995, Nashville, TN, pp. 497–502. ACM Press (1995)
8. Cavallo, R., Pittarelli, M.: The theory of probabilistic databases. VLDB **1987**, 71–81 (1987)
9. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM **13**(6), 377–387 (1970)
10. Dalvi, N., Ré, C., Suciu, D.: Probabilistic databases: diamonds in the dirt. Commun. ACM **52**, 86–94 (2009)
11. Dalvi, N., Suciu, D.: Efficient query evaluation on probabilistic databases. VLDB J. **16**, 523–544 (2007)
12. Dalvi, N., Suciu, D.: Management of probabilistic data: foundations and challenges. In: ACM PODS 2007, pp. 1–12 (2007)
13. Date, C.J.: Database Relational Model: A Retrospective Review and Analysis. Addison Wesley, Reading (2000)
14. Date, C.J., Darwen, H.: Databases, Types and the Relational Model, 3rd edn. Addison Wesley, Reading (2006)
15. Fagin, R.: Combining fuzzy information: an overview. ACM SIGMOD Record **31**(2), 109–118 (2002)
16. Gerla, G.: Fuzzy Logic. Mathematical Tools for Approximate Reasoning. Kluwer, Dordrecht (2001)
17. Goguen, J.A.: The logic of inexact concepts. Synthese **18**, 325–373 (1968-1969)
18. Gottwald, S.: Mathematical fuzzy logics. Bull. for Symbolic Logic **14**(2), 210–239 (2008)
19. Hájek, P.: Metamathematics of Fuzzy Logic. Kluwer, Dordrecht (1998)
20. Hájek, P.: On very true. Fuzzy Sets Syst. **124**, 329–333 (2001)
21. Illyas, I.F., Aref, W.G., Elmagarmid, A.K.: Supporting top-k join queries in relational databases. VLDB J. **13**, 207–221 (2004)
22. Imieliński, T., Lipski, W.: Incomplete information in relational databases. J. ACM **31**, 761–791 (1984)
23. Koch, C.: On query algebras for probabilistic databases. SIGMOD Record **37**(4), 78–85 (2008)
24. Li, C., Chang, K.C.-C., Ilyas, I.F., Song S.: RankSQL: Query Algebra and Optimization for Relational top-k queries. In: ACM SIGMOD 2005, pp. 131–142. ACM Press (2005)
25. Maier, D.: The Theory of Relational Databases. Comp. Sci. Press, Rockville (1983)

26. Olteanu, D., Huangm, J., Koch, C.: Approximate confidence computation in probabilistic databases. IEEE ICDE **2010**, 145–156 (2010)
27. Pavelka, J.: On fuzzy logic I, II, III. Z. Math. Logik Grundlagen Math. 25, 45–52; 25, 119–134; 25, 447–464 (1979)
28. Raju, K.V.S.V.N., Majumdar, A.K.: Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems. ACM Trans. Database Systems **13**(2), 129–166 (1988)
29. Shenoi, S., Melton, A.: Proximity relations in the fuzzy relational database model. Fuzzy Sets Syst. **100**, 51–62 (1999)
30. Suciu, D., Olteanu, D., Ré, C., Koch, C.: Probabilistic Databases. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2011)
31. Takahashi, Y.: Fuzzy database query languages and their relational completeness theorem. IEEE Trans. Knowl. Data Eng. **5**, 122–125 (1993)
32. Takeuti, G., Titani, S.: Globalization of intuitionistic set theory. Ann. Pure Appl. Logic **33**, 195–211 (1987)
33. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)

A Uniform Fixpoint Approach to the Implementation of Inference Methods for Deductive Databases

Andreas Behrend(✉)

Institute of Computer Science III, University of Bonn,
Römerstraße 164, 53117 Bonn, Germany
behrend@cs.uni-bonn.de

Abstract. Within the research area of deductive databases three different database tasks have been deeply investigated: query evaluation, update propagation and view updating. Over the last thirty years various inference mechanisms have been proposed for realizing these main functionalities of a rule-based system. However, these inference mechanisms have been rarely used in commercial DB systems until now. One important reason for this is the lack of a uniform approach well-suited for implementation in an SQL-based system. In this paper, we present such a uniform approach in form of a new version of the soft consequence operator. Additionally, we present improved transformation-based approaches to query optimization and update propagation and view updating which are all using this operator as underlying evaluation mechanism.

1 Introduction

The notion deductive database refers to systems capable of inferring new knowledge using rules. Within this research area, three main database tasks have been intensively studied: (recursive) query evaluation, update propagation and view updating. Despite of many proposals for efficiently performing these tasks, however, the corresponding methods have been implemented in commercial products (such as, e.g., Oracle or DB2) in a very limited way, so far. One important reason is that many proposals employ inference methods which are not directly suited for being transferred into the SQL world. For example, proof-based methods or instance-oriented model generation techniques (e.g. based on SLDNF) have been proposed as inference methods for view updating which are hardly compatible with the set-oriented bottom-up evaluation strategy of SQL.

In this paper, we present transformation-based methods to query optimization, update propagation and view updating which are well-suited for being transferred to SQL. Transformation-based approaches like Magic Sets [1] automatically transform a given database schema into a new one such that the evaluation of rules over the rewritten schema performs a certain database task more efficiently than with respect to the original schema. These approaches are well-suited for extending database systems, as new algorithmic ideas are solely incorporated into the transformation process, leaving the actual database engine with its own optimization techniques unchanged. In fact, rewriting techniques allow for implementing various database functionalities on the basis of one common inference engine. However, the application of transformation-based approaches with respect to stratifiable views [17] may lead to unstratifiable recursion within the rewritten schemata. Consequently, an elaborate and very expensive inference mechanism is generally required for their evaluation such as the alternating fixpoint computation or the residual program approach proposed by van Gelder [20] resp. Bry [10]. This is also the case for the kind of recursive views proposed by the SQL:1999 standard, as they cover the class of stratifiable views.

As an alternative, the soft consequence operator together with the soft stratification concept has been proposed by the author in [2] which allows for the efficient evaluation of Magic Sets transformed rules. This efficient inference method is applicable to query-driven as well as update-driven derivations. Query-driven inference is typically a top-down process whereas update-driven approaches are usually designed bottom-up. During the last 6 years, the idea of combining the advantages of top-down and bottom-up oriented inference has been consequently employed to enhance existing methods to query optimization [3] as well as update propagation [6] and to develop a new approach to view updating. In order to handle alternative derivations that may occur in view updating methods, an extended version of the original soft consequence operator has to be developed. In this paper, this new version is presented, which is well-suited for efficiently determining the semantics of definite and indefinite databases but remains compatible with the set-oriented, bottom-up evaluation of SQL.

2 Basic Concepts

A Datalog *rule* is a function-free clause of the form $H_1 \leftarrow L_1 \wedge \cdots \wedge L_m$ with $m \geq 1$ where H_1 is an atom denoting the rule's head, and $L_1, \ldots, L_m$ are literals, i.e. positive or negative atoms, representing its body. We assume all deductive rules to be *safe*, i.e., all variables occurring in the head or in any negated literal of a rule must be also present in a positive literal in its body. If $A \equiv p(t_1, \ldots, t_n)$ with $n \geq 0$ is a literal, we use $\texttt{vars}(A)$ to denote the set of variables occurring in A and $\texttt{pred}(A)$ to refer to the predicate symbol p of A. If A is the head of a given rule R, we use $\texttt{pred}(R)$ to refer to the predicate symbol of A. For a set of rules $\mathcal{R}$, $\texttt{pred}(\mathcal{R})$ is defined as $\cup_{r \in \mathcal{R}} \{\texttt{pred}(r)\}$. A *fact* is a ground atom in which every t_i is a constant.

A *deductive database* $\mathcal{D}$ is a triple $\langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ where $\mathcal{F}$ is a finite set of facts (called *base facts*), $\mathcal{I}$ is a finite set of integrity constraints (i.e., positive ground atoms) and $\mathcal{R}$ a finite set of rules such that $\texttt{pred}(\mathcal{F}) \cap \texttt{pred}(\mathcal{R}) = \emptyset$ and $\texttt{pred}(\mathcal{I}) \subseteq \texttt{pred}(\mathcal{F} \cup \mathcal{R})$. Within a deductive database $\mathcal{D}$, a predicate symbol p is called derived (view predicate), if $p \in \texttt{pred}(\mathcal{R})$. The predicate p is called extensional (or base predicate), if $p \in \texttt{pred}(\mathcal{F})$. Let $\mathcal{H}_\mathcal{D}$ be the Herbrand base of $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$. The set of all derivable literals from $\mathcal{D}$ is defined as the well-founded model [21] for $(\mathcal{F} \cup \mathcal{R})$: $\mathcal{M}_\mathcal{D} := I^+ \cup \neg \cdot I^-$ where $I^+, I^- \subseteq \mathcal{H}_\mathcal{D}$ are sets of ground atoms and $\neg \cdot I^-$ includes all negations of atoms in I^-. The set I^+ represents the positive portion of the well-founded model while $\neg \cdot I^-$ comprises all negative conclusions. The semantics of a database $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ is defined as the well-founded model $\mathcal{M}_{\mathcal{D}} := I^+ \cup \neg \cdot I^-$ for $\mathcal{F} \cup \mathcal{R}$ if all integrity constraints are satisfied in $\mathcal{M}_{\mathcal{D}}$, i.e., $\mathcal{I} \subseteq I^+$. Otherwise, the semantics of $\mathcal{D}$ is undefined. For the sake of simplicity of exposition, and without loss of generality, we assume that a predicate is either base or derived, but not both, which can be easily achieved by rewriting a given database.

Disjunctive Datalog extends Datalog by disjunctions of literals in facts as well as rule heads. A disjunctive Datalog rule is a function-free clause of the form $A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n$ with $m, n \geq 1$ where the rule's head $A_1 \vee \ldots \vee A_m$ is a disjunction of positive atoms, and the rule's body $B_1, \ldots, B_n$ consists of literals, i.e. positive or negative atoms. A disjunctive fact $f \equiv f_1 \vee \ldots \vee f_k$ is a disjunction of ground atoms f_i with $i \geq 1$. f is called definite if $i = 1$. We solely consider stratifiable disjunctive rules only, that is, recursion through negative predicate occurrences is not permitted [17]. A stratification partitions a given rule set such that all positive derivations of relations can be determined before a negative literal with respect to one of those relations is evaluated. The semantics of a stratifiable disjunctive databases $\mathcal{D}$ is defined as the perfect model state $\mathcal{PM}_{\mathcal{D}}$ of $\mathcal{D}$ iff $\mathcal{D}$ is consistent [4, 11].

3 Transformation-Based Approaches

The need for a uniform inference mechanism in deductive databases is motivated by the fact that transformation-based approaches to query optimization, update propagation and view updating are still based on very different model generators. In this section, we briefly recall the state-of-the-art with respect to these transformation-based techniques by means of Magic Sets, Magic Updates and Magic View Updates. The last two approaches have been already proposed by the author in [6, 7]. Note that we solely consider stratifiable rules for the given (external) schema. The transformed internal schema, however, may not always be stratifiable such that more general inference engines are required.

3.1 Query Optimization

Various methods for efficient bottom-up evaluation of queries against the intensional part of a database have been proposed, e.g. Magic Sets [1], Counting [9], Alexander method [19]. All these approaches are rewriting techniques for deductive rules with respect to a given query such that bottom-up materialization is performed in a goal-directed manner cutting down the number of irrelevant facts generated. In the following we will focus on Magic Sets as this approach has been accepted as a kind of standard in the field.

Magic Sets rewriting is a two-step transformation in which the first phase consists of constructing an adorned rule set, while the second phase consists of the actual Magic Sets rewriting. Within an adorned rule set, the predicate symbol of a literal is associated with an adornment which is a string consisting of letters b and f. While b represents a bound argument at the time when the literal is to be evaluated, f denotes a free argument. The adorned version of the deductive rules is constructed with respect to an adorned query and a selected sip strategy [18] which basically determines for each rule the order in which the body literals are to be evaluated and which bindings are passed on to the next literal. During the second phase of Magic Sets the adorned rules are rewritten such that bottom-up materialization of the resulting database simulates a top-down evaluation of the original query on the original database. For this purpose, each adorned rule is extended with a magic literal restricting the evaluation of the rule to the given binding in the adornment of the rule's head. The magic predicates themselves are defined by rules which define the set of relevant selection constants. The initial values corresponding to the query are given by the so-called magic seed. As an example, consider the following stratifiable rules $\mathcal{R}$

$$\begin{aligned}
&\mathtt{o(X,Y)} \leftarrow \neg \mathtt{p(Y,X)} \wedge \mathtt{p(X,Y)}\\
&\mathtt{p(X,Y)} \leftarrow \mathtt{e(X,Y)}\\
&\mathtt{p(X,Y)} \leftarrow \mathtt{e(X,Z)} \wedge \mathtt{p(Z,Y)}
\end{aligned}$$

and the query $?-\mathtt{o(1,2)}$ asking whether a path from node 1 to 2 exists but not vice versa. Assuming a full left-to-right sip strategy, Magic Sets yields the following deductive rules $\mathcal{R}_{ms}$

$$\begin{array}{ll}
\mathtt{o_{bb}(X,Y)} \leftarrow \mathtt{m\_o_{bb}(X,Y)} \wedge \neg \mathtt{p_{bb}(Y,X)} \wedge \mathtt{p_{bb}(X,Y)} & \mathtt{p_{bb}(X,Y)} \leftarrow \mathtt{m\_p_{bb}(X,Y)} \wedge \mathtt{e(X,Y)}\\
\mathtt{p_{bb}(X,Y)} \leftarrow \mathtt{m\_p_{bb}(X,Y)} \wedge \mathtt{e(X,Z)} \wedge \mathtt{p_{bb}(Z,Y)} & \mathtt{m\_p_{bb}(Y,X)} \leftarrow \mathtt{m\_o_{bb}(X,Y)}\\
\mathtt{m\_p_{bb}(X,Y)} \leftarrow \mathtt{m\_o_{bb}(X,Y)} \wedge \neg \mathtt{p_{bb}(Y,X)} & \mathtt{m\_o_{bb}(X,Y)} \leftarrow \mathtt{m\_s\_o_{bb}(X,Y)}\\
\mathtt{m\_p_{bb}(Z,Y)} \leftarrow \mathtt{m\_p_{bb}(X,Y)} \wedge \mathtt{e(X,Z)} &
\end{array}$$

as well as the magic seed fact $\mathtt{m\_s\_o_{bb}(1,2)}$. The Magic Sets transformation is sound for stratifiable databases. However, the resulting rule set may be no more stratifiable (as is the case in the above example) and more general approaches than iterated fixpoint computation are needed. For determining the well-founded model of general logic programs, the alternating fixpoint computation by Van Gelder [20] or the conditional fixpoint by Bry [10] could be used. The application of these methods, however, is not really efficient because the specific reason for the unstratifiability of the transformed rule sets is not taken into account. As an efficient alternative, the soft stratification concept together with the soft consequence operator [2] could be used for determining the positive part of the well-founded model (cf. Sect. 4).

3.2 Update Propagation

Determining the consequences of base relation changes is essential for maintaining materialized views as well as for efficiently checking integrity. Update propagation (UP) methods have been proposed aiming at the efficient computation of implicit changes of derived relations resulting from explicitly performed updates of extensional facts [13,14,16,17]. We present a specific method for update propagation which fits well with the semantics of deductive databases and is based on the soft consequence operator again. We will use the notion *update* to denote the 'true' changes caused by a transaction only; that is, we solely consider sets of updates where compensation effects (i.e., given by an insertion and deletion of the same fact or the insertion of facts which already existed, for example) have already been taken into account.

The task of update propagation is to systematically compute the set of all induced modifications starting from the physical changes of base data. Technically, this is a set of delta facts for any affected relation which may be stored in corresponding delta relations. For each predicate symbol $p \in \mathtt{pred}(\mathcal{D})$, we will use a pair of delta relations $\langle \Delta^+_{\mathtt{p}}, \Delta^-_{\mathtt{p}} \rangle$ representing the insertions and deletions induced on p by an update on $\mathcal{D}$. The initial set of delta facts directly results from the given update and represents the so-called UP seeds. They form the starting point from which induced updates, represented by derived delta relations, are computed. In our transformation-based approach, so-called *propagation rules* are employed for computing delta relations. A propagation rule refers to at least one delta relation in its body in order to provide a focus on the underlying changes when computing induced updates. For showing the effectiveness of an induced update, however, references to the state of a relation before and after the base update has been performed are necessary. As an example of this propagation approach, consider again the rules for relation p from Subsect. 3.1. The UP rules $\mathcal{R}^\Delta$ with respect to insertions into e are as follows:

$$\begin{aligned}
\Delta^+_{\mathtt{p}}(\mathtt{X},\mathtt{Y}) &\leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Y}) \wedge \neg \mathtt{p}^{\mathtt{old}}(\mathtt{X},\mathtt{Y})\\
\Delta^+_{\mathtt{p}}(\mathtt{X},\mathtt{Y}) &\leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{p}^{\mathtt{new}}(\mathtt{Z},\mathtt{Y}) \wedge \neg \mathtt{p}^{\mathtt{old}}(\mathtt{X},\mathtt{Y})\\
\Delta^+_{\mathtt{p}}(\mathtt{X},\mathtt{Y}) &\leftarrow \Delta^+_{\mathtt{p}}(\mathtt{Z},\mathtt{Y}) \wedge \mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Z}) \wedge \neg \mathtt{p}^{\mathtt{old}}(\mathtt{X},\mathtt{Y})
\end{aligned}$$

For each relation p we use p^{old} to refer to its old state before the changes given in the delta relations have been applied whereas p^{new} is used to refer to the new state of p. These state relations are never completely computed but are queried with bindings from the delta relation in the propagation rule body and thus act as a test of effectiveness. In the following, we assume the old database state to be present such that the adornment *old* can be omitted. For simulating the new database state from a given update so called *transition rules* [16] are used. The transition rules $\mathcal{R}^\Delta_\tau$ for simulating the required new states of e and p are:

$$\begin{aligned}
\mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) &\leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y}) \wedge \neg \Delta^-_{\mathtt{e}}(\mathtt{X},\mathtt{Y}) & \qquad \mathtt{p}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) &\leftarrow \mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Y})\\
\mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) &\leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X},\mathtt{Y}) & \qquad \mathtt{p}^{\mathtt{new}}(\mathtt{X},\mathtt{Y}) &\leftarrow \mathtt{e}^{\mathtt{new}}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{p}^{\mathtt{new}}(\mathtt{Z},\mathtt{Y})
\end{aligned}$$

Note that the new state definition of intensional predicates only indirectly refers to the given update in contrast to extensional predicates. If $\mathcal{R}$ is stratifiable, the rule set $\mathcal{R} \uplus \mathcal{R}^\Delta \uplus \mathcal{R}^\Delta_\tau$ will be stratifiable, too (cf. [6]). As $\mathcal{R} \uplus \mathcal{R}^\Delta \uplus \mathcal{R}^\Delta_\tau$ remains to be stratifiable, iterated fixpoint computation could be employed for determining the semantics of these rules and the induced updates defined by them. However, all state relations are completely determined which leads to a very inefficient propagation process. The reason is that the supposed evaluation over the two consecutive database states is performed using deductive rules which are not specialized with respect to the particular updates that are propagated. This weakness of propagation rules in view of a bottom-up materialization will be cured by incorporating Magic Sets.

Magic Updates

The aim is to develop an UP approach which is automatically limited to the affected delta relations. The evaluation of side literals and effectiveness tests is restricted to the updates currently propagated. We use the Magic Sets approach for incorporating a top-down evaluation strategy by considering the currently propagated updates in the dynamic body literals as abstract queries on the remainder of the respective propagation rule bodies. Evaluating these propagation queries has the advantage that the respective state relations will only be partially materialized. As an example, let us consider the specific deductive database $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ with $\mathcal{R}$ consisting of the well-known rules for the transitive closure $\mathtt{p}$ of relation $\mathtt{e}$:

$$\begin{array}{ll} \underline{\mathcal{R}:} & \mathtt{p(X,Y)} \leftarrow \mathtt{e(X,Y)} \\ & \mathtt{p(X,Y)} \leftarrow \mathtt{e(X,Z),p(Z,Y)} \end{array}$$

$$\begin{array}{ll} \underline{\mathcal{F}:} & \mathtt{edge(1,2)},\ \mathtt{edge(1,4)},\ \mathtt{edge(3,4)} \\ & \mathtt{edge(10,11)},\ \mathtt{edge(11,12)},\ \ldots,\ \mathtt{edge(98,99)},\ \mathtt{edge(99,100)} \end{array}$$

Note that the derived relation $\mathtt{p}$ consists of 4098 tuples. Suppose a given update contains the new tuple $e(2,3)$ to be inserted into $\mathcal{D}$ and we are interested in finding the resulting consequences for $\mathtt{p}$. Computing the induced update by evaluating the stratifiable propagation and transition rules would lead to the generation of 94 new state facts for relation $\mathtt{e}$, 4098 old state facts for $\mathtt{p}$ and $4098 + 3$ new state facts for $\mathtt{p}$. The entire number of generated facts is 8296 for computing the three induced insertions $\{\Delta^+_{\mathtt{p}}(\mathtt{1,3}), \Delta^+_{\mathtt{p}}(\mathtt{2,3}), \Delta^+_{\mathtt{p}}(\mathtt{2,4})\}$ with respect to $\mathtt{p}$.

However, the application of the Magic Updates rewriting with respect to the propagation queries $\{\Delta^+_{\mathtt{p}}(\mathtt{Z,Y}), \Delta^+_{\mathtt{e}}(\mathtt{X,Y}), \Delta^+_{\mathtt{e}}(\mathtt{X,Z})\}$ provides a much better focus on the changes to $\mathtt{e}$. Within its application, the following subquery rules

$$\begin{array}{ll} \mathtt{m\_p^{new}_{bf}(Z)} \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X,Z}) & \mathtt{m\_p_{bb}(X,Y)} \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X,Y}) \\ \mathtt{m\_e^{new}_{fb}(Z)} \leftarrow \Delta^+_{\mathtt{p}}(\mathtt{Z,Y}) & \mathtt{m\_p_{bb}(X,Y)} \leftarrow \Delta^+_{\mathtt{e}}(\mathtt{X,Z}) \wedge \mathtt{p^{new}_{bf}(Z,Y)} \\ & \mathtt{m\_p_{bb}(X,Y)} \leftarrow \Delta^+_{\mathtt{p}}(\mathtt{Z,Y}) \wedge \mathtt{e^{new}_{fb}(X,Z)} \end{array}$$

are generated. The respective queries $Q = \{m\_e^{new}_{fb}, m\_p^{new}_{bf}, \ldots\}$ allow to specialize the employed transition rules, e.g.

$$\begin{array}{l} \mathtt{e^{new}_{fb}(X,Y)} \leftarrow \mathtt{m\_e^{new}_{fb}(Y)} \wedge \mathtt{e(X,Y)} \wedge \neg \Delta^-_{\mathtt{e}}(\mathtt{X,Y}) \\ \mathtt{e^{new}_{fb}(X,Y)} \leftarrow \mathtt{m\_e^{new}_{fb}(Y)} \wedge \Delta^+_{\mathtt{e}}(\mathtt{X,Y}) \end{array}$$

such that only relevant state tuples are generated. We denote the Magic Updates transformed rules $\mathcal{R} \uplus \mathcal{R}^{\Delta} \uplus \mathcal{R}^{\Delta}_{\tau}$ by $\mathcal{R}^{\Delta}_{mu}$. Despite of the large number of rules in

$\mathcal{R}^{\Delta}_{mu}$, the number of derived results remains relatively small. Quite similar to the Magic sets approach, the Magic Updates rewriting may result in an unstratifiable rule set. This is also the case for our example where the following negative cycle occurs in the respective dependency graph:

$$\Delta^+_{\mathtt{p}} \xrightarrow{pos} \mathtt{m\_p_{bb}} \xrightarrow{pos} \mathtt{p_{bb}} \xrightarrow{neg} \Delta^+_{\mathtt{p}}$$

In [6] it has been shown, however, that the resulting rules must be at least softly stratifiable such that the soft consequence operator could be used for efficiently computing their well-founded model. Computing the induced update by evaluating the Magic Updates transformed rules leads to the generation of two new state facts for e, one old state fact and one new state fact for p. The entire number of generated facts is 19 in contrast to 8296 for computing the three induced insertions with respect to p.

3.3 View Updates

Bearing in mind the numerous benefits of the afore mentioned methods to query optimization and update propagation, it seemed worthwhile to develop a similar, i.e., incremental and transformation-based, approach to the dual problem of view updating. In contrast to update propagation, view updating aims at determining one or more base relation updates such that all given update requests with respect to derived relations are satisfied after the base updates have been successfully applied. In the following, we recall a transformation-based approach to incrementally compute such base updates for stratifiable databases proposed by the author in [7]. The approach extends and integrates standard techniques for efficient query answering, integrity checking and update propagation. The analysis of view updating requests usually leads to alternative view update realizations which are represented in disjunctive form.

Magic View Updates

In our transformation-based approach, true view updates (VU) are considered only, i.e., ground atoms which are presently not derivable for atoms to be inserted, or are derivable for atoms to be deleted, respectively. A method for view updating determines sets of alternative updates (called VU realization) satisfying a given request. There may be infinitely many realizations and even realizations of infinite size which satisfy a given VU request. In our approach, a breadth-first search is employed for determining a set of minimal realizations. A realization is minimal in the sense that none of its updates can be removed without losing the property of being a realization. As each level of the search tree is completely explored, the result usually consists of more than one realization. If only VU realizations of infinite size exist, our method will not terminate.

Given a VU request, view updating methods usually determine subsequent VU requests in order to find relevant base updates. Similar to delta relations for UP we will use the notion *VU relation* to access individual view updates with respect to the relations of our system. For each relation $p \in \mathtt{pred}(\mathcal{R} \cup \mathcal{F})$ we use the VU relation $\nabla^+_{\mathtt{p}}(\vec{x})$ for tuples to be inserted into $\mathcal{D}$ and $\nabla^-_{\mathtt{p}}(\vec{x})$ for tuples to be deleted from $\mathcal{D}$. The initial set of delta facts resulting from a given VU request is again represented by so-called VU seeds. Starting from the seeds, so-called VU rules are employed for finding subsequent VU requests systematically. These rules perform a top-down analysis in a similar way as the bottom-up analysis implemented by the UP rules. As an example, consider the following database $\mathcal{D} = \langle \mathcal{F}, \mathcal{R}, \mathcal{I} \rangle$ with $\mathcal{F} = \{r_2(2), s(2)\}$, $\mathcal{I} = \{ic(2)\}$ and the rules $\mathcal{R}$:

$$\begin{array}{ll}
\mathtt{p(X)} \leftarrow \mathtt{q_1(X)} & \mathtt{q_1(X)} \leftarrow \mathtt{r_1(X)} \wedge \mathtt{s(X)} \\
\mathtt{p(X)} \leftarrow \mathtt{q_2(X)} & \mathtt{q_2(X)} \leftarrow \mathtt{r_2(X)} \wedge \neg \mathtt{s(X)} \\
\mathtt{ic(2)} \leftarrow \neg \mathtt{au(2)} & \mathtt{au(X)} \leftarrow \mathtt{q_2(X)} \wedge \neg \mathtt{q_1(X)}
\end{array}$$

The corresponding set of VU rules $\mathcal{R}^\nabla$ with respect to $\nabla^+_{\mathtt{p}}(2)$ is given by:

$$\begin{array}{ll}
\nabla^+_{\mathtt{q_1}}(\mathtt{X}) \vee \nabla^+_{\mathtt{q_1}}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{p}}(\mathtt{X}) & \\
\nabla^+_{\mathtt{r_1}}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q_1}}(\mathtt{X}) \wedge \neg \mathtt{r_1(X)} & \nabla^+_{\mathtt{r_2}}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q_2}}(\mathtt{X}) \wedge \neg \mathtt{r_2(X)} \\
\nabla^+_{\mathtt{s}}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q_1}}(\mathtt{X}) \wedge \neg \mathtt{s(X)} & \nabla^-_{\mathtt{s}}(\mathtt{X}) \leftarrow \nabla^+_{\mathtt{q_2}}(\mathtt{X}) \wedge \mathtt{s(X)}
\end{array}$$

In contrast to the UP rules from Sect. 3.2, no explicit references to the new database state are included in the above VU rules. The reason is that these rules are applied iteratively over several intermediate database states before the minimal set of realizations has been found. Hence, the apparent references to the old state really refer to the current state which is continuously modified while computing VU realizations. These predicates solely act as tests again queried with respect to bindings from VU relations and thus will never be completely evaluated.

Evaluating these rules using model generation with disjunctive facts leads to two alternative updates, insertion $\{r_1(2)\}$ and deletion $\{s(2)\}$, induced by the derived disjunction $\nabla^+_{\mathtt{r_1}}(2) \vee \nabla^-_{\mathtt{s}}(2)$. Obviously, the second update represented by $\nabla^-_{\mathtt{s}}(2)$ would lead to an undesired side effect by means of an integrity violation. In order to provide a complete method, however, such erroneous/incomplete paths must be also explored and side effects repaired if possible. Determining whether a computed update will lead to a consistent database state or not can be done by applying a bottom-up UP process at the end of the top-down phase leading to an irreparable constraint violation with respect to $\nabla^-_{s}(2)$:

$$\nabla^-_{\mathtt{s}}(2) \Rightarrow \Delta^+_{\mathtt{q_2}}(2) \Rightarrow \Delta^+_{\mathtt{p}}(2), \Delta^+_{\mathtt{au}}(2) \Rightarrow \Delta^-_{\mathtt{ic}}(2) \rightsquigarrow \mathit{false}$$

In order to see whether the violated constraint can be repaired, the subsequent view update request $\nabla^+_{\mathtt{ic}}(2)$ with respect to $\mathcal{D}$ ought to be answered. The application of $\mathcal{R}^\nabla$ yields

$$\begin{array}{ll}
 & \Rightarrow \nabla^-_{\mathtt{q_2}}(2), \nabla^+_{\mathtt{q_2}}(2) \rightsquigarrow \mathit{false} \\
\nabla^+_{\mathtt{ic}}(2) \Rightarrow \nabla^-_{\mathtt{aux}}(2) \Updownarrow & \\
 & \Rightarrow \nabla^+_{\mathtt{q_1}}(2) \Rightarrow \nabla^+_{\mathtt{s}}(2), \nabla^-_{\mathtt{s}}(2) \rightsquigarrow \mathit{false}
\end{array}$$

showing that this request cannot be satisfied as inconsistent subsequent view update requests are generated on this path. Such erroneous derivation paths will be indicated by the keyword *false*. The reduced set of updates - each of them leading to a consistent database state only - represents the set of realizations $\Delta^+_{\mathtt{r}_1}(\mathtt{2})$.

An induced deletion of an integrity constraint predicate can be seen as a side effect of an 'erroneous' VU. Similar side effects, however, can be also found when induced changes to the database caused by a VU request may include derived facts which had been actually used for deriving this view update. This effect is shown in the following example for a deductive database $\mathcal{D} = \langle \mathcal{R}, \mathcal{F}, \mathcal{I} \rangle$ with $\mathcal{R} = \{\mathtt{h(X)} \leftarrow \mathtt{p(X)} \wedge \mathtt{q(X)} \wedge \mathtt{i}, \mathtt{i} \leftarrow \mathtt{p(X)} \wedge \neg \mathtt{q(X)}\}$, $\mathcal{F} = \{\mathtt{p(1)}\}$, and $\mathcal{I} = \emptyset$. Given the VU request $\nabla^+_{\mathtt{h}}(\mathtt{1})$, the overall evaluation scheme for determining the only realization $\{\Delta^+_{\mathtt{q}}(1), \Delta^+_{\mathtt{p}}(c^{new_1})\}$ would be as follows:

$$\begin{array}{rl} & \Rightarrow \nabla^+_{\mathtt{p}}(\mathtt{c}^{\mathtt{new}_1}) \\ \nabla^+_{\mathtt{h}}(\mathtt{1}) \Rightarrow \nabla^+_{\mathtt{q}}(\mathtt{1}) \Rightarrow \Delta^+_{\mathtt{q}}(\mathtt{1}) \Rightarrow \Delta^-_{\mathtt{i}} \Rightarrow \nabla^+_{\mathtt{i}} & \Updownarrow \\ & \Rightarrow \nabla^-_{\mathtt{q}}(\mathtt{1}), \nabla^+_{\mathtt{q}}(\mathtt{1}) \leadsto false \end{array}$$

The example shows the necessity of compensating side effects, i.e., the compensation of the 'deletion' Δ^-_i (that prevents the 'insertion' $\Delta^+_h(1)$) caused by the tuple $\nabla^+_q(1)$. In general the compensation of side effects, however, may in turn cause additional side effects which have to be 'repaired'. Thus, the view updating method must alternate between top-down and bottom-up phases until all possibilities for compensating side effects (including integrity constraint violations) have been considered, or a solution has been found. To this end, so-called *VU transition rules* $\mathcal{R}^{\nabla}_{\tau}$ are used for restarting the VU analysis. For example, the compensation of violated integrity constraints can be realized by using the following kind of transition rule $\Delta^-_{ic}(\vec{c}) \rightarrow \nabla^+_{ic}(\vec{c})$ for each ground literal $ic(\vec{c}) \in \mathcal{I}$. VU transition rules make sure that erroneous solutions are evaluated to *false* and side effects are repaired.

Having the rules for the direct and indirect consequences of a given VU request, a general application scheme for systematically determining VU realizations can be defined (see [7] for details). Instead of using simple propagation rules $\mathcal{R} \uplus \mathcal{R}^{\Delta} \uplus \mathcal{R}^{\Delta}_{\tau}$, however, it is much more efficient to employ the corresponding Magic Update rules. The top-down analysis rules $\mathcal{R} \uplus \mathcal{R}^{\nabla}$ and the bottom-up consequence analysis rules $\mathcal{R}^{\Delta}_{mu} \uplus \mathcal{R}^{\nabla}_{\tau}$ are alternating applied. Note that the disjunctive rules $\mathcal{R} \uplus \mathcal{R}^{\nabla}$ are stratifiable while $\mathcal{R}^{\Delta}_{mu} \uplus \mathcal{R}^{\nabla}_{\tau}$ is softly stratifiable such that a perfect model state [4,11] and a well-founded model generation must alternately be applied. The iteration stops as soon as a realization for the given VU request has been found. The correctness of this approach has been already shown in [7].

4 Consequence Operators and Fixpoint Computations

In the following, we summarize the most important fixpoint-based approaches for definite as well as indefinite rules. All these methods employ so-called consequence operators which formalize the application of deductive rules for deriving new data. Based on their properties, a new uniform consequence operator is developed subsequently.

4.1 Definite Rules

First, we recall the iterated fixpoint method for constructing the well-founded model of a stratifiable database which coincides with its perfect model [17].

Definition 1. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a deductive database, λ a stratification on $\mathcal{D}$, $\mathcal{R}_1 \uplus \ldots \uplus \mathcal{R}_n$ the partition of $\mathcal{R}$ induced by λ, $I \subseteq \mathcal{H}_\mathcal{D}$ a set of ground atoms, and $[[\mathcal{R}]]_I$ the set of all ground instances of rules in $\mathcal{R}$ with respect to the set I. Then we define*

1. *the immediate consequence operator $T_\mathcal{R}(I)$ as*

$$\begin{aligned} T_\mathcal{R}(I) := \{H \mid \quad & H \in I \vee \exists r \in [[\mathcal{R}]]_I : r \equiv H \leftarrow L_1 \wedge \ldots \wedge L_n \\ & \textit{such that } L_i \in I \textit{ for all positive literals } L_i \\ & \textit{and } L \notin I \textit{ for all negative literals } L_j \equiv \neg L\}, \end{aligned}$$

2. *the iterated fixpoint M_n as the last Herbrand model of the sequence*

$$M_1 := \mathtt{lfp}(T_{\mathcal{R}_1}, \mathcal{F}), M_2 := \mathtt{lfp}(T_{\mathcal{R}_2}, M_1), \ldots, M_n := \mathtt{lfp}(T_{\mathcal{R}_n}, M_{n-1}),$$

   where $\mathtt{lfp}$ $(T_\mathcal{R}, \mathcal{F})$ *denotes the least fixpoint of operator $T_\mathcal{R}$ containing $\mathcal{F}$.*

3. *and the iterated fixpoint model $\mathcal{M}^i_\mathcal{D}$ as*
   $\mathcal{M}^i_\mathcal{D} := M_n \uplus \neg \cdot \overline{M_n}$.

This constructive definition of the iterated fixpoint model is based on the immediate consequence operator introduced by van Emden and Kowalski. In [17] it has been shown that the perfect model of a stratifiable database $\mathcal{D}$ is identical with the iterated fixpoint model $\mathcal{M}^i_\mathcal{D}$ of $\mathcal{D}$.

Stratifiable rules represent the most important class of deductive rules as they cover the expressiveness of recursion in SQL:1999. Our transformation-based approaches, however, may internally lead to unstratifiable rules for which a more general inference method is necessary. In case that unstratifiability is caused by the application of Magic Sets, the so-called soft stratification approach proposed by the author in [2] could be used.

Definition 2. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a deductive database, λ^s a soft stratification on $\mathcal{D}$, $\mathcal{P} = P_1 \uplus \ldots \uplus P_n$ the partition of $\mathcal{R}$ induced by λ^s, and $I \subseteq \mathcal{H}_\mathcal{D}$ a set of ground atoms. Then we define*

1. *the soft consequence operator $T^s_\mathcal{P}(I)$ as*

$$T^s_\mathcal{P}(I) := \begin{cases} I & \textit{if } T_{P_j}(I) = I \textit{ for all } j \in \{1, \ldots, n\} \\ T_{P_i}(I) & \textit{with } i \;=\; \mathtt{min}\{j \mid T_{P_j}(I) \supsetneq I\}, \textit{ otherwise.} \end{cases}$$

   where T_{P_i} denotes the immediate consequence operator.

2. and the soft fixpoint model $\mathcal{M}^s_\mathcal{D}$ as

$$\mathcal{M}^s_\mathcal{D} := \texttt{lfp}\ (T^s_\mathcal{P}, \mathcal{F}) \cup \neg\, \overline{(\texttt{lfp}\ (T^s_\mathcal{P}, \mathcal{F}))}.$$

Note that the sequence operator is based upon the immediate consequence operator and can even be used to determine the iterated fixpoint model of a stratifiable database [6]. As an even more general alternative, the alternating fixpoint model for arbitrary unstratifiable rules has been proposed in [12] on the basis of the eventual consequence operator.

Definition 3. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a deductive database, $I^+, I^- \subseteq \mathcal{H}_\mathcal{D}$ sets of ground atoms, and $[[\mathcal{R}]]_{I^+}$ the set of all ground instances of rules in $\mathcal{R}$ with respect to the set I^+. Then we define*

1. the eventual consequence operator $\widehat{T}_\mathcal{R}\langle I^- \rangle$ as

$$\begin{aligned}\widehat{T}_\mathcal{R}\langle I^- \rangle(I^+) := \{H \mid \quad & H \in I^+ \vee \exists r \in [[\mathcal{R}]]_{I^+} : r \equiv H \leftarrow L_1 \wedge \ldots \wedge L_n \\ & \textit{such that } L_i \in I^+ \textit{ for all positive literals } L_i \\ & \textit{and } L \notin I^- \textit{ for all negative literals } L_j \equiv \neg L\},\end{aligned}$$

2. the eventual consequence transformation $\widehat{S}_\mathcal{D}$ as

$$\widehat{S}_\mathcal{D}(I^-) := \ \texttt{lfp}(\widehat{T}_\mathcal{R}\langle I^- \rangle, \mathcal{F}),$$

3. and the alternating fixpoint model $\mathcal{M}^a_\mathcal{D}$ as

$$\mathcal{M}^a_\mathcal{D} := \texttt{lfp}\ (\widehat{S}^2_\mathcal{D}, \emptyset) \cup \neg\, \overline{\widehat{S}^2_\mathcal{D}(\texttt{lfp}\ (\widehat{S}^2_\mathcal{D}, \emptyset))},$$

where $\widehat{S}^2_\mathcal{D}$ denotes the nested application of the eventual consequence transformation, i.e., $\widehat{S}^2_\mathcal{D}(I^-) = \widehat{S}_\mathcal{D}(\widehat{S}_\mathcal{D}(I^-))$.

In [12] it has been shown that the alternating fixpoint model $\mathcal{M}^a_\mathcal{D}$ coincides with the well-founded model of a given database $\mathcal{D}$. The induced fixpoint computation may indeed serve as a universal model generator for arbitrary classes of deductive rules. However, the eventual consequence operator is computationally expensive due to the intermediate determination of supersets of sets of true atoms. With respect to the discussed transformation-based approaches, the iterated fixpoint model could be used for determining the semantics of the stratifiable subset of rules in $\mathcal{R}_{ms}$ for query optimization, $\mathcal{R}^\Delta_{mu}$ for update propagation, and $\mathcal{R}^\Delta_{mu} \cup \mathcal{R}^\nabla_\tau$ for view updating. If these rule sets contain unstratifiable rules, the soft or alternating fixpoint model generator ought be used while the first has proven to be more efficient than the latter [2]. None of the above mentioned consequence operators, however, can deal with indefinite rules necessary for evaluating the view updating rules $\mathcal{R} \cup \mathcal{R}^\nabla$.

4.2 Indefinite Rules

In [4], the author proposed a consequence operator for the efficient bottom-up state generation of stratifiable disjunctive deductive databases. To this end, a new version of the immediate consequence operator based on hyperresolution has been introduced which extends Minker's operator for positive disjunctive Datalog rules [15]. In contrast to already existing model generation methods our approach for efficiently computing perfect models is based on state generation. Within this disjunctive consequence operator, the mapping `red` on indefinite facts is employed which returns non-redundant and subsumption-free representations of disjunctive facts. Additionally, the mapping `min_models`(F) is used for determining the set of minimal Herbrand models from a given set of disjunctive facts F. We identify a disjunctive fact with a set of atoms such that the occurrence of a ground atom A within a fact f can also be written as $A \in f$. The set difference operator can then be used to remove certain atoms from a disjunction while the empty set as result is interpreted as $false$.

Definition 4. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a stratifiable disjunctive database rules,λ a stratification on $\mathcal{D}$, $\mathcal{R}_1 \uplus \ldots \uplus \mathcal{R}_n$ the partition of $\mathcal{R}$ induced by λ, I an arbitrary subset of indefinite facts from the disjunctive Herbrand base [11] of $\mathcal{D}$, and $[[\mathcal{R}]]_I$ the set of all ground instances of rules in $\mathcal{R}$ with respect to the set I Then we define.*

1. *the disjunctive consequence operator $T^{state}_{\mathcal{R}}$ as*

$$\begin{aligned}
T^{state}_{\mathcal{R}}(I) := \mathtt{red}(\{H \mid H \in I \vee \exists r \in [[\mathcal{R}]]_I : r \equiv A_1 \vee \ldots \vee A_l \leftarrow L_1 \wedge \ldots \wedge L_n \\
\textit{with } H = (A_1 \vee \cdots \vee A_l \vee f_1 \setminus L_1 \vee \cdots \vee f_n \setminus L_n \vee C) \\
\textit{such that } f_i \in I \wedge L_i \in f_i \textit{ for all positive literals } L_i \\
\textit{and } L_j \notin I \textit{ for all negative literals } L_j \equiv \neg L \\
\textit{and } (L_j \in C \Leftrightarrow \exists \mathcal{M} \in \mathtt{min\_models}(I) : \\
L_j \in \mathcal{M} \textit{ for at least one negative literal } L_j \\
\textit{and } L_k \in \mathcal{M} \textit{ for all positive literals } L_k \\
\textit{and } A_l \notin \mathcal{M} \textit{ for all head literals of } r)\})
\end{aligned}$$

2. *the iterated fixpoint state S_n as the last minimal model state of the sequence*

$$S_1 := \mathtt{lfp}(T^{state}_{\mathcal{R}_1}, \mathcal{F}),\ sLS_2 := \mathtt{lfp}(T^{state}_{\mathcal{R}_2}, S_1), \ldots, S_n := \mathtt{lfp}(T^{state}_{\mathcal{R}_n}, S_{n-1}),$$

3. *and the iterated fixpoint state model $\mathcal{MS}_{\mathcal{D}}$ as*
   $\mathcal{MS}_{\mathcal{D}} := S_n \uplus \neg \cdot \overline{S_n}$.

In [4] it has been shown that the iterated fixpoint state model $\mathcal{MS}_{\mathcal{D}}$ of a disjunctive database $\mathcal{D}$ coincides with the perfect model state of $\mathcal{D}$. It induces a constructive method for determining the semantics of stratifiable disjunctive databases. The only remaining question is how integrity constraints are handled in the context of disjunctive databases. We consider again definite facts as integrity constraints, only, which must be derivable in every model of the disjunctive database. Thus, only those models from the iterated fixpoint state are selected in which the respective definite facts are derivable. To this end, the already introduced keyword *false* can be used for indicating and removing inconsistent model states. The database is called consistent iff at least one consistent model state exists.

This proposed inference method is well-suited for determining the semantics of stratifiable disjunctive databases with integrity constraints. And thus, it seems to be suited as the basic inference mechanism for evaluating view updating rules. The problem is, however, that the respective rules contain unstratifiable definite rules which cannot be evaluated using the inference method proposed above. Hence, the evaluation techniques for definite (Sect. 4.1) and indefinite rules (Sect. 4.2) do not really fit together and a new uniform approach is needed.

5 A Uniform Fixpoint Approach

In this section, a new version of the soft consequence operator is proposed which is suited as efficient state generator for softly stratifiable definite as well as stratifiable indefinite databases. The original version of the soft consequence operator $T^s_{\mathcal{P}}$ is based on the immediate consequence operator by van Emden and Kowalski and can be applied to an arbitrary partition $\mathcal{P}$ of a given set of definite rules. Consequently, its application does not always lead to correct derivations. In fact, this operator has been designed for the application to softly stratified rules resulting from the application of Magic Sets. However, this operator is also suited for determining the perfect model of a stratifiable database.

Lemma 1. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a stratifiable database and λ a stratification of $\mathcal{R}$ inducing the partition $\mathcal{P}$ of $\mathcal{R}$. The perfect model $\mathcal{M}_{\mathcal{D}}$ of $\langle \mathcal{F}, \mathcal{R} \rangle$ is identical with the soft fixpoint model of $\mathcal{D}$, i.e.,*

$$\mathcal{M}_{\mathcal{D}} = \mathtt{lfp}(T^s_{\mathcal{P}}, \mathcal{F}) \uplus \neg \cdot \overline{\mathtt{lfp}(T^s_{\mathcal{P}}, \mathcal{F})}.$$

Proof. This property follows from the fact that for every partition $\mathcal{P} = P_1 \uplus \dots P_n$ induced by a stratification, the condition $\mathtt{pred}(P_i) \cap \mathtt{pred}(P_j) = \emptyset$ with $i \neq j$ must necessarily hold. As soon as the application of the immediate consequence operator T_{P_i} with respect to a certain layer P_i generates no new facts anymore, the rules in P_i can never fire again. The application of the incorporated `min` function then induces the same sequence of Herbrand models as in the case of the iterated fixpoint computation. □

Another property we need for extending the original soft consequence operator is about the application of T^{state} to definite rules and facts.

Lemma 2. *Let r be an arbitrary definite rule and f be a set of arbitrary definite facts. The single application of r to f using the immediate consequence operator or the disjunctive consequence operator, always yields the same result, i.e.,*

$$T_r(f) \; = \; T^{state}_r(f).$$

Proof. The proof follows from the fact that all non-minimal conclusions of T^{state} are immediately eliminated by the subsumption operator `red`. □

The above proposition establishes the relationship between the definite and indefinite case showing that the disjunctive consequence operator represents a generalization of the immediate one. Thus, its application to definite rules and facts can be used to realize the same derivation process as the one performed by using the immediate consequence operator. Based on the two properties from above, we can now consistently extend the definition of the soft consequence operator which allows its application to indefinite rules and facts, too.

Definition 5. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be an arbitrary disjunctive database, I an arbitrary subset of indefinite facts from the disjunctive Herbrand base of $\mathcal{D}$, and $\mathcal{P} = P_1 \uplus \ldots \uplus P_n$ a partition of $\mathcal{R}$. The general soft consequence operator $T^g_{\mathcal{P}}(I)$ is defined as*

$$T^g_{\mathcal{P}}(I) := \begin{cases} I & \text{if } T_{P_j}(I) = I \text{ for all } j \in \{1,\ldots,n\} \\ T^{state}_{P_i}(I) & \text{with } i = \mathtt{min}\{j \mid T^{state}_{P_j}(I) \supsetneq I\}, \quad \text{otherwise.} \end{cases}$$

where $T^{state}_{P_i}$ denotes the disjunctive consequence operator.

In contrast to the original definition, the general soft consequence operator is based on the disjunctive operator $T^{state}_{P_i}$ instead of the immediate consequence operator. The least fixpoint of $T^g_{\mathcal{P}}$ can be used to determine the perfect model of definite as well as indefinite stratifiable databases and the well-founded model of softly stratifiable definite databases.

Theorem 1. *Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ be a stratifiable disjunctive database and λ a stratification of $\mathcal{R}$ inducing the partition $\mathcal{P}$ of $\mathcal{R}$. The perfect model state $\mathcal{PS}_{\mathcal{D}}$ of $\langle \mathcal{F}, \mathcal{R} \rangle$ is identical with the least fixpoint model of $T^g_{\mathcal{P}}$, i.e.,*

$$\mathcal{PS}_{\mathcal{D}} = \mathtt{lfp}(T^g_{\mathcal{P}}, \mathcal{F}) \uplus \neg \cdot \overline{\mathtt{lfp}(T^g_{\mathcal{P}}, \mathcal{F})}.$$

Proof. The proof directly follows from the correctness of the fixpoint computations for each stratum as shown in [4] and the same structural argument already used in Lemma 1. □

The definition of $\mathtt{lfp}(T^g_{\mathcal{P}}, \mathcal{F})$ induces a constructive method for determining the perfect model state as well as the well-founded model of a given database. Thus, it forms a suitable basis for the evaluation of the rules $\mathcal{R}_{ms}$ for query optimization, $\mathcal{R}^{\Delta}_{mu}$ for update propagation, and $\mathcal{R}^{\Delta}_{mu} \uplus \mathcal{R}^{\nabla}_{\tau}$ as well as $\mathcal{R} \uplus \mathcal{R}^{\nabla}$ for view updating. This general approach to defining the semantics of different classes of deductive rules is surprisingly simple and induces a rather efficient inference mechanism in contrast to general well-founded model generators. The soft stratification concept, however, is not yet applicable to indefinite databases because ordinary Magic Sets can not be used for indefinite clauses. Nevertheless, the resulting extended version of the soft consequence operator can be used as a uniform basis for the evaluation of all transformation-based techniques mentioned in this paper.

6 Conclusion

In this paper, we have presented an extended version of the soft consequence operator for the efficient top-down and bottom-up reasoning in deductive databases. This operator allows for the efficient evaluation of softly stratifiable incremental expressions and stratifiable disjunctive rules. It solely represents a theoretical approach but provides insights into design decisions for extending the inference component of commercial database systems. The relevance and quality of the transformation-based approaches, however, has been already shown in various practical research projects (e.g. [5,8]) at the University of Bonn.

References

1. Bancilhon, F., Ramakrishnan, R.: An Amateur's introduction to recursive query processing strategies. In: SIGMOD Conference 1986, pp. 16–52 (1986)
2. Behrend, A.: Soft stratification for magic set based query evaluation in deductive databases. In: PODS 2003, New York, pp. 102–110, 9–12 June 2003
3. Behrend, A.: Optimizing exitstential queries in stratifiable deductive databases. In: SAC 2005, pp. 623–628 (2005)
4. Behrend, A.: A fixpoint approach to state generation for stratifiable disjunctive deductive databases. In: Ioannidis, Y., Novikov, B., Rachev, B. (eds.) ADBIS 2007. LNCS, vol. 4690, pp. 283–297. Springer, Heidelberg (2007)
5. Behrend, A., Dorau, C., Manthey, R., Schüller, G.: Incremental view-based analysis of stock market data streams. In: IDEAS 2008, pp. 269–275. ACM, New York (2008)
6. Behrend, A., Manthey, R.: Update propagation in deductive databases using soft stratification. In: Benczúr, A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 22–36. Springer, Heidelberg (2004)
7. Behrend, A., Manthey, R.: A transformation-based approach to view updating in stratifiable deductive databases. In: Hartmann, S., Kern-Isberner, G. (eds.) FoIKS 2008. LNCS, vol. 4932, pp. 253–271. Springer, Heidelberg (2008)
8. Behrend, A., Schüller, G., Manthey, R.: AIMS: an sql-based system for airspace monitoring. In: IWGS 2010, pp. 31–38. ACM, New York (2010)
9. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Logic Program. **10**(1/2/3 & 4), 255–299 (1991)
10. Bry, F.: Logic programming as constructivism: a formalization and its application to databases. In: PODS 1989, pp. 34–50 (1989)
11. Fernandez, J.A., Minker, J.: Semantics of disjunctive deductive databases. In: Hull, R., Biskup, J. (eds.) ICDT 1992. LNCS, vol. 646, pp. 21–50. Springer, Heidelberg (1992)
12. Kemp, D., Srivastava, D., Stuckey, P.: Bottom-up evaluation and query optimization of well-founded models. Theoret. Comput. Sci. **146**(1 & 2), 145–184 (1995)
13. Kuchenhoff, V.: On the efficient computation of the difference between consecutive database states. In: Delobel, C., Masunaga, Y., Kifer, M. (eds.) DOOD 1991. LNCS, vol. 566, pp. 478–502. Springer, Heidelberg (1991)
14. Manthey, R.: Reflections on some fundamental issues of rule-based incremental update propagation. In: DAISD 1994, pp. 255–276, Universitat Politècnica de Catalunya, 19–21 September 1994

15. Minker, J.: On indefinite databases and the closed world assumption. In: Loveland, D.W. (ed.) CADE 1982. LNCS, vol. 138, pp. 292–308. Springer, Heidelberg (1982)
16. Olivé, A.: Integrity constraints checking in deductive databases. In: VLDB 1991, pp. 513–523 (1991)
17. Przymusinski, T.C.: On the declarative semantics of deductive databases and logic programs. In: Foundations of Deductive Databases and Logic Programming, pp. 193–216. Morgan Kaufmann, Los Altos (1988)
18. Ramakrishnan, R.: Magic templates: a spellbinding approach to logic programs. J. Logic Program. **11**(3&4), 189–216 (1991)
19. Rohmer, J., Lescoeur, R., Kerisit, J.-M.: The Alexander method - a technique for the processing of recursive axioms in deductive databases. New Gener. Comput. **4**(3), 273–285 (1986)
20. Van Gelder, A.: The alternating fixpoint of logic programs with negation. J. Comput. Syst. Sci. **47**(1), 185–221 (1993)
21. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38**(3), 620–650 (1991)

INAP Technical Papers II: Answer-Set Programming and Abductive Reasoning

Translating Answer-Set Programs into Bit-Vector Logic

Mai Nguyen, Tomi Janhunen(✉), and Ilkka Niemelä

Department of Information and Computer Science,
Aalto University School of Science, Espoo, Finland
{Tomi.Janhunen, Ilkka.Niemela}@aalto.fi

Abstract. Answer set programming (ASP) is a paradigm for declarative problem solving where problems are first formalized as rule sets, i.e., answer-set programs, in a uniform way and then solved by computing answer sets for programs. The satisfiability modulo theories (SMT) framework follows a similar modelling philosophy but the syntax is based on extensions of propositional logic rather than rules. Quite recently, a translation from answer-set programs into difference logic was provided—enabling the use of particular SMT solvers for the computation of answer sets. In this paper, the translation is revised for another SMT fragment, namely that based on fixed-width bit-vector theories. Consequently, even further SMT solvers can be harnessed for the task of computing answer sets. The results of a preliminary experimental comparison are also reported. They suggest a level of performance which is similar to that achieved via difference logic.

1 Introduction

Answer set programming (ASP) is a rule-based approach to declarative problem solving [5,13,21,23]. The idea is to first formalize a given problem as a set of rules, also called an *answer-set program*, so that the answer sets of the program correspond to the solutions of the problem. Such problem descriptions are typically devised in a *uniform* way which distinguishes general principles and constraints of the problem in question from any instance-specific data. To this end, term variables are deployed for compact representation of rules. Solutions themselves can then be found out by *grounding* the rules of the answer-set program, and by computing answer sets for the resulting ground program using an answer set solver. State-of-the-art answer set solvers are already very efficient search engines [7,10] and have a wide range of industrial applications.

The satisfiability modulo theories (SMT) framework [3] follows a similar modelling philosophy but the syntax is based on extensions of propositional logic rather than rules with term variables. The SMT framework enriches traditional satisfiability (SAT) checking [4] in terms of background theories which are selected amongst a number of alternatives.[1] Parallel to propositional atoms, also

[1] http://combination.cs.uiowa.edu/smtlib/

theory atoms involving non-Boolean variables[2] can be used to model potentially infinite domains. Theory atoms are typically used to express various constraints such as linear constraints, difference constraints, etc., and they enable very concise representations of certain problem domains for which plain Boolean logic would be more verbose or insufficient in the first place.

As regards the relationship of ASP and SMT, it was quite recently shown [18,24] that answer-set programs can be efficiently translated into a simple SMT fragment, namely *difference logic* (DL) [25]. This fragment is based on theory atoms of the form $x - y \leq k$ formalizing an upper bound k on the *difference* of two integer-domain variables x and y. Although the required transformation is linear, it cannot be expected that such theories are directly written by humans in order to express the essentials of ASP in SMT. The translations from [18,24] and their implementation called LP2DIFF[3] enable the use of particular SMT solvers for the computation of answer sets. Our experimental results [18] indicate that the performance obtained in this way is surprisingly close to that of state-of-the-art answer set solvers. The results of the third ASP competition [7], however, suggest that the performance gap has grown since the previous competition. To address this trend, our current and future agendas include a number of points:

– To gradually increase the number of supported SMT fragments which enables the use of further SMT solvers for the task of computing answer sets.
– To continue the development of new translation techniques from ASP to SMT.
– To submit ASP-based benchmark sets to future SMT competitions (SMT-COMPs) to foster the efficiency of SMT solvers on problems that are relevant for ASP.
– To develop new integrated languages that combine features of ASP and SMT, and aim at implementations via translation into pure SMT as initiated in [16].

This paper contributes to the first item by devising a translation from answer-set programs into theories of bit-vector logic. There is a great interest to develop efficient solvers for this particular SMT fragment due to its industrial relevance. In view of the second item, we generalize an existing translation from [18] to the case of bit-vector logic. Using an implementation of the new translation, viz. LP2BV, new benchmark classes can be created to support the third item on our agenda. Finally, the translation also creates new potential for language integration. In the long run, rule-based languages and, in particular, the modern grounders exploited in ASP can provide valuable machinery for the generation of SMT theories in analogy to answer-set programs—meaning that the *source code* of an SMT theory can be compacted using rules and term variables [16] and specified in a uniform way which is independent of any concrete problem instances. Analogous approaches [2,12,22] combine ASP and constraint programming techniques without performing a translation into a single formalism.

The rest of this paper is organized as follows. First, the basic definitions and concepts of answer-set programs and fixed-width bit-vector logic are briefly

[2] However, variables in SMT are syntactically represented by (functional) constants having a free interpretation over a specific domain such as integers or reals.
[3] http://www.tcs.hut.fi/Software/lp2diff/

reviewed in Sect. 2. The new translation from answer-set programs into bit-vector theories is then devised in Sect. 3. The extended rule types of SMODELS-compatible systems are addressed in Sect. 4. Such extensions can be covered either by native translations into bit-vector logic or translations into normal programs. As part of this research, we carried out a number of experiments using benchmarks from the second ASP competition [10] and two state-of-the-art SMT solvers, viz. BOOLECTOR and Z3. The results of the experiments are reported in Sect. 5. Finally, we conclude this paper in Sect. 6.

2 Preliminaries

The goal of this section is to briefly review the source and target formalisms for the new translation devised in the sequel. First, in Sect. 2.1, we recall normal logic programs subject to answer-set semantics and the main concepts exploited in their translation. A formal account of bit-vector logic follows in Sect. 2.2.

2.1 Normal Logic Programs

As usual, we define a *normal logic program* P as a finite set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \tag{1}$$

where a, $b_1, \ldots, b_n$, and $c_1, \ldots, c_m$ are propositional atoms and $\sim$ denotes *default negation*. The *head* of a rule r of the form (1) is $\mathrm{hd}(r) = a$ whereas the part after the symbol $\leftarrow$ forms the *body* of r, denoted by $\mathrm{bd}(r)$. The body $\mathrm{bd}(r)$ consists of the positive part $\mathrm{bd}^+(r) = \{b_1, \ldots, b_n\}$ and the negative part $\mathrm{bd}^-(r) = \{c_1, \ldots, c_m\}$ so that $\mathrm{bd}(r) = \mathrm{bd}^+(r) \cup \{\sim c \mid c \in \mathrm{bd}^-(r)\}$. Intuitively, a rule r of the form (1) appearing in a program P is used as follows: the head $\mathrm{hd}(r)$ can be inferred by r if the *positive body atoms* in $\mathrm{bd}^+(r)$ are inferable by the other rules of P, but not the *negative body atoms* in $\mathrm{bd}^-(r)$. The positive part of the rule, r^+ is defined as $\mathrm{hd}(r) \leftarrow \mathrm{bd}^+(r)$. A normal logic program is called *positive* if $r = r^+$ holds for every rule $r \in P$.

Semantics. To define the semantics of a normal program P, we let $\mathrm{At}(P)$ stand for the set of atoms that appear in P. An *interpretation* of P is any subset $I \subseteq \mathrm{At}(P)$ such that for an atom $a \in \mathrm{At}(P)$, a is *true* in I, denoted $I \models a$, iff $a \in I$. For any negative literal $\sim c$, $I \not\models \sim c$ iff $I \not\models c$ iff $c \notin I$. A rule r is satisfied in I, denoted $I \models r$, iff $I \models \mathrm{bd}(r)$ implies $I \models \mathrm{hd}(r)$. An interpretation I is a *classical model* of P, denoted $I \models P$, iff, $I \models r$ holds for every $r \in P$. A model $M \models P$ is a *minimal model* of P iff there is no $M' \models P$ such that $M' \subset M$. Each positive normal program P has a unique minimal model, i.e., the *least model* of P denoted by $\mathrm{LM}(P)$ in the sequel. The least model semantics can be extended for an arbitrary normal program P by *reducing* P into a positive program $P^M = \{r^+ \mid r \in P \mathrm{and} M \cap \mathrm{bd}^-(r) = \emptyset\}$ with respect to $M \subseteq \mathrm{At}(P)$. Then *answer sets*, originally known as *stable models* [14], can be defined.

Definition 1 (Gelfond and Lifschitz *[14]***).** *An interpretation* $M \subseteq \mathrm{At}(P)$ *is an* answer set *of a normal program* P *iff* $M = \mathrm{LM}(P^M)$.

Generally speaking, the number of answer sets associated with a normal program P can vary. Thus, we write $\mathrm{AS}(P)$ for the set of answer sets possessed by the program P.

Example 1. Consider a normal program P [18] consisting of the following six rules:

$$\begin{array}{lll} a \leftarrow b, c. & a \leftarrow d. & b \leftarrow a, \sim d. \\ b \leftarrow a, \sim c. & c \leftarrow \sim d. & d \leftarrow \sim c. \end{array}$$

The answer sets of P are $M_1 = \{a, b, d\}$ and $M_2 = \{c\}$. The reduct P^{M_1} associated with the first answer set is $\{a \leftarrow b, c;\; a \leftarrow d;\; b \leftarrow a;\; d \leftarrow\}$. Thus, we obtain $\mathrm{LM}(P^{M_1}) = \{a, b, d\} = M_1$. To verify the latter, we note that the reduct $P^{M_2} = \{a \leftarrow b, c;\; b \leftarrow a;\; c \leftarrow;\; a \leftarrow d\}$ for which $\mathrm{LM}(P^{M_2}) = \{c\}$. On the other hand, we have $P^{M_3} = P^{M_2}$ for $M_3 = \{a, b, c\}$ witnessing $M_3 \notin \mathrm{AS}(P)$. ■

In what follows, we present some concepts and results that are relevant to capture answer sets in terms of propositional logic and its extensions in the SMT framework.

***Completion*.** Given a normal program P and an atom $a \in \mathrm{At}(P)$, the *definition* of a in P is the set of rules $\mathrm{Def}_P(a) = \{r \in P \mid \mathrm{hd}(r) = a\}$. The *completion* of a normal program P, denoted by $\mathrm{Comp}(P)$, is a propositional theory [8] which contains

$$a \leftrightarrow \bigvee_{r \in \mathrm{Def}_P(a)} \Big(\bigwedge_{b \in \mathrm{bd}^+(r)} b \;\wedge \bigwedge_{c \in \mathrm{bd}^-(r)} \neg c \Big) \tag{2}$$

for each atom $a \in \mathrm{At}(P)$. Given a propositional theory T and its signature $\mathrm{At}(T)$, the semantics of T is determined by $\mathrm{CM}(T) = \{M \subseteq \mathrm{At}(T) \mid M \models T\}$. It is possible to relate $\mathrm{CM}(\mathrm{Comp}(P))$ with the models of a normal program P by distinguishing *supported models* [1] for P. A model $M \models P$ is a supported model of P iff for every atom $a \in M$ there is a rule $r \in P$ such that $\mathrm{hd}(r) = a$ and $M \models \mathrm{bd}(r)$. In general, the set of supported models $\mathrm{SuppM}(P)$ of a normal program P coincides with $\mathrm{CM}(\mathrm{Comp}(P))$. It can be shown [20] that stable models are also supported models but not necessarily vice versa. This means that in order to capture $\mathrm{AS}(P)$ using $\mathrm{Comp}(P)$, the latter has to be extended in terms of additional constraints as done, e.g., in [15,18].

Example 2. For the program P of Example 1, the theory $\mathrm{Comp}(P)$ consists of formulas

$$\begin{array}{l} a \leftrightarrow (b \wedge c) \vee d, \\ b \leftrightarrow (a \wedge \neg d) \vee (a \wedge \neg c), \\ c \leftrightarrow \neg d, \text{and} \\ d \leftrightarrow \neg c. \end{array}$$

The models of $\mathrm{Comp}(P)$, i.e., the supported models of P, are $M_1 = \{a, b, d\}$, $M_2 = \{c\}$, and $M_3 = \{a, b, c\}$. Recall that M_3 is not stable as shown in Example 1. ■

Dependency Graphs. The *positive dependency graph* of a normal program P, denoted by $\mathrm{DG}^+(P)$, is a pair $\langle \mathrm{At}(P), \leq \rangle$ where $b \leq a$ holds iff there is a rule $r \in P$ such that $\mathrm{hd}(r) = a$ and $b \in \mathrm{bd}^+(r)$. Let $\leq^*$ denote the *reflexive* and *transitive* closure of $\leq$. A *strongly connected component* (SCC) of $\mathrm{DG}^+(P)$ is a maximal non-empty subset $S \subseteq \mathrm{At}(P)$ such that $a \leq^* b$ and $b \leq^* a$ hold for each $a, b \in S$. The set of defining rules is generalized for an SCC S by $\mathrm{Def}_P(S) = \bigcup_{a \in S} \mathrm{Def}_P(a)$. This set can be partitioned into the respective sets of *externally* and *internally* supporting rules:

$$\mathrm{Ext}_P(S) = \{r \in \mathrm{Def}_P(S) \mid \mathrm{bd}^+(r) \cap S = \emptyset\} \tag{3}$$

$$\mathrm{Int}_P(S) = \{r \in \mathrm{Def}_P(S) \mid \mathrm{bd}^+(r) \cap S \neq \emptyset\} \tag{4}$$

It is clear by (3) and (4) that $\mathrm{Def}_P(S) = \mathrm{Ext}_P(S) \sqcup \mathrm{Int}_P(S)$ holds in general.

Example 3. In the case of the program P from Example 1, the SCCs of $\mathrm{DG}^+(P)$ are $S_1 = \{a, b\}$, $S_2 = \{c\}$, and $S_3 = \{d\}$. For S_1, we have $\mathrm{Ext}_P(S_1) = \{a \leftarrow d\}$ while $\mathrm{Int}_P(S_1)$ consists of the remaining rules $a \leftarrow b, c$; $b \leftarrow a, \sim c$; and $b \leftarrow a, \sim d$ from the definitions of a and b. But for the singleton SCCs, $\mathrm{Int}_P(S_2) = \mathrm{Int}_P(S_3) = \emptyset$. ■

The strongly connected components of $\mathrm{DG}^+(P)$ are essential when it comes to the modularization of answer-set programs. Definition 1 concerns an entire normal program and is therefore global by nature. The results from [26] allow for localizing the stability condition using the strongly connected components $S_1, \ldots, S_n$ of $\mathrm{DG}^+(P)$. These components partition P into $\mathrm{Def}_p(S_1) \sqcup \ldots \sqcup \mathrm{Def}_p(S_n)$. Without going into all details, it is possible to define answer sets for each part $\mathrm{Def}_P(S_i)$ where $1 \leq i \leq n$ separately so that $\mathrm{AS}(P) = AS(\mathrm{Def}_P(S_1)) \bowtie \ldots \bowtie AS(\mathrm{Def}_P(S_n))$. Here the operator $\bowtie$ carries out the *natural join* of the answer sets, i.e., if $M_1, \ldots, M_n$ are *compatible*[4] interpretations for $\mathrm{Def}_P(S_1), \ldots, \mathrm{Def}_P(S_n)$, then it holds for $M = M_1 \cup \ldots \cup M_n$, that $M \in \mathrm{AS}(P)$ iff $M_1 \in \mathrm{AS}(\mathrm{Def}_P(S_1)), \ldots, M_n \in \mathrm{AS}(Def_P(S_n))$. This result is known as the *module theorem* [26] and, in a sense, it captures the limits how far the idea of a *modular translation* can be pushed in ASP. For certain purposes rule-level translations are insufficient and entire modules have to be considered (see [15,17]). This is also the case for this paper and the translation into bit-vector logic will take SCCs and the respective modules into account. This also reduces the length of the translation.

2.2 Bit-Vector Logic

Fixed-width bit-vector theories have been introduced for high-level reasoning about digital circuitry and computer programs in the SMT framework [3]. Such theories are expressed in an extension of propositional logic where atomic formulas speak about bit vectors in terms of a rich variety of operators. A particular file format[5] is used.

[4] Compatible interpretations assign the same truth values to their joint atoms.
[5] http://goedel.cs.uiowa.edu/smtlib/papers/pdpar-proposal.pdf

Syntax. As usual in the context of SMT, variables are realized as *free constants* that are interpreted over a particular domain (such as integers or reals).[6] In the case of fixed-width bit-vector theories, this means that each constant symbol x represents a vector $x[1 \dots m]$ of bits of particular width m, denoted by $\mathrm{w}(x)$ in the sequel. Such vectors enable a more compact representation of structures like registers and often allow more efficient reasoning about them. A special notation $\overline{n}$ is introduced to denote a bit vector that equals to n, i.e., $\overline{n}$ provides a binary representation of n. We assume that the actual width $m \geq \log_2(n+1)$ is determined by the context where the notation $\overline{n}$ is used.

For the purposes of this paper, the most interesting arithmetic operator for combining bit vectors is the addition of two m-bit vectors, denoted by the parameterized function symbol $+_m$ in an infix notation. The resulting vector is also m-bit which can lead to an overflow if the sum exceeds $2^m - 1$. We use Boolean operators $=_m$ and $<_m$ with the usual meanings for comparing the (unsigned) values of two m-bit vectors. Thus, assuming that x and y are m-bit free constants, we may write atomic formulas like $x =_m y$ and $x <_m y$ in order to compare the m-bit values of x and y. In addition to syntactic elements mentioned so far, we can use the primitives of propositional logic to build more complex *well-formed formulas* of bit-vector logic. The syntax defined for the SMT library contains further primitives which are skipped in this paper. A theory T in bit-vector logic is a set of well-formed bit-vector formulas as illustrated next.

Example 4. Consider a system of two processes, say A and B, and a theory $T = \{a \rightarrow (x <_2 y),\ b \rightarrow (y <_2 x)\}$ formalizing a scheduling policy for them. The intuitive reading of a (resp. b) is that process A (resp. B) is scheduled with a higher priority and, thus, should start earlier. The constants x and y denote the respective starting times of A and B. Thus, e.g., $x <_2 y$ means that process A starts before process B. ■

Semantics. Given a bit-vector theory T, we write $\mathrm{At}(T)$ and $\mathrm{FC}(T)$ for the sets of propositional atoms and free constants, respectively, appearing in T. To determine the semantics of T, we define *interpretations* for T as pairs $\langle I, \tau \rangle$ where $I \subseteq \mathrm{At}(T)$ is a standard propositional interpretation and τ is a partial function that maps a free constant $x \in \mathrm{FC}(T)$ and an index $1 \leq i \leq \mathrm{w}(x)$ to the set of bits $\{0, 1\}$. Given τ, a free constant $x \in \mathrm{FC}(T)$ is mapped onto $\tau(x) = \sum_{i=1}^{\mathrm{w}(x)} (\tau(x, i) \cdot 2^{\mathrm{w}(x)-i})$ and, in particular, $\tau(\overline{n}) = n$ for any n. To cover any *well-formed terms* [7] t_1 and t_2 involving $+_m$ and m-bit free constants from $\mathrm{FC}(T)$, we define $\tau(t_1 +_m t_2) = \tau(t_1) + \tau(t_2) \mod 2^m$ and $\mathrm{w}(t_1 +_m t_2) = m$. Hence, the value $\tau(t)$ can be determined for any well-formed term t which enables the evaluation of more complex formulas as formalized below.

[6] We use typically symbols x, y, z to denote such free (functional) constants and symbols a, b, c to denote propositional atoms.

[7] The constants and operators appearing in a well-formed term t are based on a fixed width m. Moreover, the width $\mathrm{w}(x)$ of each free constant $x \in \mathrm{FC}(T)$ must be the same throughout T.

Definition 2. *Let T be a bit-vector theory, $a \in \mathrm{At}(T)$ a propositional atom, t_1 and t_2 well-formed terms over $\mathrm{FC}(T)$ such that $\mathrm{w}(t_1) = \mathrm{w}(t_2)$, and ϕ and ψ well-formed formulas. Given an interpretation $\langle I, \tau \rangle$ for the theory T, we define*

1. $\langle I, \tau \rangle \models a \iff a \in I$,
2. $\langle I, \tau \rangle \models t_1 =_m t_2 \iff \tau(t_1) = \tau(t_2)$,
3. $\langle I, \tau \rangle \models t_1 <_m t_2 \iff \tau(t_1) < \tau(t_2)$,
4. $\langle I, \tau \rangle \models \neg\phi \iff \langle I, \tau \rangle \not\models \phi$,
5. $\langle I, \tau \rangle \models \phi \vee \psi \iff \langle I, \tau \rangle \models \phi$ *or* $\langle I, \tau \rangle \models \psi$,
6. $\langle I, \tau \rangle \models \phi \rightarrow \psi \iff \langle I, \tau \rangle \not\models \phi$ *or* $\langle I, \tau \rangle \models \psi$, *and*
7. $\langle I, \tau \rangle \models \phi \leftrightarrow \psi \iff \langle I, \tau \rangle \models \phi$ *if and only if* $\langle I, \tau \rangle \models \psi$.

The interpretation $\langle I, \tau \rangle$ is a model of T, i.e., $\langle I, \tau \rangle \models T$, iff $\langle I, \tau \rangle \models \phi$ for all $\phi \in T$.

It is clear by Definition 2 that pure propositional theories T are treated classically, i.e., $\langle I, \tau \rangle \models T$ iff $I \models T$ in the sense of propositional logic.

Example 5. Recalling the theory $T = \{a \rightarrow (x <_2 y),\ b \rightarrow (y <_2 x)\}$ from Example 4, we have the sets of symbols $\mathrm{At}(T) = \{a, b\}$ and $\mathrm{FC}(T) = \{x, y\}$. Furthermore, we observe that there is no model of T of the form $\langle \{a, b\}, \tau \rangle$ because it is impossible to satisfy $x <_2 y$ and $y <_2 x$ simultaneously using any partial function τ. On the other hand, there are 6 models of the form $\langle \{a\}, \tau \rangle$ because $x <_2 y$ can be satisfied in $3 + 2 + 1 = 6$ ways by picking different values for the 2-bit vectors x and y. ■

3 Translation

In this section, we present a translation of a logic program P into a bit-vector theory $\mathrm{BV}(P)$ that is similar to an existing translation [18] into difference logic. As its predecessor, the translation $\mathrm{BV}(P)$ consists of two parts. Clark's completion [8], denoted by $\mathrm{CC}(P)$, forms the first part of $\mathrm{BV}(P)$. The second part, i.e., $\mathrm{R}(P)$, is based on *ranking constraints* from [24] so that $\mathrm{BV}(P) = \mathrm{CC}(P) \cup \mathrm{R}(P)$. Intuitively, the idea is that the completion $\mathrm{CC}(P)$ captures *supported models* of P [1] and the further formulas in $\mathrm{R}(P)$ exclude the non-stable ones so that any classical model of $\mathrm{BV}(P)$ corresponds to a stable model of the input program P. The completion $\mathrm{CC}(P)$ is formed for each atom $a \in \mathrm{At}(P)$ on the basis of (2) but by taking into account a number of special cases:

1. If $\mathrm{Def}_P(a) = \emptyset$, the formula $\neg a$ captures the resulting empty disjunction in (2).
2. If there is $r \in \mathrm{Def}_P(a)$ such that $\mathrm{bd}(r) = \emptyset$, then one of the disjuncts in (2) is trivially true and the formula a can be used as such to capture the definition of a.

3. If $\mathrm{Def}_P(a) = \{r\}$ for a single rule $r \in P$ with $n + m > 0$, then we simplify (2) to a formula of the form

$$a \leftrightarrow \bigwedge_{b \in \mathrm{bd}^+(r)} b \quad \wedge \bigwedge_{c \in \mathrm{bd}^-(r)} \neg c. \tag{5}$$

4. Otherwise, the set $\mathrm{Def}_P(a)$ contains at least two rules (1) with $n + m > 0$ and

$$a \leftrightarrow \bigvee_{r \in \mathrm{Def}_P(a)} \mathrm{bd}_r \tag{6}$$

is introduced using a new atom bd_r for each $r \in \mathrm{Def}_P(a)$ together with a formula

$$\mathrm{bd}_r \leftrightarrow \bigwedge_{b \in \mathrm{bd}^+(r)} b \quad \wedge \bigwedge_{c \in \mathrm{bd}^-(r)} \neg c. \tag{7}$$

The rest of the translation exploits the SCCs of the positive dependency graph of P that was defined in Sect. 2.1. The motivation is to limit the scope of ranking constraints which favors the length of the resulting translation. In particular, singleton components $\mathrm{SCC}(a) = \{a\}$ require no special treatment if *tautological* rules with $a \in \{b_1, \ldots, b_n\}$ in (1) have been removed as a preprocessing step. Plain completion (2) is sufficient for atoms involved in such components. However, for each atom $a \in \mathrm{At}(P)$ having a non-trivial component $\mathrm{SCC}(a)$ in $\mathrm{DG}^+(P)$ such that $|\mathrm{SCC}(a)| > 1$, two new atoms ext_a and int_a are introduced to formalize the *external* and *internal* support for a, respectively. These atoms are defined in terms of equivalences

$$\mathrm{ext}_a \leftrightarrow \bigvee_{r \in \mathrm{Ext}_P(a)} \mathrm{bd}_r \tag{8}$$

$$\mathrm{int}_a \leftrightarrow \bigvee_{r \in \mathrm{Int}_P(a)} \Big[\mathrm{bd}_r \wedge \bigwedge_{b \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)} (x_b <_m x_a)\Big] \tag{9}$$

where x_a and x_b are bit vectors of width $m = \lceil \log_2(|\mathrm{SCC}(a)|+1) \rceil$ introduced for all atoms involved in $\mathrm{SCC}(a)$. The formulas (8) and (9) are called *weak* ranking constraints and they are accompanied by propositional formulas

$$a \rightarrow \mathrm{ext}_a \vee \mathrm{int}_a, \tag{10}$$

$$\neg \mathrm{ext}_a \vee \neg \mathrm{int}_a. \tag{11}$$

Moreover, when $\mathrm{Ext}_P(a) \neq \emptyset$ and the atom a happens to gain external support from these rules, the value of x_a is fixed to 0 by including the formula

$$\mathrm{ext}_a \rightarrow (x_a =_m \overline{0}). \tag{12}$$

Example 6. Recall the program P from Example 1. The completion $\mathrm{CC}(P)$ is:

$$\begin{array}{lll}
a \leftrightarrow \mathrm{bd}_1 \vee \mathrm{bd}_2. & \mathrm{bd}_1 \leftrightarrow b \wedge c. & \mathrm{bd}_2 \leftrightarrow d. \\
b \leftrightarrow \mathrm{bd}_3 \vee \mathrm{bd}_4. & \mathrm{bd}_3 \leftrightarrow a \wedge \neg d. & \mathrm{bd}_4 \leftrightarrow a \wedge \neg c. \\
c \leftrightarrow \neg d. & & \\
d \leftrightarrow \neg c. & &
\end{array}$$

Since P has only one non-trivial SCC, i.e., the component $\mathrm{SCC}(a) = \mathrm{SCC}(b) = \{a, b\}$, the weak ranking constraints resulting in $\mathrm{R}(P)$ are

$$\begin{array}{ll} \mathrm{ext}_a \leftrightarrow \mathrm{bd}_2. & \mathrm{int}_a \leftrightarrow \mathrm{bd}_1 \wedge (x_b <_2 x_a). \\ \mathrm{ext}_b \leftrightarrow \bot\,. & \\ \multicolumn{2}{l}{\mathrm{int}_b \leftrightarrow [\mathrm{bd}_3 \wedge (x_a <_2 x_b)] \vee [\mathrm{bd}_4 \wedge (x_a <_2 x_b)].} \end{array}$$

In addition to these, the formulas

$$\begin{array}{lll} a \rightarrow \mathrm{ext}_a \vee \mathrm{int}_a. & \neg\mathrm{ext}_a \vee \neg\mathrm{int}_a. & \mathrm{ext}_a \rightarrow (x_a =_2 \overline{0}). \\ b \rightarrow \mathrm{ext}_b \vee \mathrm{int}_b. & \neg\mathrm{ext}_b \vee \neg\mathrm{int}_b. & \end{array}$$

are also included in $\mathrm{R}(P)$. ■

Weak ranking constraints are sufficient whenever the goal is to compute only one answer set, or to check the existence of answer sets. However, they do not guarantee a one-to-one correspondence between the elements of $\mathrm{AS}(P)$ and the set of models obtained for the translation $\mathrm{BV}(P)$. To address this discrepancy, and to potentially make the computation of all answer sets or counting the number of answer sets more effective, *strong* ranking constraints can be imported from [18] as well. Actually, there are two mutually compatible variants of strong ranking constraints:

$$\mathrm{bd}_r \rightarrow \bigvee_{b \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)} \neg(x_b +_m \overline{1} <_m x_a) \tag{13}$$

$$\mathrm{int}_a \rightarrow \bigvee_{r \in \mathrm{Int}_P(a)} [\mathrm{bd}_r \wedge \bigvee_{b \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)} (x_a =_m x_b +_m \overline{1})]. \tag{14}$$

The *local* strong ranking constraint (13) is introduced for each $r \in \mathrm{Int}_P(a)$. It is worth pointing out that the condition $\neg(x_b +_m \overline{1} <_m x_a)$ is equivalent to $x_b +_m \overline{1} \geq_m x_a$.[8]
On the other hand, the *global* variant (14) covers the internal support of a entirely. Finally, in order to prune copies of models of the translation that would correspond to the exactly same answer set of the original program, a formula

$$\neg a \rightarrow (x_a =_m \overline{0}) \tag{15}$$

is included for every atom a involved in a non-trivial SCC. We write $\mathrm{R}^{\mathrm{l}}(P)$ and $\mathrm{R}^{\mathrm{g}}(P)$ for the respective extensions of $\mathrm{R}(P)$ with local/global strong ranking constraints, and $\mathrm{R}^{\mathrm{lg}}(P)$ obtained using both. Similar conventions are applied to $\mathrm{BV}(P)$ to distinguish four variants in total. The correctness of these translations is addressed next.

[8] However, the form in (13) is used in our implementation, since $+_m$ and $<_m$ are amongst the base operators of the BOOLECTOR system.

Theorem 1. *Let P be a normal program and $\mathrm{BV}(P)$ its bit-vector translation.*

1. *If S is an answer set of P, then there is a model $\langle M, \tau \rangle$ of $\mathrm{BV}(P)$ such that $S = M \cap \mathrm{At}(P)$.*
2. *If $\langle M, \tau \rangle$ is a model of $\mathrm{BV}(P)$, then $S = M \cap \mathrm{At}(P)$ is an answer set of P.*

Proof. To establish the correspondence of answer sets and models as formalized above, we appeal to the analogous property of the translation of P into difference logic (DL), denoted here by $\mathrm{DL}(P)$. In DL, theory atoms $x \leq y + k$ constrain the difference of two integer variables x and y. Models can be represented as pairs $\langle I, \tau \rangle$ where I is a propositional interpretation and τ maps free constants of theory atoms to integers so that $\langle I, \tau \rangle \models x \leq y + k \iff \tau(x) \leq \tau(y) + k$. The rest is analogous to Definition 2.

($\implies$) Suppose that S is an answer set of P. Then the results of [18] imply that there is a model $\langle M, \tau \rangle$ of $\mathrm{DL}(P)$ such that $S = M \cap \mathrm{At}(P)$. Since the valuation τ ranges over integers, the image of each non-trivial SCC S of $\mathrm{DG}^+(P)$ under τ might contain gaps. Thus, we have to condense τ for each S as follows. An SCC S is partitioned into $S_0 \sqcup \ldots \sqcup S_n$ such that (i) $\tau(x_a) = \tau(x_b)$ for each $0 \leq i \leq n$ and $a, b \in S_i$, (ii) $\tau(x_a) = \tau(z)$[9] for each $a \in S_0$, and (iii) for each $0 \leq i < j \leq n$, $a \in S_i$, and $b \in S_j$, $\tau(x_a) \leq \tau(x_b)$. Then define τ' for the bit vector x_a associated with an atom $a \in S_i$ by setting $\tau'(x_a, j) = 1$ iff the j^{th} bit of $\bar{i}$ is 1, i.e., $\tau'(x_a) = i$. It follows that $\langle I, \tau \rangle \models x_b \leq x_a - 1$ iff $\langle I, \tau' \rangle \models x_b <_m x_a$ for any $a, b \in S$. Moreover, we have $\langle M, \tau \rangle \models (x_a \leq z + 0) \wedge (z \leq x_a + 0)$ iff $\langle M, \tau' \rangle \models x_a =_m \bar{0}$ for any $a \in S$. Due to the similar structures of $\mathrm{DL}(P)$ and $\mathrm{BV}(P)$, we obtain $\langle M, \tau \rangle \models \mathrm{BV}(P)$ as desired.

($\impliedby$) Let $\langle M, \tau \rangle$ be a model of $\mathrm{BV}(P)$. Then define τ' such that $\tau'(x) = \sum_{i=1}^{\mathrm{w}(x)} (\tau(x, i) \cdot 2^{\mathrm{w}(x) - i})$ where x on the left hand side stands for the integer variable corresponding to the bit vector x on the right hand side. It follows that $\langle I, \tau \rangle \models x_b <_m x_a$ iff $\langle I, \tau' \rangle \models x_b \leq x_a - 1$. By setting $\tau'(z) = 0$, we obtain $\langle M, \tau \rangle \models x_a =_m \bar{0}$ if and only if $\langle M, \tau' \rangle \models (x_a \leq z + 0) \wedge (z \leq x_a + 0)$. The strong analogy present in the structures of $\mathrm{BV}(P)$ and $\mathrm{DL}(P)$ implies that $\langle M, \tau' \rangle$ is a model of $\mathrm{DL}(P)$. Thus, $S = M \cap \mathrm{At}(P)$ is an answer set of P by [18]. $\square$

Even tighter relationships of answer sets and models can be established for the translations $\mathrm{BV}^{\mathrm{l}}(P)$, $\mathrm{BV}^{\mathrm{g}}(P)$, and $\mathrm{BV}^{\mathrm{lg}}(P)$. It can be shown that the model $\langle M, \tau \rangle$ of $\mathrm{BV}^*(P)$ corresponding to an answer set S of P is unique, i.e., there is no other model $\langle N, \tau' \rangle$ of the translation such that $S = N \cap \mathrm{At}(P)$. These results contrast with [18]: the analogous extensions $\mathrm{DL}^*(P)$ guarantee the uniqueness of M in a model $\langle M, \tau \rangle$ but there are always infinitely many copies $\langle M, \tau' \rangle$ of $\langle M, \tau \rangle$ such that $\langle M, \tau' \rangle \models \mathrm{DL}^*(P)$. Such a valuation τ' can be simply obtained by setting $\tau'(x) = \tau(x) + 1$ for any x.

[9] A special variable z is used as a placeholder for the constant 0 in the translation $\mathrm{DL}(P)$ [18].

4 Native Support for Extended Rule Types

The input syntax of the SMODELS system was soon extended by further rule types [27]. In solver interfaces, the rule types usually take the following simple syntactic forms:

$$\{a_1,\ldots,a_k\} \leftarrow b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m. \tag{16}$$

$$a \leftarrow l\{b_1,\ldots,b_n,\sim c_1,\ldots,\sim c_m\}. \tag{17}$$

$$a \leftarrow l\{b_1 = w_{b_1},\ldots,b_n = w_{b_n},\sim c_1 = w_{c_1},\ldots,\sim c_m = w_{c_m}\}. \tag{18}$$

The body of a *choice rule* (16) is interpreted in the same way as that of a normal rule (1). The head, in contrast, allows to derive any subset of atoms $a_1,\ldots,a_k$, if the body is satisfied, and to make a *choice* in this way. The head a of a *cardinality rule* (17) is derived, if its body is satisfied, i.e., the number of satisfied literals amongst $b_1,\ldots,b_n$ and $\sim c_1,\ldots,\sim c_m$ is at least l acting as the *lower bound*. A *weight rule* of the form (18) generalizes this idea by assigning arbitrary positive weights to literals (rather than 1s). The body is satisfied if the sum of weights assigned to satisfied literals is at least l, thus, enabling one to infer the head a using the rule. In practice, the grounding components used in ASP systems allow for more versatile use of cardinality and weight rules, but the primitive forms (16), (17), and (18) provide a solid basis for efficient implementation via translations. The reader is referred to [27] for a generalization of answer sets for programs involving such extended rule types. The respective class of *weight constraint programs* (WCPs) is typically supported by SMODELS-compatible systems.

Whenever appropriate, it is possible to translate extended rule types as introduced above back to normal rules. To this end, a number of transformations are addressed in [17] and they have been implemented as a tool called LP2NORMAL.[10] For instance, the head of a choice rule (16) can be captured in terms of rules

$$\begin{array}{lll} a_1 \leftarrow b, \sim \overline{a_1}. & \ldots & a_k \leftarrow b, \sim \overline{a_k}. \\ \overline{a_1} \leftarrow \sim a_1. & \ldots & \overline{a_k} \leftarrow \sim a_k. \end{array}$$

where $\overline{a_1},\ldots,\overline{a_k}$ are new atoms and b is a new atom standing for the body of (16) which can be defined using (16) with the head replaced by b. We assume that this transformation is applied at first to remove choice rules when the goal is to translate extended rule types into bit-vector logic. The strengths of this transformation are *locality*, i.e., it can be applied on a rule-by-rule basis, and *linearity* with respect to the length of the original rule (16). In contrast, linear normalization of cardinality and weight rules seems impossible. Thus, we also provide direct translations into formulas of bit-vector logic.

We present the translation of a weight rule (18) whereas the translation of a cardinality rule (17) is obtained as a special case $w_{b_1} = \ldots = w_{b_n} = w_{c_1} = \ldots = w_{c_m} = 1$. The body of a weight rule can be evaluated using bit vectors $s_1,\ldots,s_{n+m}$ of

[10] http://www.tcs.hut.fi/Software/asptools/

width $k = \lceil \log_2(\sum_{i=1}^{n} w_{b_i} + \sum_{i=1}^{m} w_{c_i} + 1) \rceil$ constrained by $2 \times (n+m)$ formulas

$$\begin{array}{ll}
b_1 \rightarrow (s_1 =_k \overline{w_{b_1}}), & \neg b_1 \rightarrow (s_1 =_k \overline{0}), \\
b_2 \rightarrow (s_2 =_k s_1 +_k \overline{w_{b_2}}), & \neg b_2 \rightarrow (s_2 =_k s_1), \\
\vdots & \vdots \\
b_n \rightarrow (s_n =_k s_{n-1} +_k \overline{w_{b_n}}), & \neg b_n \rightarrow (s_n =_k s_{n-1}), \\
 & \\
c_1 \rightarrow (s_{n+1} =_k s_n), & \neg c_1 \rightarrow (s_{n+1} =_k s_n +_k \overline{w_{c_1}}), \\
\vdots & \vdots \\
c_m \rightarrow (s_{n+m} =_k s_{n+m-1}), & \neg c_m \rightarrow (s_{n+m} =_k s_{n+m-1} +_k \overline{w_{c_m}}).
\end{array}$$

The translation formalizes a case analysis where the truth values of literals are checked one-by-one and the resulting weight sum is accumulated in bit vectors $s_1, \ldots, s_{n+m}$. The final sum appears as the value of s_{n+m}. The lower bound l of (18) can be checked in terms of the formula $\neg(s_{n+m} <_k \bar{l})$ where we assume that $\bar{l}$ is of width k, since the rule can be safely deleted otherwise. In view of the overall translation, the formula $\mathrm{bd}_r \leftrightarrow \neg(s_{n+m} <_k \bar{l})$ can be used in conjunction with the completion formula (6).

Weight rules also contribute to the dependency graph $\mathrm{DG}^+(P)$ in analogy to normal rules, i.e., the head a potentially depends on all positive body atoms $b_1, \ldots, b_n$ in (18). For this reason, the effect on the external (8) and internal (9) support of the head atom a has to be formalized in analogy to [19]. The contribution of a weight rule r with $\mathrm{hd}(r) = a$ to the external support of a amounts to a further disjunct to be incorporated in (8). The condition checks whether the weight sum for satisfied positive literals in $\mathrm{bd}^+(r) \setminus \mathrm{SCC}(a)$ and negative literals in $\{\sim c \mid c \in \mathrm{bd}^-(r)\}$ is greater than equal to the lower bound l. The translation for this condition is obtained from the translation presented above by dropping the positive literals of $\mathrm{SCC}(a)$ from the case analysis. A similar condition for checking the internal support provided by a weight rule r can be devised. Probably the easiest way is to introduce a new atom s_b for each $b \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)$ and to make it equivalent to the conjunction appearing in (9) using a formula

$$s_b \leftrightarrow \mathrm{bd}_r \wedge \bigwedge_{b' \in \mathrm{bd}^+(r) \cap \mathrm{SCC}(a)} (x_{b'} <_m x_a). \tag{19}$$

The intuitive meaning of s_b is that b has internal support. These new atoms can then be substituted for the positive literals of $\mathrm{bd}^+(r) \cap \mathrm{SCC}(a)$ in the translation. With a little bit of reorganization most of the translations for the completion, ext_a, and int_a may share structure. For instance, the case analysis over negative literals can be done only once and the resulting weight sum can be used as the initial value for the other weight sums of interest. To summarize, we have basically explained how the translation $\mathrm{BV}(P)$ can be generalized for programs P having rules of the extended types (16)–(18).

```
gringo program.lp instance.lp \
| smodels -internal -nolookahead \
| lpcat -s=symbols.txt \
| lp2bv [-l] [-g] \
| boolector -fm
```

Fig. 1. Unix shell pipeline for running a benchmark instance

5 Experimental Results

A new translator called LP2BV was implemented as a derivative of LP2DIFF[11] that translates logic programs into difference logic. In contrast, the new translator provides its output in the bit-vector format. In analogy to its predecessor, it expects to receive its input in the SMODELS[12] file format. Models of the resulting bit-vector theory are searched for using BOOLECTOR[13] (v. 1.4.1) [6] and Z3[14] (v. 2.11) [9] as back-end solvers. The goal of our preliminary experiments was to see how the performances of systems based on LP2BV compare with the performance of a state-of-the-art ASP solver CLASP[15] (v. 1.3.5) [11]. The experiments were based on the NP-complete benchmarks of the ASP Competition 2009. In this benchmark collection, there are 23 benchmark problems with 516 instances in total. Before invoking a translator and the respective SMT solver, we performed a few preprocessing steps using the following tools:

– GRINGO (v. 2.0.5), for grounding the problem encoding and a given instance;
– SMODELS[16] (v. 2.34), for simplifying the resulting ground program;
– LPCAT (v. 1.18), for removing all unused atom numbers, for making the atom table of the ground program contiguous, and for extracting the symbols for later use; and
– LP2NORMAL (version 1.11), for normalizing the program.

The last step is optional and not included as part of the pipeline in Fig. 1. Pipelines of this kind were executed under Linux/Ubuntu operating system running on six-core AMD Opteron$^{(\mathrm{TM})}$ 2435 processors under 2.6 GHz clock rate and with 2.7 GB memory limit that corresponds to the amount of memory available in the ASP Competition 2009.

For each system based on a translator and a back-end solver, there are four variants of the system to consider: W indicates that only weak ranking constraints are used, while L, G, and LG mean that either local, or global, or both local and global strong ranking constraints, respectively, are employed when translating the logic program.

[11] http://www.tcs.hut.fi/Software/lp2diff/
[12] http://www.tcs.hut.fi/Software/smodels/
[13] http://fmv.jku.at/boolector/
[14] http://research.microsoft.com/en-us/um/redmond/projects/z3/
[15] http://www.cs.uni-potsdam.de/clasp/
[16] http://www.tcs.hut.fi/Software/smodels/

Table 1. Experimental results without normalization

	INST	CLASP	LP2BV+BOOLECTOR				LP2BV+Z3			
Benchmark			W	L	G	LG	W	L	G	LG
Overall Performance	516	465 347/118	**276 188/88**	244 161/83	261 174/87	256 176/80	217 142/75	216 147/69	194 124/70	204 135/69
KnightTour	10	8/0	**2/0**	1/0	0/0	0/0	1/0	0/0	0/0	1/0
GraphColouring	29	8/0	**7/0**	**7/0**	**7/0**	**7/0**	6/0	**7/0**	**7/0**	**7/0**
WireRouting	23	11/11	**2/3**	1/1	1/2	0/2	1/3	0/0	0/0	0/1
DisjunctiveScheduling	10	5/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
GraphPartitioning	13	6/7	3/0	3/0	3/0	3/0	**4/0**	**4/0**	**4/0**	3/0
ChannelRouting	11	6/2	**6/2**	**6/2**	**6/2**	**6/2**	5/2	**6/2**	**6/2**	**6/2**
Solitaire	27	19/0	2/0	**5/0**	1/0	4/0	0/0	0/0	0/0	0/0
Labyrinth	29	26/0	**1/0**	0/0	0/0	0/0	0/0	0/0	0/0	0/0
WeightBoundedDominatingSet	29	26/0	**18/0**	**18/0**	17/0	**18/0**	12/0	12/0	11/0	12/0
MazeGeneration	29	10/15	**8/15**	1/15	0/15	0/16	5/16	1/15	0/15	1/15
15Puzzle	16	16/0	**16/0**	15/0	14/0	15/0	4/0	4/0	5/0	5/0
BlockedNQueens	29	15/14	**2/2**	0/2	1/2	0/2	1/0	2/0	2/0	0/0
ConnectedDominatingSet	21	10/10	**10/11**	9/8	**10/11**	6/3	10/10	9/10	10/9	10/9
EdgeMatching	29	29/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Fastfood	29	10/19	9/16	**10/16**	**10/16**	9/16	9/9	9/9	9/10	9/9
GeneralizedSlitherlink	29	29/0	**29/0**	20/0	**29/0**	**29/0**	**29/0**	**29/0**	16/0	**29/0**
HamiltonianPath	29	29/0	27/0	25/0	**29/0**	28/0	26/0	27/0	25/0	26/0
Hanoi	15	15/0	**15/0**	**15/0**	**15/0**	**15/0**	5/0	5/0	5/0	4/0
HierarchicalClustering	12	8/4	**8/4**	**8/4**	**8/4**	**8/4**	4/4	4/4	4/4	4/4
SchurNumbers	29	13/16	6/16	5/16	5/16	5/16	**9/16**	**9/16**	**9/16**	**9/16**
Sokoban	29	9/20	**9/19**	8/19	8/19	8/19	7/15	7/13	7/14	5/13
Sudoku	10	10/0	**5/0**	4/0	4/0	**5/0**	4/0	4/0	4/0	4/0
TravellingSalesperson	29	29/0	3/0	0/0	6/0	**10/0**	0/0	8/0	0/0	0/0

Table 1 collects the results from our experiments without normalization whereas Table 2 shows the results when LP2NORMAL [17] was used to remove extended rule types discussed in Sect. 4. In both tables, the first column gives the name of the benchmark, followed by the number of instances of that particular benchmark in the second column. The following columns indicate the numbers of instances that were solved by the systems considered in our experiments. A notation like 8/4 means that the system was able to solve eight satisfiable and four unsatisfiable instances in that particular benchmark. Hence, if there are 15 instances in a benchmark and the system could only solve 8/4, this means that the system was unable to solve the remaining three instances within the time limit of 600 seconds, i.e., ten minutes, per instance.[17] As regards the number of solved instances in each benchmark, the best performing translation-based approaches are highlighted in boldface. We also run the experiments using translator LP2DIFF with Z3 as back-end solver. Rather than providing detailed performance statistics, a summary can be found from Table 3—giving an overview of experimental results in terms of total numbers of instances solved out of 516.

It is apparent that the systems based on LP2BV did not perform very well without normalization. As indicated by Table 3, the overall performance was

[17] One observation is that the performance of systems based on LP2BV is quite stable: even when we extended the time limit to 20 minutes, the results did not change much (differences of only one or two instances were perceived in most cases).

Table 2. Experimental results with normalization

	INST	CLASP	LP2BV+BOOLECTOR				LP2BV+Z3			
Benchmark			W	L	G	LG	W	L	G	LG
Overall Performance	516	459	**381**	343	379	**381**	346	330	325	331
		346/113	**279/102**	243/100	278/101	**281/100**	240/106	231/99	224/101	232/99
KnightTour	10	10/0	**2/0**	**2/0**	1/0	0/0	1/0	0/0	0/0	0/0
GraphColouring	29	9/0	8/0	8/0	8/0	8/0	**9/2**	**9/2**	**9/2**	**9/2**
WireRouting	23	11/11	2/6	1/3	1/3	1/3	**2/7**	1/4	1/4	1/3
DisjunctiveScheduling	10	5/0	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**	**5/0**
GraphPartitioning	13	4/1	**5/0**	**5/0**	4/0	**5/0**	2/1	2/1	2/1	2/0
ChannelRouting	11	6/2	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**	**6/2**
Solitaire	27	18/0	**23/0**	**23/0**	**23/0**	**23/0**	22/0	22/0	22/0	22/0
Labyrinth	29	27/0	1/0	1/0	2/0	**3/0**	0/0	0/0	0/0	0/0
WeightBoundedDominatingSet	29	25/0	15/0	15/0	15/0	**16/0**	10/0	10/0	10/0	10/0
MazeGeneration	29	10/15	**8/15**	0/15	0/15	0/16	5/16	0/15	0/15	0/15
15Puzzle	16	15/0	**16/0**	**16/0**	**16/0**	**16/0**	11/0	10/0	11/0	11/0
BlockedNQueens	29	15/14	14/14	14/14	14/14	14/14	**15/14**	**15/14**	**15/14**	**15/14**
ConnectedDominatingSet	21	10/11	**10/11**	8/11	9/11	9/10	**10/11**	9/11	9/11	9/11
EdgeMatching	29	29/0	**29/0**	**29/0**	**29/0**	**29/0**	**29/0**	**29/0**	**29/0**	**29/0**
Fastfood	29	10/19	9/14	9/15	**9/16**	9/15	0/13	0/10	0/12	0/12
GeneralizedSlitherlink	29	29/0	**29/0**	21/0	**29/0**	**29/0**	**29/0**	**29/0**	21/0	**29/0**
HamiltonianPath	29	29/0	**29/0**	28/0	**29/0**	**29/0**	**29/0**	**29/0**	**29/0**	**29/0**
Hanoi	15	15/0	**15/0**	**15/0**	**15/0**	**15/0**	**15/0**	**15/0**	**15/0**	**15/0**
HierarchicalClustering	12	8/4	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**	**8/4**
SchurNumbers	29	13/16	10/16	10/16	9/16	10/16	**13/16**	**13/16**	**13/16**	**13/16**
Sokoban	29	9/20	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**	**9/20**
Sudoku	10	10/0	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**	**10/0**
TravellingSalesperson	29	29/0	16/0	0/0	**27/0**	**27/0**	0/0	0/0	0/0	0/0

Table 3. Summary of the experimental results

System	W	L	G	LG
LP2BV+BOOLECTOR	276	244	261	256
LP2BV+Z3	217	216	194	204
LP2DIFF+Z3	360	349	324	324
CLASP	465			
LP2NORMAL2BV+BOOLECTOR	381	343	379	381
LP2NORMAL2BV+Z3	346	330	325	331
LP2NORMAL2DIFF+Z3	364	357	349	349
LP2NORMAL+CLASP	459			

even worse than that of systems using LP2DIFF for translation and Z3 for model search. However, if the input was first translated into a normal logic program using LP2NORMAL, i.e., before translation into a bit-vector theory, the performance was clearly better. Actually, it exceeded that of the systems based on LP2DIFF and became closer to that of CLASP. We note that normalization does not help so much in case of LP2DIFF and the experimental results obtained using both normalized and unnormalized instances are quite similar in terms of solved instances. Thus, it seems that solvers for bit-vector logic are not able to make the best of native translations of cardinality and weight rules from Section 4.

We anticipate that this drawback goes back to the fact that BOOLECTOR internally translates bit vectors into Boolean variables. Hence, the treatment of case analysis formulas and bit vectors involved in the native translations can become computationally costly. However, if an analogous translation into difference logic is used, as implemented by LP2DIFF, such a negative effect was not perceived using Z3. Our understanding is that the efficient graph-theoretic satisfiability check for difference constraints used in the search procedure of Z3 turns the native translation feasible as well. As indicated by our test results, BOOLECTOR is clearly better back-end solver for LP2BV than Z3. This was to be expected since BOOLECTOR is a native solver for bit-vector logic whereas Z3 supports a wider variety of SMT fragments and is in that respect more generic. In addition, the design of LP2BV takes into account operators of bit-vector logic which are directly supported by BOOLECTOR and not implemented as syntactic sugar.

In addition, we note on the basis of our results that the performance of the state-of-the-art ASP solver CLASP is significantly better, and the translation-based approaches to computing stable models are still left behind. By the results of Table 2, even the best variants of systems based on LP2BV did not work well enough to compete with CLASP. The difference is especially due to the following benchmarks: *Knight Tour*, *Wire Routing*, *Graph Partitioning*, *Labyrinth*, *Weight Bounded Dominating Set*, *Fastfood*, and *Travelling Salesperson*. All of them involve either recursive rules (*Knight Tour*, *Wire Routing*, and *Labyrinth*), weight rules (*Weight Bounded Dominating Set* and *Fastfood*), or both (*Graph Partitioning* and *Travelling Salesperson*). Hence, it seems that handling recursive rules and weight constraints in the translational approach is less efficient compared to their native implementation in CLASP. When using the current normalization techniques to remove cardinality and weight rules, the sizes of ground programs tend to increase significantly and, in particular, if weight rules are abundant. For example, after normalization the ground programs are ten times larger for the benchmark *Weight Bounded Dominating Set*, and five times larger for *Fastfood*. It is also worth pointing out that the efficiency of CLASP turned out to be insensitive to normalization.

While having trouble with recursive rules and weight constraints for particular benchmarks, the translational approach handles certain large instances quite well. The largest instances in the experiments belong to the *Disjunctive Scheduling* benchmark, of which all instances are ground programs of size over one megabyte but after normalization[18] the systems based on LP2BV can solve as many instances as CLASP.

6 Conclusion

In this paper, we present a novel and concise translation from normal logic programs into fixed-width bit-vector theories. Moreover, the extended rule types supported by SMODELS-compatible answer set solvers can be covered via native

[18] In this benchmark, normalization does not affect the size of grounded programs significantly.

translations. The length of the resulting translation is linear with respect to the length of the original program. The translation has been implemented as a translator, LP2BV, which enables the use of bit-vector solvers in the search for answer sets. Our preliminary experimental results indicate a level of performance which is similar to that obtained using solvers for difference logic. However, this presumes one first to translate extended rule types into normal rules and then to apply the translation into bit-vector logic. One potential explanation for such behavior is the way in which SMT solvers implement reasoning with bit vectors: a predominant strategy is to translate theory atoms involving bit vectors into propositional formulas and to apply satisfiability checking techniques systematically. We anticipate that an improved performance could be obtained if native support for certain bit vector primitives were incorporated into SMT solvers directly. When comparing with the state-of-the-art ASP solver CLASP, we noticed that the performance of the translation-based approach compared unfavorably, in particular, in benchmarks which contained recursive rules or weight constraints or both. This indicates that the performance can be improved by developing new translation techniques for these two features. In order to obtain a more comprehensive view of the performance characteristics of the translational approach, the plan is to extend our experimental setup to include benchmarks that were used in the third ASP competition [7]. Moreover, we intend to use the new SMT library format[19] in the future versions of our translators.

Acknowledgments. This research has been partially funded by the Academy of Finland under the project "*Methods for Constructing and Solving Large Constraint Models*" (MCM, #122399).

References

1. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann, Los Altos (1988)
2. Balduccini, M.: Industrial-size scheduling with ASP+CP. In: Delgrande, J.P. (ed.) LPNMR 2011. LNCS, vol. 6645, pp. 284–296. Springer, Heidelberg (2011)
3. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
4. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
5. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
6. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)

[19] http://goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.12.21.pdf

7. Calimeri, F., et al.: The third answer set programming competition: preliminary report of the system competition track. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 388–403. Springer, Heidelberg (2011)
8. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press, New York (1977)
9. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
11. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp*: a conflict-driven answer set solver. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 260–265. Springer, Heidelberg (2007)
12. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 235–249. Springer, Heidelberg (2009)
13. Gelfond, M., Leone, N.: Logic programming and knowledge representation - the A-Prolog perspective. Artif. Intell. **138**(1–2), 3–38 (2002)
14. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080 (1988)
15. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. J. Appl. Non-Classical Logics **16**(1–2), 35–86 (2006)
16. Janhunen, T., Liu, G., Niemelä, I.: Tight integration of non-ground answer set programming and satisfiability modulo theories. In: Working Notes of Grounding and Transformations for Theories with Variables, Vancouver, Canada, pp. 1–13, May 2011
17. Janhunen, T., Niemelä, I.: Compact Translations of Non-disjunctive Answer Set Programs to Propositional Clauses. In: Balduccini, M., Son, T.C. (eds.) Logic programming, knowledge representation, and nonmonotonic reasoning, vol. 6565, pp. 111–130. Springer, Heidelberg (2011)
18. Janhunen, T., Niemelä, I., Sevalnev, M.: Computing stable models via reductions to difference logic. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 142–154. Springer, Heidelberg (2009)
19. Liu, G., Janhunen, T., Niemelä, I.: Answer set programming via mixed integer programming. In: Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR), pp. 32–42. AAAI Press (2012)
20. Marek, V., Subrahmanian, V.: The relationship between stable, supported, default and autoepistemic semantics for general logic programs. Theor. Comput. Sci. **103**(2), 365–386 (1992)
21. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K.R., et al. (eds.) The Logic Programming Paradigm: A 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1999)
22. Mellarkod, V.S., Gelfond, M.: Integrating answer set reasoning with constraint solving techniques. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 15–31. Springer, Heidelberg (2008)
23. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. **25**(3–4), 241–273 (1999)
24. Niemelä, I.: Stable models and difference logic. Ann. Math. Artif. Intell. **53**(1–4), 313–329 (2008)

25. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
26. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory Pract. Logic Program. **8**(5–6), 717–761 (2008)
27. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. **138**(1–2), 181–234 (2002)

Making Use of Advances in Answer-Set Programming for Abstract Argumentation Systems

Wolfgang Dvořák[3], Sarah Alice Gaggl[2], Johannes Peter Wallner[1](✉), and Stefan Woltran[1]

[1] Institute of Information Systems, Database and Artificial Intelligence Group, Vienna University of Technology, Favoritenstraße 9-11, 1040 Wien, Austria
{wallner, woltran}@dbai.tuwien.ac.at

[2] Technische Universität Dresden, Institute of Artificial Intelligence, Nöthnitzer Straße 46, Dresden-Räcknitz, D-01062 Dresden, Germany
sarah.gaggl@tu-dresden.de

[3] Faculty of Computer Science, University of Vienna, Währinger Straße 29/6.49, 1090 Wien, Austria
wolfgang.dvorak@univie.ac.at

Abstract. Dung's famous abstract argumentation frameworks represent the core formalism for many problems and applications in the field of argumentation which significantly evolved within the last decade. Recent work in the field has thus focused on implementations for these frameworks, whereby one of the main approaches is to use Answer-Set Programming (ASP). While some of the argumentation semantics can be nicely expressed within the ASP language, others required rather cumbersome encoding techniques. Recent advances in ASP systems, in particular, the `metasp` optimization front-end for the ASP-package `gringo/claspD` provide direct commands to filter answer sets satisfying certain subset-minimality (or -maximality) constraints. This allows for much simpler encodings compared to the ones in standard ASP language. In this paper, we experimentally compare the original encodings (for the argumentation semantics based on preferred, semi-stable, and respectively, stage extensions) with new `metasp` encodings. Moreover, we provide novel encodings for the recently introduced resolution-based grounded semantics. Our experimental results indicate that the `metasp` approach works well in those cases where the complexity of the encoded problem is adequately mirrored within the `metasp` approach.

Keywords: Abstract argumentation · Answer-set programming · Meta programming

This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT08-028.

1 Introduction

In Artificial Intelligence (AI), the area of argumentation (the survey by Bench-Capon and Dunne [3] gives an excellent overview) has become one of the central issues during the last decade. Although there are now several branches within this area, there is a certain agreement that Dung's famous abstract argumentation frameworks (AFs) [7] still represent the core formalism for many of the problems and applications in the field. In a nutshell, AFs formalize statements together with a relation denoting rebuttals between them, such that the semantics gives a handle to solve the inherent conflicts between statements by selecting admissible subsets of them, but without taking the concrete contents of the statements into account. Several semantical principles how to select those subsets have already been proposed by Dung [7] but numerous other proposals have been made over the last years. In this paper we shall focus on the preferred [7], semi-stable [4], stage [18], and the resolution-based grounded semantics [1]. Each of these semantics is based on some kind of $\subseteq$-maximality (resp. -minimality) and thus is well amenable for the novel `metasp` concepts which we describe below.

Let us first talk about the general context of the paper, which is the realization of abstract argumentation within the paradigm of Answer-Set Programming (see [17] for an overview). We follow here the ASPARTIX[1]approach [11], where a single program is used to encode a particular argumentation semantics, while the instance of an argumentation framework is given as an input database. For problems located on the second level of the polynomial hierarchy (i.e. for preferred, stage, and semi-stable semantics) ASP encodings turned out to be quite complicated and hardly accessible for non-experts in ASP (we will sketch here the encoding for the stage semantics in some detail, since it has not been presented in [11]). This is due to the fact that tests for subset-maximality have to be done "by hand" in ASP requiring a certain saturation technique. However, recent advances in ASP solvers, in particular, the `metasp` optimization front-end for the ASP-system `gringo/claspD` allows for much simpler encodings for such tests. More precisely, `metasp` allows to use the traditional #*minimize* statement (which in its standard variant minimizes wrt. cardinality or weights, but not wrt. subset inclusion) also for selection among answer sets which are minimal wrt. subset inclusion in certain predicates. Details about `metasp` can be found in [13].

Our first main contribution will be the practical comparison between handcrafted encodings (i.e. encodings in the standard ASP language without the new semantics for the #*minimize* statement) and the much simpler `metasp` encodings for argumentation semantics. The experiments show that the `metasp` encodings do not necessarily result in longer runtimes. In fact, the `metasp` encodings for the semantics located on the second level of the polynomial hierarchy outperform the handcrafted saturation-based encodings. We thus can give additional

[1] See http://rull.dbai.tuwien.ac.at:8080/ASPARTIX for a web front-end of ASPARTIX.

evidence to the observations in [13], where such a speed-up was reported for encodings in a completely different application area.

Our second contribution is the presentation of ASP encodings for the resolution-based grounded semantics [1]. To the best of our knowledge, no implementation for this recently proposed semantics has been released so far. In this paper, we present a rather involved handcrafted encoding (basically following the NP-algorithm presented in [1]) but also two much simpler encodings (using `metasp`) which rely on the original definition of the semantics.

Our results indicate that `metasp` is a very useful tool for problems known to be hard for the second-level, but one might loose performance in case `metasp` is used for "easier" problems just for the sake of comfortability. Nonetheless, we believe that the concept of the advanced #*minimize* statement is vital for ASP, since it allows for rapid prototyping of second-level encodings without being an ASP guru.

The remainder of the paper is organized as follows: Section 2 provides the necessary background. Section 3 then contains the ASP encodings for the argumentation semantics we are interested in this work. We begin with the handcrafted saturation-based encoding for stage semantics. Then, in Sect. 3.2 we provide the novel `metasp` encodings for all considered semantics and afterwards, in Sect. 3.3, we present an alternative encoding for the resolution-based grounded semantics which better mirrors the complexity of this semantics. Section 4 then presents our experimental evaluation. We conclude the paper with a brief summary and discussion for future research directions.

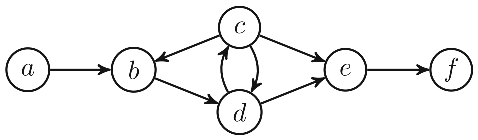
2 Background

2.1 Abstract Argumentation

In this section we introduce (abstract) argumentation frameworks [7] and recall the semantics we study in this paper (see also [1,2]). Moreover, we highlight complexity results for typical decision problems associated to such frameworks.

Definition 1. *An* argumentation framework (AF) *is a pair $F = (A, R)$ where A is a set of arguments and $R \subseteq A \times A$ is the attack relation. The pair $(a, b) \in R$ means that a attacks b.An argument $a \in A$ is* defended *by a set $S \subseteq A$ if, for each $b \in A$ such that $(b, a) \in R$, there exists a $c \in S$ such that $(c, b) \in R$.*

Example 1. Consider the AF $F = (A, R)$ with $A = \{a, b, c, d, e, f\}$ and $R = \{(a, b), (b, d), (c, b), (c, d), (c, e), (d, c), (d, e), (e, f)\}$, and the graph representation of F:



Semantics for argumentation frameworks are given via a function σ which assigns to each AF $F = (A, R)$ a set $\sigma(F) \subseteq 2^A$ of extensions. We shall consider here for σ the functions *stb*, *adm*, *com*, *prf*, *grd*, *grd**, *stg*, and *sem* which stand for stable, admissible, complete, preferred, grounded, resolution-based grounded, stage, and semi-stable semantics respectively. Towards the definition of these semantics we have to introduce two more formal concepts.

Definition 2. *Given an AF $F = (A, R)$. The characteristic function $\mathcal{F}_F : 2^A \Rightarrow 2^A$ of F is defined as $\mathcal{F}_F(S) = \{x \in A \mid x$ is defended by $S\}$. Moreover, for a set $S \subseteq A$, we denote the set of arguments attacked by S as $S^{\oplus}_R = \{x \mid \exists y \in S$ such that $(y,x) \in R\}$, and define the* range of S *as $S^+_R = S \cup S^{\oplus}_R$.*

Definition 3. *Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is* conflict-free (in F), *if there are no $a, b \in S$, such that $(a, b) \in R$. $cf(F)$ denotes the collection of conflict-free sets of F. For a conflict-free set $S \in cf(F)$, it holds that*

- *$S \in stb(F)$, if $S^+_R = A$;*
- *$S \in adm(F)$, if $S \subseteq \mathcal{F}_F(S)$;*
- *$S \in com(F)$, if $S = \mathcal{F}_F(S)$;*
- *$S \in grd(F)$, if $S \in com(F)$ and there is no $T \in com(F)$ with $T \subset S$;*
- *$S \in prf(F)$, if $S \in adm(F)$ and there is no $T \in adm(F)$ with $T \supset S$;*
- *$S \in sem(F)$, if $S \in adm(F)$ and there is no $T \in adm(F)$ with $T^+_R \supset S^+_R$;*
- *$S \in stg(F)$, if there is no $T \in cf(F)$ in F, such that $T^+_R \supset S^+_R$.*

We recall that for each AF F, the grounded semantics yields a unique extension, the grounded extension, which is the least fixed point of the characteristic function $\mathcal{F}_F$.

Example 2. Consider the AF F from Example 1. We have $\{a, d, f\}$ and $\{a, c, f\}$ as the stable extensions and thus $stb(F) = stg(F) = sem(F) = \{\{a, d, f\}, \{a, c, f\}\}$. The admissible sets of F are $\{\}, \{a\}, \{c\}, \{a, c\}, \{a, d\}, \{c, f\}, \{a, c, f\}, \{a, d, f\}$ and therefore $prf(F) = \{\{a, c, f\}, \{a, d, f\}\}$. Finally we have $com(F) = \{\{a\}, \{a, c, f\}, \{a, d, f\}\}$, with $\{a\}$ being the grounded extension.

On the base of these semantics one can define the family of resolution-based semantics [1], with the resolution-based grounded semantics being the most popular instance.

Definition 4. *A resolution $\beta \subset R$ of an AF $F = (A, R)$ contains exactly one attack from each bidirectional attack in F, i.e. $\forall a, b \in A$, if $(a, b), (b, a) \in R$ then $\mid \{(a, b), (b, a)\} \cap \beta \mid = 1$ and $\{(c, d) \mid (c, d) \in R, (d, c) \notin R\} \cap \beta = \emptyset$. A set $S \subseteq A$ is a* resolution-based grounded extension *of F, denoted by $S \in grd^*(F)$, if (i) there exists a resolution β such that $S = grd((A, R \setminus \beta))$ [2]; and (ii) there is no resolution β' such that $grd((A, R \setminus \beta')) \subset S$.*

Example 3. Recall the AF $F = (A, F)$ from Example 1. There is one mutual attack and thus we have two resolutions $\beta_1 = \{(c, d)\}$ and $\beta_2 = \{(d, c)\}$. Definition 4 gives us two candidates, namely $grd((A, R \setminus \beta_1)) = \{a, d, f\}$ and $grd((A, R \setminus \beta_2)) = \{a, c, f\}$; as they are not in $\subset$-relation they are the resolution-based grounded extensions of F.

[2] Abusing notation slightly, we use $grd(F)$ for denoting the unique grounded extension of F.

Table 1. Complexity of abstract argumentation ($\mathcal{C}$-c denotes completeness for class $\mathcal{C}$).

	prf	*sem*	*stg*	*grd**
Cred_σ	NP-c	Σ_2^P-c	Σ_2^P-c	NP-c
Skept_σ	Π_2^P-c	Π_2^P-c	Π_2^P-c	coNP-c
Ver_σ	coNP-c	coNP-c	coNP-c	in P

We now turn to the complexity of reasoning in AFs. To this end, we define the following decision problems for the semantics σ introduced in Definitions 3 and 4:

– *Credulous Acceptance* Cred_σ: Given AF $F = (A, R)$ and an argument $a \in A$. Is a contained in some $S \in \sigma(F)$?
– *Skeptical Acceptance* Skept_σ: Given AF $F = (A, R)$ and an argument $a \in A$. Is a contained in each $S \in \sigma(F)$?
– *Verification of an extension* Ver_σ: Given AF $F = (A, R)$ and a set of arguments $S \subseteq A$. Is $S \in \sigma(F)$?

We assume the reader has knowledge about standard complexity classes like P and NP and recall that Σ_2^P is the class of decision problems that can be decided in polynomial time using a nondeterministic Turing machine with access to an NP-oracle. The class Π_2^P is defined as the complementary class of Σ_2^P, i.e. $\Pi_2^P = \mathrm{co}\Sigma_2^P$.

In Table 1 we summarize complexity results relevant for our work [1,6,8–10].

2.2 Answer-Set Programming

We give a brief overview of the syntax and semantics of disjunctive logic programs under the answer-sets semantics [15]; for further background, see [16].

We fix a countable set $\mathcal{U}$ of *(domain) elements*, also called *constants*; and suppose a total order $<$ over the domain elements. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity $n \geq 0$ and each t_i is either a variable or an element from $\mathcal{U}$. An atom is *ground* if it is free of variables. $B_\mathcal{U}$ denotes the set of all ground atoms over $\mathcal{U}$.

A (*disjunctive*)*rule* r with $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$ is of the form

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \leftarrow b_1, \ldots, b_k, \ \mathit{not}\, b_{k+1}, \ldots, \ \mathit{not}\, b_m$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms, and "*not*" stands for *default negation*. An atom a is a positive literal, while $\mathit{not}\, a$ is a default negated literal. The *head* of r is the set $H(r) = \{a_1, \ldots, a_n\}$ and the *body* of r is $B(r) = B^+(r) \cup B^-(r)$ with $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. A rule r is *normal* if $n \leq 1$ and a *constraint* if $n = 0$. A rule r is *safe* if each variable in r occurs in $B^+(r)$. A rule r is *ground* if no variable occurs in r. A *fact* is a ground rule without disjunction and with an empty body. An *(input) database* is a set of facts. A program is a finite set of disjunctive rules. For a program π and an input database D, we often write $\pi(D)$ instead of $D \cup \pi$. If each rule in a program is normal

Table 2. Data Complexity for logic programs (all results are completeness results).

e	Normal programs	Disjunctive program	Optimization programs
$\models_c$	NP	Σ_2^P	Σ_2^P
$\models_s$	coNP	Π_2^P	Π_2^P

(resp. ground), we call the program normal (resp. ground). Besides disjunctive and normal program, we consider here the class of optimization programs, i.e. normal programs which additionally contain *#minimize* statements

$$\#minimize[l_1 = w_1@J_1, \ldots, l_k = w_k@J_k] \tag{1}$$

where l_i is a literal, w_i an integer weight and J_i an integer priority level.

For any program π, let U_π be the set of all constants appearing in π. $Gr(\pi)$ is the set of rules $r\tau$ obtained by applying, to each rule $r \in \pi$, all possible substitutions τ from the variables in r to elements of U_π. An *interpretation* $I \subseteq B_{\mathcal{U}}$ *satisfies* a ground rule r iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. I satisfies a ground program π, if each $r \in \pi$ is satisfied by I. A non-ground rule r (resp., a program π) is satisfied by an interpretation I iff I satisfies all groundings of r (resp., $Gr(\pi)$). $I \subseteq B_{\mathcal{U}}$ is an *answer set* of π iff it is a subset-minimal set satisfying the *Gelfond-Lifschitz reduct* $\pi^I = \{H(r) \leftarrow B^+(r) \mid I \cap B^-(r) = \emptyset, r \in Gr(\pi)\}$. For a program π, we denote the set of its answer sets by $\mathcal{AS}(\pi)$.

For semantics of optimization programs, we interpret the *#minimize* statement wrt. subset-inclusion: For any sets X and Y of atoms, we have $Y \subseteq_J^w X$, if for any weighted literal $l = w@J$ occurring in (1), $Y \models l$ implies $X \models l$. Then, M is a collection of relations of the form $\subseteq_J^w$ for priority levels J and weights w. A standard answer set (i.e. not taking the minimize statements into account) Y of π *dominates* a standard answer set X of π wrt. M if there are a priority level J and a weight w such that $X \subseteq_J^w Y$ does not hold for $\subseteq_J^w \in M$, while $Y \subseteq_{J'}^{w'} X$ holds for all $\subseteq_{J'}^{w'} \in M$ where $J' \geq J$. Finally a standard answer set X is an answer set of an optimization program π wrt. M if there is no standard answer set Y of π that dominates X wrt. M.

Credulous and skeptical reasoning in terms of programs is defined as follows. Given a program π and a set of ground atoms A. Then, we write $\pi \models_c A$ (credulous reasoning), if A is contained in some answer set of π; we write $\pi \models_s A$ (skeptical reasoning), if A is contained in each answer set of π.

We briefly recall some complexity results for disjunctive logic programs. In fact, since we will deal with fixed programs we focus on results for data complexity. Depending on the concrete definition of $\models$, we give the complexity results in Table 2 (cf. [5] and the references therein). We note here, that even normal programs together with the optimization technique have a worst case complexity of Σ_2^P (resp. Π_2^P). Inspecting Table 1 one can see which kind of encoding is appropriate for an argumentation semantics.

3 Encodings of AF Semantics

In this section we first show how to represent AFs in ASP and we discuss three programs which we need later on in this section.[3] Then, in Subsect. 3.1 we exemplify on the stage semantics the saturation technique for encodings that solve associated problems which are on the second level of the polynomial hierarchy. In Subsect. 3.2 we will make use of the newly developed `metasp` optimization technique. In Subsect. 3.3 we give an alternative encoding based on the algorithm by Baroni et al. in [1], which respects the lower complexity of resolution-based grounded semantics.

All our programs are fixed which means that the only translation required, is to give an AF F as input database $\hat{F}$ to the program π_σ for a semantics σ. In fact, for an AF $F = (A, R)$, we define $\hat{F}$ as

$$\hat{F} = \{\ \mathrm{arg}(a) \mid a \in A\} \cup \{\mathrm{defeat}(a,b) \mid (a,b) \in R\ \}.$$

In what follows, we use unary predicates in/1 and out/1 to perform a guess for a set $S \subseteq A$, where $\mathrm{in}(a)$ represents that $a \in S$. The following notion of correspondence is relevant for our purposes.

Definition 5. *Let $\mathcal{S} \subseteq 2^{\mathcal{U}}$ be a collection of sets of domain elements and let $\mathcal{I} \subseteq 2^{B_{\mathcal{U}}}$ be a collection of sets of ground atoms. We say that $\mathcal{S}$ and $\mathcal{I}$ correspond to each other, in symbols $\mathcal{S} \cong \mathcal{I}$, iff (i) for each $S \in \mathcal{S}$, there exists an $I \in \mathcal{I}$, such that $\{a \mid \mathrm{in}(a) \in I\} = S$; (ii) for each $I \in \mathcal{I}$, it holds that $\{a \mid \mathrm{in}(a) \in I\} \in \mathcal{S}$; and (iii) $|\mathcal{S}| = |\mathcal{I}|$.*

Consider an AF F. The following program fragment guesses, when augmented by $\hat{F}$, any subset $S \subseteq A$ and then checks whether the guess is conflict-free in F:

$$\begin{aligned}\pi_{cf} = \{\ & \mathrm{in}(X) \leftarrow \mathit{not}\,\mathrm{out}(X), \mathrm{arg}(X);\\ & \mathrm{out}(X) \leftarrow \mathit{not}\,\mathrm{in}(X), \mathrm{arg}(X);\\ & \leftarrow \mathrm{in}(X), \mathrm{in}(Y), \mathrm{defeat}(X,Y)\ \ \}.\end{aligned}$$

Proposition 1. *For any AF F, $cf(F) \cong \mathcal{AS}(\pi_{cf}(\hat{F}))$.*

Sometimes we have to avoid the use of negation. This might either be the case for the saturation technique or if a simple program can be solved without a Guess&Check approach. Then, encodings typically rely on a form of loops where all domain elements are visited and it is checked whether a desired property holds for all elements visited so far. We will use this technique in our saturation-based encoding in the upcoming subsection, but also for computing the grounded extension in Subsect. 3.2. For this purpose, an order $<$ over the domain elements (usually provided by common ASP solvers) is used together with a few helper

[3] We make use of some program modules already defined in [11].

predicates defined in the program $\pi_<$ below; in fact, predicates inf /1, succ/2 and sup /1 denote infimum, successor and supremum of the order $<$.

$$\begin{aligned}\pi_< = \{\ & \mathrm{lt}(X,Y) \leftarrow \mathrm{arg}(X), \mathrm{arg}(Y), X < Y;\\ & \mathrm{nsucc}(X,Z) \leftarrow \mathrm{lt}(X,Y), \mathrm{lt}(Y,Z);\\ & \mathrm{succ}(X,Y) \leftarrow \mathrm{lt}(X,Y), \mathit{not}\, \mathrm{nsucc}(X,Y);\\ & \mathrm{ninf}(Y) \leftarrow \mathrm{lt}(X,Y);\\ & \mathrm{inf}(X) \leftarrow \mathrm{arg}(X), \mathit{not}\, \mathrm{ninf}(X);\\ & \mathrm{nsup}(X) \leftarrow \mathrm{lt}(X,Y);\\ & \mathrm{sup}(X) \leftarrow \mathrm{arg}(X), \mathit{not}\, \mathrm{nsup}(X)\ \}.\end{aligned}$$

Finally, the following module computes for a guessed subset $S \subseteq A$ the range S_R^+ (see Def. 2) of S in an AF $F = (A, R)$.

$$\begin{aligned}\pi_{range} = \{\ & \mathrm{in\_range}(X) \leftarrow \mathrm{in}(X);\\ & \mathrm{in\_range}(X) \leftarrow \mathrm{in}(Y), \mathrm{defeat}(Y,X);\\ & \mathrm{not\_in\_range}(X) \leftarrow \mathrm{arg}(X), \mathit{not}\, \mathrm{in\_range}(X)\ \}.\end{aligned}$$

3.1 Saturation Encodings

In this subsection we make use of the saturation technique introduced by Eiter and Gottlob in [12]. In [11], this technique was already used to encode the preferred and semi-stable semantics. Here we give the encodings for the stage semantics, which is similar to the one of semi-stable semantics, to exemplify the use of the saturation technique.

In fact, for an AF $F = (A, R)$ and $S \in \mathit{cf}(F)$ we need to check whether no $T \in \mathit{cf}(F)$ with $S_R^+ \subset T_R^+$ exists. Therefore we have to guess an arbitrary set T and saturate in case (i) T is not conflict-free, or (ii) $S_R^+ \not\subset T_R^+$. Together with π_{cf} this is done with the following module, where in/1 holds the current guess for S and inN/1 holds the current guess for T. More specifically, rule fail $\leftarrow$ inN(X), inN(Y), defeat(X, Y) checks for (i) and the remaining two rules with fail in the head fire in case $S_R^+ = T_R^+$ (indicated by predicate eqplus/0 described below), or there exists an $a \in S_R^+$ such that $a \notin T_R^+$ (here we use predicate in_range/1 from above and predicate not_in_rangeN/1 which we also present below). As is easily checked one of these two conditions holds exactly if (ii) holds.

$$\begin{aligned}\pi_{satstage} = \{\ & \mathrm{inN}(X) \vee \mathrm{outN}(X) \leftarrow \mathrm{arg}(X);\\ & \mathrm{fail} \leftarrow \mathrm{inN}(X), \mathrm{inN}(Y), \mathrm{defeat}(X,Y);\\ & \mathrm{fail} \leftarrow \mathrm{eqplus};\\ & \mathrm{fail} \leftarrow \mathrm{in\_range}(X), \mathrm{not\_in\_rangeN}(X);\\ & \mathrm{inN}(X) \leftarrow \mathrm{fail}, \mathrm{arg}(X);\\ & \mathrm{outN}(X) \leftarrow \mathrm{fail}, \mathrm{arg}(X);\\ & \leftarrow \mathit{not}\, \mathrm{fail}\ \}.\end{aligned}$$

For the definition of predicates not_in_rangeN/1 and eqplus/0 we make use of the aforementioned loop technique and predicates from program $\pi_<$.

$$\begin{aligned}\pi_{rangeN} = \{\ & \text{undefeated\_upto}(X,Y) \leftarrow \text{inf}(Y), \text{outN}(X), \text{outN}(Y);\\ & \text{undefeated\_upto}(X,Y) \leftarrow \text{inf}(Y), \text{outN}(X), \textit{not}\ \text{defeat}(Y,X);\\ & \text{undefeated\_upto}(X,Y) \leftarrow \text{succ}(Z,Y), \text{undefeated\_upto}(X,Z),\\ & \qquad\qquad \text{outN}(Y);\\ & \text{undefeated\_upto}(X,Y) \leftarrow \text{succ}(Z,Y), \text{undefeated\_upto}(X,Z),\\ & \qquad\qquad \textit{not}\ \text{defeat}(Y,X);\\ & \text{not\_in\_rangeN}(X) \leftarrow \text{sup}(Y), \text{outN}(X), \text{undefeated\_upto}(X,Y);\\ & \text{in\_rangeN}(X) \leftarrow \text{inN}(X);\\ & \text{in\_rangeN}(X) \leftarrow \text{outN}(X), \text{inN}(Y), \text{defeat}(Y,X)\ \}.\end{aligned}$$

$$\begin{aligned}\pi^+_{eq} = \{\ & \text{eqp\_upto}(X) \leftarrow \text{inf}(X), \text{in\_range}(X), \text{in\_rangeN}(X);\\ & \text{eqp\_upto}(X) \leftarrow \text{inf}(X), \text{not\_in\_range}(X), \text{not\_in\_rangeN}(X);\\ & \text{eqp\_upto}(X) \leftarrow \text{succ}(Z,X), \text{in\_range}(X), \text{in\_rangeN}(X), \text{eqp\_upto}(Z);\\ & \text{eqp\_upto}(X) \leftarrow \text{succ}(Y,X), \text{not\_in\_range}(X), \text{not\_in\_rangeN}(X),\\ & \qquad\qquad \text{eqp\_upto}(Y);\\ & \text{eqplus} \leftarrow \text{sup}(X), \text{eqp\_upto}(X)\ \}.\end{aligned}$$

Proposition 2. *For any AF F, $stg(F) \cong \mathcal{AS}(\pi_{stg}(\hat{F}))$, where $\pi_{stg} = \pi_{cf} \cup \pi_< \cup \pi_{range} \cup \pi_{rangeN} \cup \pi^+_{eq} \cup \pi_{satstage}$.*

3.2 Meta ASP Encodings

The following encodings for preferred, semi-stable and stage semantics are written using the *#minimize* statement when evaluated with the subset-minimization semantics provided by metasp. For our encodings we do not need prioritization and weights, therefore these are omitted (i.e. set to default) in the minimization statements. The minimization technique is realized through meta programming techniques, which themselves are answer-set programs. This works as follows: The ASP encoding to solve is given to the grounder gringo which reifies the program, i.e. outputs a ground program consisting of facts, which represent the rules and facts of the original input encoding. The grounder is then again executed on this output with the meta programs which encode the optimization. Finally, claspD computes the answer sets. Note that here we use the version of clasp which supports disjunctive rules. Therefore for a program π and an AF F we have the following execution.

```
gringo --reify π(F̂) | \
   gringo - {meta.lp,metaO.lp,metaD.lp} \
   <(echo "optimize(1,1,incl).") | claspD 0
```

Here, `meta.lp`, `metaO.lp` and `metaD.lp` are the encodings for the minimization statement. The statement `optimize(incl,1,1)` indicates that we use subset inclusion for the optimization technique using priority and weight 1.

We now look at the encodings for the preferred, semi-stable and stage semantics using this minimization technique. First, we need one auxiliary module for admissible extensions.

$$\begin{aligned}\pi_{adm} = \pi_{cf} \cup \{&\text{defeated}(X) \leftarrow \text{in}(Y), \text{defeat}(Y,X);\\ &\leftarrow \text{in}(X), \text{defeat}(Y,X), \textit{not}\,\text{defeated}(Y)\}.\end{aligned}$$

Now the modules for preferred, semi-stable and stage semantics are easy to encode using the minimization statement of `metasp`. For the preferred semantics we take the module π_{adm} and minimize the out/1 predicate. This in turn gives us the subset-maximal admissible extensions which captures the definition of preferred semantics. The encodings for the semi-stable and stage semantics are similar. Here we minimize the predicate not_in_range/1 from the π_{range} module.

$$\begin{aligned}\pi_{prf\_metasp} &= \pi_{adm} \cup \{\#\textit{minimize}[\text{out}]\}.\\ \pi_{sem\_metasp} &= \pi_{adm} \cup \pi_{range} \cup \{\#\textit{minimize}[\text{not\_in\_range}]\}.\\ \pi_{stg\_metasp} &= \pi_{cf} \cup \pi_{range} \cup \{\#\textit{minimize}[\text{not\_in\_range}]\}.\end{aligned}$$

The following results follow now directly.

Proposition 3. *For any AF F, we have*

1. $\textit{prf}(F) \cong \mathcal{AS}(\pi_{prf\_metasp}(\hat{F}))$,
2. $\textit{sem}(F) \cong \mathcal{AS}(\pi_{sem\_metasp}(\hat{F}))$, *and*
3. $\textit{stg}(F) \cong \mathcal{AS}(\pi_{stg\_metasp}(\hat{F}))$.

Next we give two different encodings for computing resolution-based grounded extensions. Both encodings use subset-minimization for the resolution part, i.e. the resulting extension is subset-minimal with respect to all possible resolutions. The difference between the two encodings is that the first one computes the grounded extension for the guessed resolution explicitly (making use of looping concepts presented already in [11]). The second encoding uses the `metasp` subset-minimization also to get the grounded extension from the complete extensions of the current resolution (recall that the grounded extension is in fact the unique subset-minimal complete extension).The module π_{grd} below for computing the grounded extension is taken from [11] with a small modification: instead of the defeat predicate we use defeat_minus_beta, since we need the grounded extensions of a restricted defeat relation. In fact, the π_{res} module guesses this restricted defeat relation $\{R \setminus \beta\}$ for a resolution β.

$$\begin{aligned}\pi_{res} = \{\ &\text{defeat\_minus\_beta}(X,Y) \leftarrow \text{defeat}(X,Y), \textit{not}\,\text{defeat\_minus\_beta}(Y,X),\\ &\qquad\qquad X \neq Y;\\ &\text{defeat\_minus\_beta}(X,Y) \leftarrow \text{defeat}(X,Y), \textit{not}\,\text{defeat}(Y,X);\\ &\text{defeat\_minus\_beta}(X,X) \leftarrow \text{defeat}(X,X)\ \}.\end{aligned}$$

We repeat the definition of π_{grd} here, which includes the module $\pi_{defended}$.

$$\begin{aligned}\pi_{defended} = \{&\text{defended\_upto}(X,Y) \leftarrow \text{inf}(Y), \text{in}(X), \mathit{not}\ \text{defeat\_minus\_beta}(Y,X);\\ &\text{defended\_upto}(X,Y) \leftarrow \text{inf}(Y), \text{in}(Z), \text{defeat\_minus\_beta}(Z,Y),\\ &\qquad\qquad\qquad\qquad \text{defeat\_minus\_beta}(Y,X);\\ &\text{defended\_upto}(X,Y) \leftarrow \text{succ}(Z,Y), \text{defended\_upto}(X,Z),\\ &\qquad\qquad\qquad\qquad \mathit{not}\ \text{defeat\_minus\_beta}(Y,X);\\ &\text{defended\_upto}(X,Y) \leftarrow \text{succ}(Z,Y), \text{in}(V), \text{defeat\_minus\_beta}(V,Y),\\ &\qquad\qquad\qquad\qquad \text{defeat\_minus\_beta}(Y,X);\\ &\text{defended}(X) \leftarrow \text{sup}(Y), \text{defended\_upto}(X,Y)\}.\\ \pi_{grd} \ = \ &\pi_< \cup \pi_{defended} \cup \{\text{in}(X) \leftarrow \text{defended}(X)\}.\end{aligned}$$

Now we can give the first encoding for resolution-based grounded semantics.

$$\pi_{grd^*\_metasp} = \pi_{grd} \cup \pi_{res} \cup \{\#\mathit{minimize}[\text{in}]\}.$$

The second encoding for resolution-based grounded semantics performs the `metasp` subset-minimization from the complete extensions of the current resolution to compute the grounded extension. We again use the restricted defeat relation.

$$\begin{aligned}\pi_{com} = \pi_{adm} \cup \{&\text{undefended}(X) \leftarrow \text{defeat\_minus\_beta}(Y,X), \mathit{not}\ \text{defeated}(Y);\\ &\leftarrow \text{out}(X), \mathit{not}\ \text{undefended}(X)\ \}.\end{aligned}$$

We obtain the following `metasp` encoding:

$$\pi'_{grd^*\_metasp} = \pi_{com} \cup \pi_{res} \cup \{\#\mathit{minimize}[\text{in}]\}.$$

Proposition 4. *For any AF F and $\pi \in \{\pi_{grd^*\_metasp}, \pi'_{grd^*\_metasp}\}$, $grd^*(F)$ corresponds to $\mathcal{AS}(\pi(\hat{F}))$ in the sense of Definition 5, but without property (iii).*

As the proposition suggests there is a caveat for these two encodings of the resolution-based grounded semantics. In general we have that several answer sets map to the same extension, i.e. there is no one-to-one correspondence between answer sets and extensions. The reason for this behavior lies in the guessing of a resolution. Whereas the other encodings guess basically the in/1 predicate, these two `metasp` encodings guess the resolution. Therefore the result might include the same extension with different resolutions guessed. While this does not harm credulous or skeptical reasoning, some measures have to be taken to remove these duplicates when enumerating or counting extensions. The solver `clasp` already features such a technique which is presented in [14]. This feature is not yet implemented in `claspD`. Furthermore the meta encodings for `metasp` use disjunctive ASP, which increases the computational complexity to the second level of the polynomial hierarchy, whereas the problem of resolution based grounded semantics is situated on the first level.

3.3 Alternative Encodings for Resolution-based Grounded Semantics

So far, we have shown two encodings for the resolution-based grounded semantics via optimization programs, i.e. we made use of the *#minimize* statement under the subset-inclusion semantics. From the complexity point of view this is not adequate, since we expressed a problem on the NP-layer (see Table 1) via an encoding which implicitly makes use of disjunction (see Table 2 for the actual complexity of optimization programs). Hence, we provide here an alternative encoding for the resolution-based grounded semantics based on the verification algorithm proposed by Baroni et al. in [1]. This encoding is just a normal program and thus located at the right level of complexity.

We need some further notation. For an AF $F = (A, R)$ and a set $S \subseteq A$ we define $F|_S = ((A \cap S), R \cap (S \times S))$ as the *sub-framework* of F wrt. S; furthermore we also use $F - S$ as a shorthand for $F|_{A \setminus S}$. By $SCCs(F)$, we denote the set of strongly connected components of an AF $F = (A, R)$ which identify the vertices of a maximal strongly connected[4] subgraph of F; $SCCs(F)$ is thus a partition of A. A partial order $\prec_F$ over $SCCs(F) = \{C_1, \ldots, C_n\}$, denoted as $(C_i \prec_F C_j)$ for $i \neq j$, is defined, if $\exists x \in C_i, y \in C_j$ such that there is a directed path from x to y in F.

Definition 6. *A $C \in SCCs(F)$ is* minimal relevant *(in an AF F) iff C is a minimal element of $\prec_F$ and $F|_C$ satisfies the following:*

(a) the attack relation $R(F|_C)$ of F is irreflexive, i.e. $(x, x) \notin R(F|_C)$ for all arguments x;
(b) $R(F|_C)$ is symmetric, i.e. $(x, y) \in R(F|_C) \Leftrightarrow (y, x) \in R(F|_C)$;
(c) the undirected graph *obtained by replacing each (directed) pair $\{(x, y), (y, x)\}$ in $F|_C$ with a single undirected edge $\{x, y\}$ is acyclic.*

The set of minimal relevant SCCs in F is denoted by $MR(F)$.

Proposition 5. ([1]). *Given an AF $F = (A, R)$ such that $(F - S_R^+) \neq (\emptyset, \emptyset)$ and $MR(F - S_R^+) \neq \emptyset$, where $S = grd(F)$, a set $U \subseteq A$ of arguments is* resolution-based grounded *in F, i.e. $U \in grd^*(F)$ iff the following conditions hold:*

(i) $U \cap S_R^+ = S$;
(ii) $(T \cap \Pi_F) \in stb(F|_{\Pi_F})$, where $T = U \setminus S_R^+$, and $\Pi_F = \bigcup_{V \in MR(F - S_R^+)} V$;
(iii) $(T \cap \Pi_F^C) \in grd^*(F|_{\Pi_F^C} - (S_R^+ \cup (T \cap \Pi_F)_R^{\oplus}))$, where T and Π_F are as in (ii) and $\Pi_F^C = A \setminus \Pi_F$.

To illustrate the conditions of Proposition 5, let us have a look at our example.

Example 4. Consider the AF F of Example 1. Let us check whether $U = \{a, d, f\}$ is resolution-based grounded in F, i.e. whether $U \in grd^*(F)$. $S = \{a\}$ is the

[4] A directed graph is called *strongly connected* if there is a directed path from each vertex in the graph to every other vertex of the graph.

grounded extension of F and $S_R^+ = \{a, b\}$, hence the Condition (i) is satisfied. We obtain $T = \{d, f\}$ and $\Pi_F = \{c, d\}$. We observe that $T \cap \Pi_F = \{d\}$ is a stable extension of the AF $F|_{\Pi_F}$; that satisfies Condition (ii). Now we need to check Condition (iii); we first identify the necessary sets: $\Pi_F^C = \{a, b, e, f\}$, $T \cap \Pi_F^C = \{f\}$ and $(T \cap \Pi_F)_R^\oplus = \{c, e\}$. It remains to check $\{f\} \in \mathit{grd}^*(\{f\}, \emptyset)$ which is easy to see. Hence, $U \in \mathit{grd}^*(F)$.

The following encoding is based on the Guess&Check procedure which was also used for the encodings in [11]. After guessing all conflict-free sets with the program π_{cf}, we check whether the conditions of Definition 6 and Proposition 5 hold. Therefore the program π_{arg_set} makes a copy of the actual arguments, defeats and the guessed set to the predicates arg_set/2, defeatN/3 and inU/2. The first variable in these three predicates serves as an identifier for the iteration of the algorithm (this is necessary to handle the recursive nature of Proposition 5). In all following predicates we will use the first variable of each predicate like this. As in some previous encodings in this paper, we use the program $\pi_<$ to obtain an order over the arguments, and we start our computation with the infimum represented by the predicate inf /1.

$$
\begin{aligned}
\pi_{arg\_set} = \{\ & \mathrm{arg\_set}(N, X) \leftarrow \mathrm{arg}(X), \mathrm{inf}(N);\\
& \mathrm{inU}(N, X) \leftarrow \mathrm{in}(X), \mathrm{inf}(N);\\
& \mathrm{defeatN}(N, Y, X) \leftarrow \mathrm{arg\_set}(N, X), \mathrm{arg\_set}(N, Y), \mathrm{defeat}(Y, X)\ \}.
\end{aligned}
$$

We use here the program $\pi_{defendedN}$ (which is a slight variant of the program $\pi_{defended}$) together with the program $\pi_{groundN}$ where we perform a fixed-point computation of the predicate defendedN/2, as in the definition of the characteristic function $\mathcal{F}_F$ in Definition 2. The basic difference here is that now, we use an additional argument N for the iteration step where predicates arg_set/2, defeatN/3 and inS/2 replace arg /1, defeat/2 and in/1.

$$
\begin{aligned}
\pi_{defendedN} = \{\ & \mathrm{def\_uN}(N, X, Y) \leftarrow \mathrm{inf}(Y), \mathrm{arg\_set}(N, X), \mathit{not}\, \mathrm{defeatN}(N, Y, X);\\
& \mathrm{def\_uN}(N, X, Y) \leftarrow \mathrm{inf}(Y), \mathrm{inS}(N, Z), \mathrm{defeatN}(N, Z, Y),\\
& \qquad\qquad\qquad\qquad \mathrm{defeatN}(N, Y, X);\\
& \mathrm{def\_uN}(N, X, Y) \leftarrow \mathrm{succ}(Z, Y), \mathit{not}\, \mathrm{defeatN}(N, Y, X),\\
& \qquad\qquad\qquad\qquad \mathrm{def\_uN}(N, X, Z);\\
& \mathrm{def\_uN}(N, X, Y) \leftarrow \mathrm{succ}(Z, Y), \mathrm{def\_uN}(N, X, Z), \mathrm{inS}(N, V),\\
& \qquad\qquad\qquad\qquad \mathrm{defeatN}(N, V, Y), \mathrm{defeatN}(N, Y, X)\\
& \mathrm{defendedN}(N, X) \leftarrow \mathrm{sup}(Y), \mathrm{def\_uN}(N, X, Y)\ \}.
\end{aligned}
$$

In $\pi_{groundN}$ we then obtain the predicate inS(N, X) which identifies argument X to be in the grounded extension of the iteration N.

$$
\pi_{groundN} = \pi_{cf} \cup \pi_< \cup \pi_{arg\_set} \cup \pi_{defendedN} \cup \{\ \mathrm{inS}(N, X) \leftarrow \mathrm{defendedN}(N, X)\ \}.
$$

The next module $\pi_{F\_minus\_range}$ computes the arguments in $(F - S_R^+)$, represented by the predicate notInSplusN/2, via predicates in_SplusN/2 and u_cap_Splus/2 (for S_R^+ and $U \cap S_R^+$). The two constraints check condition (i) of Proposition 5.

$$
\begin{aligned}
\pi_{F\_minus\_range} = \{\; & \text{in\_SplusN}(N,X) \leftarrow \text{inS}(N,X);\\
& \text{in\_SplusN}(N,X) \leftarrow \text{inS}(N,Y), \text{defeatN}(N,Y,X);\\
& \text{u\_cap\_Splus}(N,X) \leftarrow \text{inU}(N,X), \text{in\_SplusN}(N,X);\\
& \leftarrow \text{u\_cap\_Splus}(N,X), \mathit{not}\, \text{inS}(N,X);\\
& \leftarrow \mathit{not}\, \text{u\_cap\_Splus}(N,X), \text{inS}(N,X);\\
& \text{notInSplusN}(N,X) \leftarrow \text{arg\_set}(N,X), \mathit{not}\, \text{in\_SplusN}(N,X)\; \}.
\end{aligned}
$$

The module π_{MR} computes $\Pi_F = \bigcup_{V \in MR(F-S_R^+)} V$, where mr$(N,X)$ denotes that an argument is contained in a set $V \in MR$. Therefore we need to check all three conditions of Definition 6. The first two rules compute the predicate reach(N,X,Y) if there is a path between the arguments $X, Y \in (F - S_R^+)$. With this predicate we will identify the SCCs. The third rule computes self_defeat/2 for all arguments violating Condition (a). Next we need to check Condition (b). With nsym/2 we obtain those arguments which do not have a symmetric attack to any other argument from the same component. Condition (c) is a bit more tricky. With predicate reachnotvia/4 we say that there is a path from X to Y not going over argument V in the framework $(F - S_R^+)$. With this predicate at hand we can check for cycles with cyc/4. Then, to complete Condition (c) we derive bad/2 for all arguments which are connected to a cycle (or a self-defeating argument). In the predicate pos_mr/2, we put all the three conditions together and say that an argument x is possibly in a set $V \in MR$ if (i) $x \in (F - S_R^+)$, (ii) x is neither connected to a cycle nor self-defeating, and (iii) for all y it holds that $(x,y) \in (F - S_R^+) \Leftrightarrow (y,x) \in (F - S_R^+)$. Finally we only need to check if the SCC obtained with pos_mr/2 is a minimal element of $\prec_F$. Hence we get with notminimal/2 all arguments not fulfilling this, and in the last rule we obtain with mr/2 the arguments contained in a minimal relevant SCC.

$$
\begin{aligned}
\pi_{MR} = \{\; & \text{reach}(N,X,Y) \leftarrow \text{notInSplusN}(N,X), \text{notInSplusN}(N,Y), \text{defeatN}(N,X,Y);\\
& \text{reach}(N,X,Y) \leftarrow \text{notInSplusN}(N,X), \text{defeatN}(N,X,Z), \text{reach}(N,Z,Y),\\
& \qquad X! = Y;\\
& \text{self\_defeat}(N,X) \leftarrow \text{notInSplusN}(N,X), \text{defeatN}(N,X,X);\\
& \text{nsym}(N,X) \leftarrow \text{notInSplusN}(N,X), \text{notInSplusN}(N,Y), \text{defeatN}(N,X,Y),\\
& \qquad \mathit{not}\, \text{defeatN}(N,Y,X), \text{reach}(N,X,Y), \text{reach}(N,Y,X), X! = Y;\\
& \text{nsym}(N,Y) \leftarrow \text{notInSplusN}(N,X), \text{notInSplusN}(N,Y), \text{defeatN}(N,X,Y),\\
& \qquad \mathit{not}\, \text{defeatN}(N,Y,X), \text{reach}(N,X,Y), \text{reach}(N,Y,X), X! = Y;\\
& \text{reachnotvia}(N,X,V,Y) \leftarrow \text{defeatN}(N,X,Y), \text{notInSplusN}(N,V),\\
& \qquad \text{reach}(N,X,Y), \text{reach}(N,Y,X), X! = V, Y! = V;\\
& \text{reachnotvia}(N,X,V,Y) \leftarrow \text{reachnotvia}(N,X,V,Z), \text{reach}(N,X,Y),\\
& \qquad \text{reachnotvia}(N,Z,V,Y), \text{reach}(N,Y,X),\\
& \qquad Z! = V, X! = V, Y! = V;\\
& \text{cyc}(N,X,Y,Z) \leftarrow \text{defeatN}(N,X,Y), \text{defeatN}(N,Y,X),\\
& \qquad \text{defeatN}(N,Y,Z), \text{defeatN}(N,Z,Y),
\end{aligned}
$$

$$
\begin{aligned}
&\qquad\qquad \text{reachnotvia}(N, X, Y, Z), X! = Y, Y! = Z, X! = Z;\\
&\text{bad}(N, Y) \leftarrow \text{cyc}(N, X, U, V), \text{reach}(N, X, Y), \text{reach}(N, Y, X);\\
&\text{bad}(N, Y) \leftarrow \text{self\_defeat}(N, X), \text{reach}(N, X, Y), \text{reach}(N, Y, X);\\
&\text{bad}(N, Y) \leftarrow \text{nsym}(N, X), \text{reach}(N, X, Y), \text{reach}(N, Y, X);\\
&\text{pos\_mr}(N, X) \leftarrow \text{notInSplusN}(N, X), \textit{not}\,\text{bad}(N, X), \textit{not}\,\text{self\_defeat}(N, X),\\
&\qquad\qquad \textit{not}\,\text{nsym}(N, X);\\
&\text{notminimal}(N, Z) \leftarrow \text{reach}(N, X, Y), \text{reach}(N, Y, X),\\
&\qquad\qquad \text{reach}(N, X, Z), \textit{not}\,\text{reach}(N, Z, X);\\
&\text{mr}(N, X) \leftarrow \text{pos\_mr}(N, X), \textit{not}\,\text{notminimal}(N, X)\ \}.
\end{aligned}
$$

We now turn to Condition (ii) of Proposition 5, where the first rule in $\pi_{stableN}$ computes the set $T = U \backslash S_R^+$. Then we check whether $T = \emptyset$ and $MR(F - S_R^+) = \emptyset$ via predicates emptyT/1 and not_exists_mr/1. If this is so, we terminate the iteration in the last module $\pi_{iterate}$. The first constraint eliminates those guesses where $MR(F - S_R^+) = \emptyset$ but $T \neq \emptyset$, because the algorithm is only defined for AFs fulfilling this. Finally we derive the arguments which are defeated by the set T in the MR denoted by defeated/2, and with the last constraint we eliminate those guesses where there is an argument not contained in T and not defeated by T in MR and hence $(T \cap \Pi_F) \notin stb(F|_{\Pi_F})$.

$$
\begin{aligned}
\pi_{stableN} = \{\ &\text{t}(N, X) \leftarrow \text{inU}(N, X), \textit{not}\,\text{inS}(N, X);\\
&\text{nemptyT}(N) \leftarrow \text{t}(N, X);\\
&\text{emptyT}(N) \leftarrow \textit{not}\,\text{nemptyT}(N), \text{arg\_set}(N, X);\\
&\text{existsMR}(N) \leftarrow \text{mr}(N, X), \text{notInSplusN}(N, X);\\
&\text{not\_exists\_mr}(N) \leftarrow \textit{not}\,\text{existsMR}(N), \text{notInSplusN}(N, X);\\
&\text{true}(N) \leftarrow \text{emptyT}(N), \textit{not}\,\text{existsMR}(N);\\
&\leftarrow \text{not\_exists\_mr}(N), \text{nemptyT}(N);\\
&\text{defeated}(N, X) \leftarrow \text{mr}(N, X), \text{mr}(N, Y), \text{t}(N, Y), \text{defeatN}(N, Y, X);\\
&\leftarrow \textit{not}\,\text{t}(N, X), \textit{not}\,\text{defeated}(N, X), \text{mr}(N, X)\ \}.
\end{aligned}
$$

With the last module $\pi_{iterate}$ we perform Step (iii) of Proposition 5. The predicate t_mrOplus/2 computes the set $(T \cap \Pi_F)_R^{\oplus}$ and with the second rule we start the next iteration for the AF $(F|_{\Pi_F^C} - (S_R^+ \cup (T \cap \Pi_F)_R^{\oplus}))$ and the set $(T \cap \Pi_F^C)$.

$$
\begin{aligned}
\pi_{iterate} = \{\ &\text{t\_mrOplus}(N, Y) \leftarrow \text{t}(N, X), \text{mr}(N, X), \text{defeatN}(N, X, Y);\\
&\text{arg\_set}(M, X) \leftarrow \text{notInSplusN}(N, X), \textit{not}\,\text{mr}(N, X),\\
&\qquad\qquad \textit{not}\,\text{t\_mrOplus}(N, X), \text{succ}(N, M), \textit{not}\,\text{true}(N);\\
&\text{inU}(M, X) \leftarrow \text{t}(N, X), \textit{not}\,\text{mr}(N, X), \text{succ}(N, M), \textit{not}\,\text{true}(N)\ \}.
\end{aligned}
$$

Finally we put everything together and obtain the program π_{grd^*}.

$$\pi_{grd^*} = \pi_{groundN} \cup \pi_{F\_minus\_range} \cup \pi_{MR} \cup \pi_{stableN} \cup \pi_{iterate}.$$

Proposition 6. *For any AF F, $grd^*(F) \cong \mathcal{AS}(\pi_{grd^*}(\hat{F}))$.*

4 Experimental Evaluation

In this section we present our results of the performance evaluation. We compared the time needed for computing all extensions for the semantics described earlier, using both the handcraft saturation-based and the alternative `metasp` encodings.

The tests were executed on an openSUSE based machine with eight Intel Xeon processors (2.33 GHz) and 49 GB memory. For computing the answer sets, we used `gringo` (version 3.0.3) for grounding and the solver `claspD` (version 1.1.1). The latter being the variant for disjunctive answer-set programs.

We randomly generated AFs (i.e. graphs) ranging from 20 to 110 arguments. We used two parametrized methods for generating the attack relation. The first generates arbitrary AFs and inserts for any pair (a, b) the attack from a to b with a given probability p. The other method generates AFs with an $n \times m$ grid-structure. We consider two different neighborhoods, one connecting arguments vertically and horizontally and one that additionally connects the arguments diagonally. Such a connection is a mutual attack with a given probability p and in only one direction otherwise. The probability p was chosen between 0.1 and 0.4.

Overall 14388 tests were executed, with a timeout of five minutes for each execution. Timed out instances are considered as solved in 300 seconds. The time consumption was measured using the Linux `time` command. For all the tests we let the solver generate all answer sets, but only outputting the number of models. To minimize external influences on the test runs, we alternated the different encodings during the tests.

Figures 1, 2 and 3 depict the results for the preferred, semi-stable and stage semantics respectively. The figures show the average computation time for both

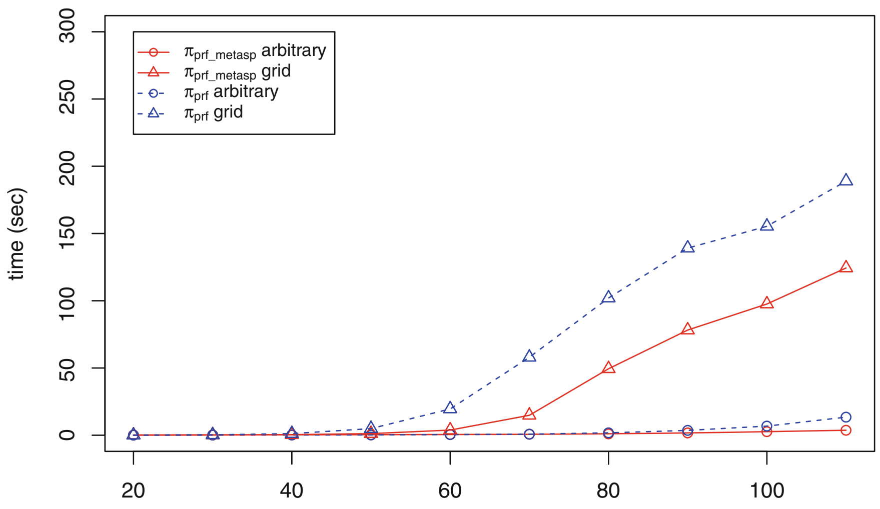


Fig. 1. Average computation time for preferred semantics.

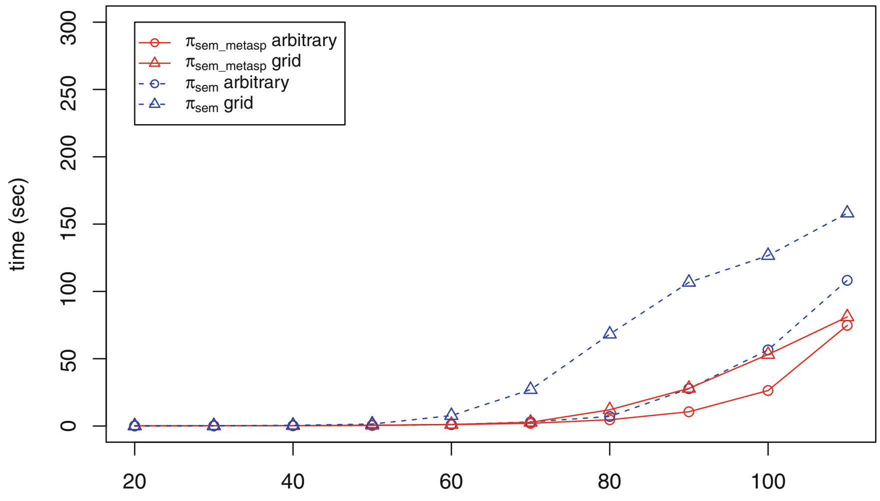


Fig. 2. Average computation time for semi-stable semantics.

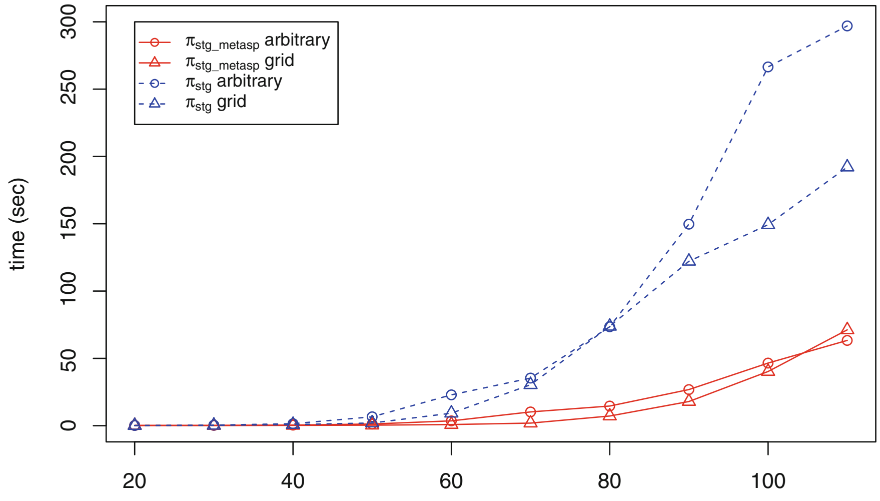


Fig. 3. Average computation time for stage semantics.

the handcraft and the `metasp` encoding for a certain number of arguments. We distinguish here between arbitrary, i.e. completely random AFs and grid structured ones. One can see that the `metasp` encodings have a better performance, compared to the handcraft encodings. In particular, for the stage semantics the performance difference is noticeable. Recall that the average computation time includes the timeouts, which strongly influence the diagrams.

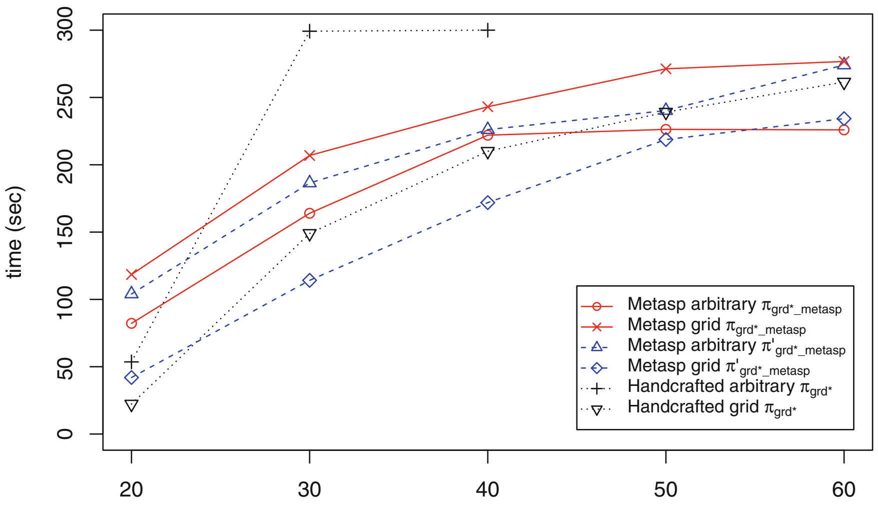


Fig. 4. Average computation time for resolution-based grounded semantics.

For the resolution-based grounded semantics, Fig. 4 shows again the average computation time needed for a certain number of arguments. Let us first consider the case of arbitrary AFs. The handcraft encoding struggled with AFs of size 40 or larger. Many of those instances could not be solved due to memory faults. This is indicated by the missing data points. Both `metasp` encodings performed better overall, but still many timeouts were encountered. If we look more closely at the structured AFs then we see that $\pi'_{grd^*\_metasp}$ performs better overall than the other `metasp` variant. Interestingly, computing the grounded part with a handcraft encoding without a Guess&Check part did not result in a lower computation time on average. The handcraft encoding performed better than $\pi_{grd^*\_metasp}$ on grids.

One reason for the performance problems of the handcraft encoding lies in the relatively high arity of some predicates. The encoding uses predicates with up to four variables, in contrast to the encoding for e.g. the stage semantics which needs only predicates with up to three variables. This can increase the time needed for grounding drastically. On the other side, the `metasp` encodings, as mentioned in Proposition 4, suffer from the fact that the answer sets are not in a one-to-one correspondence to the solutions, i.e. several answer sets may represent the same extension.

Overall the `metasp` encodings outperform the direct encodings. This is partially due to the fact that the former utilize encodings tailored to the `gringo/claspD` package.

5 Conclusion

In this paper, we inspected various ASP encodings for four prominent semantics in the area of abstract argumentation. (1) For the preferred and the semi-stable semantics, we compared existing saturation-based encodings [11] (here we called them handcrafted encodings) with novel alternative encodings which are based on the recently developed `metasp` approach [13], where subset-minimization can be directly specified and a front-end (i.e. a meta-interpreter) compiles such statements back into the core ASP language. (2) For the stage semantics, we presented here both a handcrafted and a `metasp` encoding. Finally, (3) for the resolution-based grounded semantics we provided three novel encodings, two of them using the `metasp` techniques.

While with some performance optimization techniques for ASP the readability of the encodings change for the worse, the `metasp` encodings are shorter than the handcrafted saturation encodings. Furthermore, they are much simpler to design (since saturation techniques are delegated to the meta-interpreter), and they perform surprisingly well when compared with the handcrafted encodings which are directly given to the ASP solver. This shows the practical relevance of the `metasp` technique also in the area of abstract argumentation. Future work will be to investigate performance improvements of other optimization features like aggregates, which are provided by most of the prominent ASP solvers.

References

1. Baroni, P., Dunne, P.E., Giacomin, M.: On the resolution-based family of abstract argumentation semantics and its grounded instance. Artif. Intell. **175**(3–4), 791–813 (2011)
2. Baroni, P., Giacomin, M.: Semantics of abstract argument systems. In: Rahwan, I., Simari, G.R. (eds.) Argumentation in Artificial Intelligence, pp. 25–44. Springer, Berlin (2009)
3. Bench-Capon, T.J.M., Dunne, P.E.: Argumentation in artificial intelligence. Artif. Intell. **171**(10–15), 619–641 (2007)
4. Caminada, M.: Semi-stable semantics. In: Proceedings of COMMA 2006, pp. 121–130 (2006)
5. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. **33**(3), 374–425 (2001)
6. Dimopoulos, Y., Torres, A.: Graph theoretical structures in logic programs and default theories. Theor. Comput. Sci. **170**(1–2), 209–244 (1996)
7. Dung, P.M.: On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. Artif. Intell. **77**(2), 321–358 (1995)
8. Dunne, P.E., Bench-Capon, T.J.M.: Coherence in finite argument systems. Artif. Intell. **141**(1/2), 187–203 (2002)
9. Dunne, P.E., Caminada, M.: Computational complexity of semi-stable semantics in abstract argumentation frameworks. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 153–165. Springer, Heidelberg (2008)

10. Dvořák, W., Woltran, S.: Complexity of semi-stable and stage semantics in argumentation frameworks. Inf. Process. Lett. **110**(11), 425–430 (2010)
11. Egly, U., Gaggl, S.A., Woltran, S.: Answer-set programming encodings for argumentation frameworks. Argument Comput. **1**(2), 147–177 (2010)
12. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: propositional case. Ann. Math. Artif. Intell. **15**(3–4), 289–323 (1995)
13. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theory Pract. Logic Program. **11**(4–5), 821–839 (2011)
14. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 71–86. Springer, Heidelberg (2009)
15. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**(3/4), 365–386 (1991)
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7**(3), 499–562 (2006)
17. Toni, F., Sergot, M.: Argumentation and answer set programming. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. LNCS (LNAI), vol. 6565, pp. 164–180. Springer, Heidelberg (2011)
18. Verheij, B.: Two approaches to dialectical argumentation: admissible sets and argumentation stages. In: Proc. NAIC'96, pp. 357–368 (1996)

Confidentiality-Preserving Publishing of EDPs for Credulous and Skeptical Users

Katsumi Inoue[1], Chiaki Sakama[2], and Lena Wiese[3](✉)

[1] National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ki@nii.ac.jp

[2] Department of Computer and Communication Sciences, Wakayama University,
930 Sakaedani, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

[3] Institute of Computer Science, Georg-August-Universität Göttingen,
Goldschmidtstr. 7, 37077 Göttingen, Germany
wiese@cs.uni-goettingen.de

Abstract. Publishing private data on external servers incurs the problem of how to avoid unwanted disclosure of confidential data. We study the problem of confidentiality-preservation when publishing extended disjunctive logic programs and show how it can be solved by extended abduction. In particular, we analyze how the differences between users who employ either credulous or skeptical non-monotonic reasoning affect confidentiality.

Keywords: Data publishing · Confidentiality · Privacy · Extended abduction · Answer set programming · Negation as failure · Non-monotonic reasoning

1 Introduction

Confidentiality of data (also called privacy or secrecy in some contexts) is a major security goal. Releasing data to a querying user without disclosing confidential information has long been investigated in areas like access control, k-anonymity, inference control, and data fragmentation. Such approaches prevent disclosure according to some security policy by restricting data access (denial, refusal), by modifying some data (perturbation, noise addition, cover stories, lying, weakening), or by breaking sensitive associations (fragmentation). Several approaches (like [2,3,8,14–16]) employ logic-based mechanisms to ensure data confidentiality. In particular, [5] uses brave reasoning in default logic theories to solve a privacy problem in a classical database (a set of ground facts). For a non-classical knowledge base (where negation as failure *not* is allowed) [17] studies correctness of access rights. Confidentiality of predicates in collaborative multi-agent abduction is a topic in [11].

Lena Wiese was partially supported by a postdoctoral research grant of the German Academic Exchange Service (DAAD) while preparing this work.

In this article we analyze **confidentiality-preserving data publishing** in a knowledge base setting: data as well as integrity constraints or deduction rules are represented as logical formulas. If such a knowledge base is released to the public for general querying (e.g., microcensus data) or outsourced to a storage provider (e.g., database-as-a-service in cloud computing), confidential data could be disclosed. This article is a revised and extended version of [10]; in particular, we extend [10] to also cover confidentiality-preserving data publishing for users who deduce information by *skeptical* non-monotonic reasoning. This article is one of only few papers (see [11,12,17]) covering confidentiality for logic programs. This formalism however has relevance in multi-agent communications where agent knowledge is modeled by logic programs. In our settings (as already in [10]), knowledge bases come in the form of extended disjunctive logic programs (EDPs) as defined below. Hence, with this formalism we achieve high expressiveness by allowing negation as failure *not* as well as disjunctions in rule heads. In this article, we assume that users accessing the published knowledge base use either credulous or skeptical reasoning to retrieve data from it; users also possess some invariant "a priori knowledge" that can be applied to these data to deduce further information – again by using either credulous or skeptical reasoning. On the knowledge base side, a confidentiality policy specifies which is the confidential information that must never be disclosed.

With **extended abduction** [13] we obtain a "secure version" of the knowledge base that can safely be published even when a priori knowledge is applied. In this article, we show how confidentiality-preservation for skeptical users differs from the one for credulous users. More precisely, while computing the secure version for a *credulous* user corresponds to finding a *skeptical* anti-explanation for all the elements of the confidentiality policy, computing the secure version for a *skeptical* user corresponds to finding a *credulous* anti-explanation for the elements of the confidentiality policy followed by an additional consistency check. Extended abduction has been used in different applications like for example providing a logical framework for dishonest reasoning [12]. It can be solved by computing the answer sets of an update program (see [13]); thus an implementation of extended abduction can profit from current answer set programming (ASP) solvers [4]. To retrieve the confidentiality-preserving knowledge base K^{pub} from the input knowledge base K, the a priori knowledge *prior* and the confidentiality policy *policy*, a sequence of transformations are applied; the overall approach is depicted in Fig. 1.

In summary, this paper makes the following contributions:

– it formalizes confidentiality-preserving data publishing for users who retrieve data under either a credulous or a skeptical query response semantics.
– it devises a procedure to securely publish a logic program (with an expressiveness up to extended disjunctive logic programs) respecting a subset-minimal change semantics.
– it shows that confidentiality-preservation for credulous as well as skeptical users corresponds to finding anti-explanations and can be solved by extended abduction.

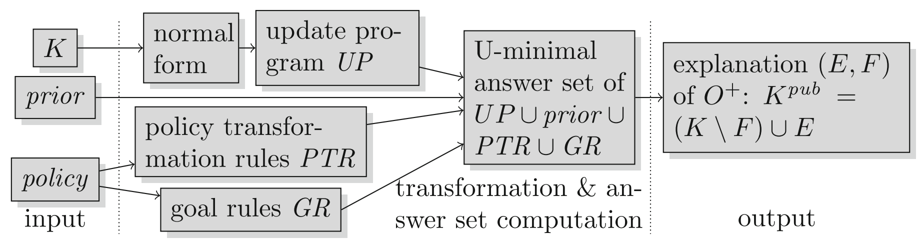

K
prior
policy
input
normal form
update program UP
policy transformation rules PTR
goal rules GR
U-minimal answer set of $UP \cup prior \cup PTR \cup GR$
transformation & answer set computation
explanation (E, F) of O^+: $K^{pub} = (K \setminus F) \cup E$
output

Fig. 1. Finding a confidentiality-preserving K^{pub}

In the remainder of this article, Sect. 2 provides background on extended disjunctive logic programs and answer set semantics; Sect. 3 defines the problem of confidentiality in data publishing; Sect. 4 recalls extended abduction and update programs; Sect. 5 shows how answer sets of update programs correspond to confidentiality-preserving knowledge bases; and Sect. 6 gives some discussion and concluding remarks.

2 EDPs and Answer Set Semantics

In this article, a knowledge base K is represented by an *extended disjunctive logic program* (EDP) – a set of formulas called *rules* of the form:

$$L_1; \ldots; L_l \leftarrow L_{l+1}, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n. \quad (n \geq m \geq l \geq 0)$$

A rule contains literals L_i, disjunction ";", conjunction ",", negation as failure "*not* ", and implication "$\leftarrow$". A literal is a first-order atom or an atom preceded by classical negation "$\neg$". *not* L is called a *NAF-literal.* The disjunction left of the implication $\leftarrow$ is called the *head*, while the conjunction right of $\leftarrow$ is called the *body* of the rule. For a rule R, we write $\mathit{head}(R)$ to denote the set of literals $\{L_1, \ldots, L_l\}$ and $\mathit{body}(R)$ to denote the set of (NAF-)literals $\{L_{l+1}, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n\}$. Rules consisting only of a singleton head $L \leftarrow$ are identified with the literal L and used interchangeably. An EDP is ground if it contains no variables. If an EDP contains variables, it is identified with the set of its ground instantiations: the elements of its Herbrand universe are substituted in for the variables in all possible ways. We assume that the language contains no function symbols, so that each rule with variables represents a finite set of ground rules. For a program K , we denote $\mathscr{L}_K$ the set of ground literals in the language of K . Note that EDPs offer a high expressiveness including disjunctive and non-monotonic reasoning.

Example 1. In a medical knowledge base $\mathit{Ill}(x, y)$ states that a patient x is ill with disease y; $\mathit{Treat}(x, y)$ states that x is treated with medicine y. Assume that if you read the record and find that one treatment ($\mathsf{Medi1}$) is recorded and another one ($\mathsf{Medi2}$) is not recorded, then you know that the patient is at least ill with Aids or Flu (and possibly has other illnesses).

$K = \{Ill(x, \mathsf{Aids}); Ill(x, \mathsf{Flu}) \leftarrow Treat(x, \mathsf{Medi1}), not\ Treat(x, \mathsf{Medi2}).\ ,$
$Ill(\mathsf{Mary}, \mathsf{Aids}).\ , \ Treat(\mathsf{Pete}, \mathsf{Medi1}).\}$ serves as a running example.

The semantics of K can be given by the answer set semantics [7]: A set $S \subseteq \mathscr{L}_K$ of ground literals *satisfies* a ground literal L if $L \in S$; S satisfies a conjunction if it satisfies every conjunct; S satisfies a disjunction if it satisfies at least one disjunct; S satisfies a ground rule if whenever the body literals are contained in S ($\{L_{l+1}, \dots, L_m\} \subseteq S$) and all NAF-literals are not contained in S ($\{L_{m+1}, \dots, L_n\} \cap S = \emptyset$), then at least one head literal is contained in S ($L_i \in S$ for an i such that $1 \leq i \leq l$). If an EDP K contains no NAF-literals ($m = n$), then such a set S is an *answer set* of K if S is a subset-minimal set such that

1. S satisfies every rule from the ground instantiation of K.
2. If S contains a pair of complementary literals L and $\neg L$, then $S = \mathscr{L}_K$.

This definition of an answer set can be extended to full EDPs (containing NAF-literals) as in [13]: For an EDP K and a set of ground literals $S \subseteq \mathscr{L}_K$, K can be transformed into a NAF-free program K^S as follows. For every ground rule from the ground instantiation of K (with respect to its Herbrand universe), the rule $L_1; \dots; L_l \leftarrow L_{l+1}, \dots, L_m$ is in K^S if $\{L_{m+1}, \dots, L_n\} \cap S = \emptyset$. Then, S is an answer set of K if S is an answer set of K^S. An answer set is *consistent* if it is not $\mathscr{L}_K$. A program K is *consistent* if it has a consistent answer set; otherwise K is *inconsistent*.

Example 2. The example K has the following two consistent answer sets

$$S_1 = \{Ill(\mathsf{Mary}, \mathsf{Aids}),\ Treat(\mathsf{Pete}, \mathsf{Medi1}),\ Ill(\mathsf{Pete}, \mathsf{Aids})\},$$
$$S_2 = \{Ill(\mathsf{Mary}, \mathsf{Aids}),\ Treat(\mathsf{Pete}, \mathsf{Medi1}),\ Ill(\mathsf{Pete}, \mathsf{Flu})\}.$$

When adding the negative fact $\neg Ill(\mathsf{Pete}, \mathsf{Flu})$ to K, there is just one consistent answer set left: for $K' := K \cup \{\neg Ill(\mathsf{Pete}, \mathsf{Flu}).\}$ the only answer set is

$$S' = \{Ill(\mathsf{Mary}, \mathsf{Aids}), \neg Ill(\mathsf{Pete}, \mathsf{Flu}),\ Treat(\mathsf{Pete}, \mathsf{Medi1}),\ Ill(\mathsf{Pete}, \mathsf{Aids})\}.$$

If a rule R is satisfied in *every* answer set of K, we write $K \models R$. In particular, $K \models L$ if a literal L is included in every answer set of K.

3 Confidentiality-Preserving Knowledge Bases

When publishing a knowledge base K^{pub} while preserving confidentiality of some data in the original knowledge base K we do this according to

- the query response semantics that a user querying K^{pub} applies
- a confidentiality policy (denoted *policy*) describing confidential information that should not be released to the public
- background (a priori) knowledge (denoted *prior*) that a user can combine with query responses from the published knowledge base

First we define the credulous and the skeptical query response semantics: in the credulous case, a ground formula Q is *true* in $\mathcal{K}$, if Q is satisfied in *some* answer set of $\mathcal{K}$ – that is, there might be answer sets that do not satisfy Q; in the skeptical case, a ground formula Q is *true* in $\mathcal{K}$, if Q is satisfied in *every* answer set of $\mathcal{K}$. If a rule Q is non-ground and contains some free variables, the response of $\mathcal{K}$ is the set of ground instantiations of Q that are *true* in $\mathcal{K}$ under either the credulous or skeptical semantics.

Definition 1 (Credulous and skeptical query response semantics). *Let U be the Herbrand universe of a consistent knowledge base $\mathcal{K}$. The* credulous query responses *of formula $Q(X)$ (with a vector X of free variables) in $\mathcal{K}$ are*

$$cred(\mathcal{K}, Q(X)) = \{Q(A) \mid A \textit{ is a vector of elements of } U \textit{ and there is an answer set of } \mathcal{K} \textit{ that satisfies } Q(A)\}$$

In particular, for a ground formula Q,

$$cred(\mathcal{K}, Q) = \begin{cases} \{Q\} & \textit{if } \mathcal{K} \textit{ has an answer set that satisfies } Q \\ \emptyset & \textit{otherwise} \end{cases}$$

The skeptical query responses of $Q(X)$ in $\mathcal{K}$ are

$$skep(\mathcal{K}, Q(X)) = \{Q(A) \mid A \textit{ is a vector of elements of } U \textit{ and } \mathcal{K} \models Q(A)\}$$

In particular, for a ground formula Q, $skep(\mathcal{K}, Q) = \begin{cases} \{Q\} & \textit{if } \mathcal{K} \models Q \\ \emptyset & \textit{otherwise} \end{cases}$

Example 3. Assume that the example $\mathcal{K}$ is queried for all patients x suffering from aids. Then, $cred(\mathcal{K}, \mathit{Ill}(x, \mathsf{Aids})) = \{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})\}$ and $skep(\mathcal{K}, \mathit{Ill}(x, \mathsf{Aids})) = \{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})\}$.

It is usually assumed that in addition to the query responses a user has some additional knowledge that he can apply to the query responses. Hence, we additionally assume given a set of rules as some *invariant* **a priori knowledge** prior; invariance is a common assumption (see [6]). We assume that prior is a consistent EDP. Thus, the a priori knowledge may consist of additional facts that the user assumes to hold in $\mathcal{K}$, or some rules that the user can apply to data in $\mathcal{K}$ to deduce new information.

A **confidentiality policy** policy specifies confidential information. We assume that policy contains conjunctions of literals or NAF-literals. We do not only have to avoid that the published knowledge base contains confidential information but also prevent the user from deducing confidential information with the help of his a priori knowledge; this is known as the inference problem [2,6].

Example 4. If we wish to declare the disease aids as confidential for any patient x we can do this with $\mathit{policy} = \{\mathit{Ill}(x, \mathsf{Aids}).\}$. A user querying $\mathcal{K}^{pub}$ might know that a person suffering from flu is not able to work. Hence $\mathit{prior} = \{\neg \mathit{AbleToWork}(x) \leftarrow \mathit{Ill}(x, \mathsf{Flu}).\}$. If we wish to also declare a lack of work ability as confidential, we can add this to the confidentiality policy: $\mathit{policy}' = \{\mathit{Ill}(x, \mathsf{Aids}).\ ,\ \neg \mathit{AbleToWork}(x).\}$.

Next, we establish a definition of confidentiality-preservation that allows for the answer set semantics as an inference mechanism and respects the credulous or skeptical query response semantics: when treating elements of the confidentiality policy as queries, the credulous or skeptical responses must be empty.

Definition 2 (Confidentiality-preservation for credulous and skeptical users). *A knowledge base K^{pub}* preserves confidentiality *of a given confidentiality policy under the* credulous *query response semantics and with respect to a given a priori knowledge prior, if for every conjunction $C(X)$ in the policy, the credulous query responses of $C(X)$ in $K^{pub} \cup \mathit{prior}$ are empty: $\mathit{cred}(K^{pub} \cup \mathit{prior}, C(X)) = \emptyset$. It* preserves confidentiality *under the* skeptical *query response semantics, if the skeptical query responses of $C(X)$ in $K^{pub} \cup \mathit{prior}$ are empty: $\mathit{skep}(K^{pub} \cup \mathit{prior}, C(X)) = \emptyset$.*

Note that the Herbrand universe of $K^{pub} \cup \mathit{prior}$ is applied in the query response semantics; hence, free variables in policy elements $C(X)$ are instantiated according to this universe. Moreover, $K^{pub} \cup \mathit{prior}$ must be consistent.

A goal secondary to confidentiality-preservation is **minimal change**: We want to publish as many data as possible and want to modify these data as little as possible. Different notions of minimal change are used in the literature (see for example [1] for a collection of minimal change semantics in a data integration setting). We apply a subset-minimal change semantics: we choose a K^{pub} that differs from K only subset-minimally. In other words, there is no other confidentiality-preserving knowledge base $K^{pub'}$ which inserts (or deletes) less rules to (from) K than K^{pub}.

Definition 3 (Subset-minimal change). *A confidentiality-preserving knowledge base K^{pub}* subset-minimally changes *K (or is* minimal*, for short) if there is no confidentiality-preserving knowledge base $K^{pub'}$ such that $((K \setminus K^{pub'}) \cup (K^{pub'} \setminus K)) \subset ((K \setminus K^{pub}) \cup (K^{pub} \setminus K))$.*

Example 5. For the example K and *policy* and no a priori knowledge, the fact $\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})$ has to be deleted under both the credulous and the skeptical query response semantics. Moreover, $\mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})$ can be deduced credulously, because it is satisfied by answer set S_1. In order to avoid this, we have two options when only deletions are used: delete $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1})$, or delete the non-literal rule in K; if insertions of literals are allowed, we have three options: insert $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi2})$, or insert $\mathit{Ill}(\mathsf{Pete}, \mathsf{Flu})$, or insert $\neg\mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})$. Each of these options blocks the credulous deduction of $\mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})$. In contrast, for K, *policy*$'$ and *prior*, the last two options (insert $\mathit{Ill}(\mathsf{Pete}, \mathsf{Flu})$, or insert $\neg\mathit{Ill}(\mathsf{Pete}, \mathsf{Aids})$) are not possible, because then the secret $\neg\mathit{AbleToWork}(\mathit{Pete})$ could be deduced credulously. The same two options are impossible for K' (defined in Sect. 2) and *policy* because $\neg\mathit{Ill}(\mathsf{Pete}, \mathsf{Flu})$ is contained in K'.

In the following sections we obtain a minimal solution K^{pub} for a given input K, *prior* and *policy* by transforming the input into a problem of *extended abduction* and solving it with an appropriate update program.

4 Extended Abduction

Traditionally, given a knowledge base K and an observation formula O, *abduction* finds a "(positive) explanation" E – a set of hypothesis formulas – such that every answer set of the knowledge base and the explanation together satisfy the observation; that is, $K \cup E \models O$. Going beyond that [9,13] use *extended* abduction with the notions of "negative observations", "negative explanations" F and "anti-explanations". An abduction problem in general can be restricted by specifying a designated set $\mathcal{A}$ of *abducibles*. This set poses syntactical restrictions on the explanation sets E and F. In particular, positive explanations are characterized by $E \subseteq \mathcal{A} \setminus K$ and negative explanations by $F \subseteq K \cap \mathcal{A}$. If $\mathcal{A}$ contains a formula with variables, it is meant as a shorthand for all ground instantiations of the formula. In this sense, an EDP K accompanied by an EDP $\mathcal{A}$ is called an *abductive program* written as $\langle K, \mathcal{A}\rangle$. The aim of extended abduction is then to find (anti-)explanations as follows:

– given a *positive* observation O, find a pair (E, F) where E is a positive explanation and F is a negative explanation such that
  1. [**explanation**]
     (a) [**skeptical**] O is satisfied in *every* answer set of $(K \setminus F) \cup E$; that is, $(K \setminus F) \cup E \models O$
     (b) [**credulous**] O is satisfied in *some* answer set of $(K \setminus F) \cup E$
  2. [**consistency**] $(K \setminus F) \cup E$ is consistent
  3. [**abducibility**] $E \subseteq \mathcal{A} \setminus K$ and $F \subseteq \mathcal{A} \cap K$
– given a *negative* observation O, find a pair (E, F) where E is a positive anti-explanation and F is a negative anti-explanation such that
  1. [**anti-explanation**]
     (a) [**skeptical**] *no* answer set of $(K \setminus F) \cup E$ satisfies O
     (b) [**credulous**] there is *some* answer set of $(K \setminus F) \cup E$ that does not satisfy O; that is, $(K \setminus F) \cup E \not\models O$
  2. [**consistency**] $(K \setminus F) \cup E$ is consistent
  3. [**abducibility**] $E \subseteq \mathcal{A} \setminus K$ and $F \subseteq \mathcal{A} \cap K$

Among (anti-)explanations, **minimal** (anti-)explanations characterize a subset-minimal alteration of the program K: an (anti-)explanation (E, F) of an observation O is called minimal if for any (anti-)explanation (E', F') of O, $E' \subseteq E$ and $F' \subseteq F$ imply $E' = E$ and $F' = F$.

For an abductive program $\langle K, \mathcal{A}\rangle$ both K and $\mathcal{A}$ are semantically identified with their ground instantiations with respect to the Herbrand universe, so that set operations over them are defined on the ground instances. Thus, when (E, F) contain formulas with variables, $(K \setminus F) \cup E$ means deleting every instance of formulas in F, and inserting any instance of formulas in E from/into K. When E contains formulas with variables, the set inclusion $E' \subseteq E$ is defined for any set E' of instances of formulas in E. Generally, given sets S and T of literals/rules containing variables, any set operation $\circ$ is defined as $S \circ T = \mathit{inst}(S) \circ \mathit{inst}(T)$ where $\mathit{inst}(S)$ is the ground instantiation of S. For example, when $p(x) \in T$, for any constant a occurring in T, it holds that $\{p(a)\} \subseteq T$, $\{p(a)\} \setminus T = \emptyset$, and $T \setminus \{p(a)\} = (T \setminus \{p(x)\}) \cup \{p(y) \mid y \neq a\}$, etc. Moreover, any literal/rule in a set is identified with its variants modulo variable renaming.

4.1 Extended Abduction and Confidentiality-Preservation

Now, the formal correspondence between confidentiality-preservation and extended abduction can be stated as follows. A confidentiality-preserving knowledge base K^{pub} can be obtained by deleting elements from the knowledge base K and by inserting rules that are made up of predicate symbols and constants occuring in $K \cup \mathit{prior}$; however, as we assume prior to be invariant, we cannot delete rules contained in prior . This is summarized in the following theorem.

Theorem 1. *Given a knowledge base K, prior and policy , $K^{pub} = (K \setminus F) \cup E$ is a (minimal) solution of confidentiality-preservation for credulous (resp. skeptical) users iff (E, F) is a (minimal) skeptical (resp. credulous) anti-explanation for every $C_i \in \mathit{policy}$ in the abductive program $\langle K \cup \mathit{prior} , \mathcal{A}_{K \cup \mathit{prior}} \setminus \mathit{prior} \rangle$ where $\mathcal{A}_{K \cup \mathit{prior}}$ is the set of all ground rules constructed in the language of $K \cup \mathit{prior}$.*

Proof. By Definition 1 we have that $\mathit{cred}((K \setminus F) \cup E, C_i) = \emptyset$ iff *no* answer set of $(K \setminus F) \cup E$ satisfies C_i (that is, (E, F) is a skeptical anti-explanation). Respectively, $\mathit{skep}((K \setminus F) \cup E, C_i) = \emptyset$ iff $(K \setminus F) \cup E \not\models C_i$ (that is, (E, F) is a credulous anti-explanation).

4.2 Normal Form

Although extended abduction can handle the very general format of EDPs, some syntactic transformations are helpful. Based on [13] we will briefly describe how a semantically equivalent normal form of an abductive program $\langle K , \mathcal{A} \rangle$ is obtained; in the end, we obtain an equivalent abductive program with only literals as abducibles (instead of general rules). This makes an automatic handling of abductive programs easier; for example, abductive programs in normal form can be easily transformed into update programs as described in Sect. 4.3. The main step is that rules in $\mathcal{A}$ can be mapped to atoms by a naming function n. Let $\mathcal{R}_{\mathcal{A}}$ be the set of abducible *rules*:

$$\mathcal{R}_{\mathcal{A}} = \{\Sigma \leftarrow \Gamma \mid (\Sigma \leftarrow \Gamma) \in \mathcal{A} \text{ and } (\Sigma \leftarrow \Gamma) \text{ is not a literal}\}$$

Then the *normal form* $\langle K^n, \mathcal{A}^n \rangle$ is defined as follows where $n(R)$ maps each rule R to a fresh atom with the same free variables as R:

$$K^n = (K \setminus \mathcal{R}_{\mathcal{A}}) \cup \{\Sigma \leftarrow \Gamma, n(R) \mid R = (\Sigma \leftarrow \Gamma) \in \mathcal{R}_{\mathcal{A}}\} \cup \{n(R) \mid R \in K \cap \mathcal{R}_{\mathcal{A}}\}$$
$$\mathcal{A}^n = (\mathcal{A} \setminus \mathcal{R}_{\mathcal{A}}) \cup \{n(R) \mid R \in \mathcal{R}_{\mathcal{A}}\}$$

We define that any abducible literal L has name L, i.e., $n(L) = L$. It is shown in [13], that there is a 1-1 correspondence between (anti-)explanations with respect to $\langle K , A \rangle$ and those with respect to $\langle K^n, A^n \rangle$ for any observation O. That is, for $n(E) = \{n(R) \mid R \in E\}$ and $n(F) = \{n(R) \mid R \in F\}$: an observation O has a minimal (anti-)explanation (E, F) with respect to $\langle K , A \rangle$ iff O has a minimal (anti-)explanation $(n(E), n(F))$ with respect to $\langle K^n, A^n \rangle$. Hence, insertion (deletion) of a rule's name in the normal form corresponds to insertion (deletion) of the rule in the original program. In sum, with the normal form transformation, any abductive program with abducible *rules* is reduced to an abductive program with only abducible *literals*.

Example 6. We transform the example knowledge base K into its normal form based on a set of abducibles that is identical to K: that is $\mathcal{A} = K$; a similar setting will be used in Sect. 5.2 to achieve deletion of formulas from K. Hence we transform $\langle K, \mathcal{A}\rangle$ into its normal form $\langle K^n, \mathcal{A}^n\rangle$ as follows where we write $n(R)$ for the naming atom of the only rule in $\mathcal{A}$:

$$\begin{aligned} K^n = \{ & \mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}).,\quad \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}).,\quad n(R).,\\ & \mathit{Ill}(x, \mathsf{Aids}); \mathit{Ill}(x, \mathsf{Flu}) \leftarrow \mathit{Treat}(x, \mathsf{Medi1}), \mathit{not}\ \mathit{Treat}(x, \mathsf{Medi2}), n(R).\}\\ \mathcal{A}^n = \{ & \mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}),\quad \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}),\quad n(R)\ \} \end{aligned}$$

4.3 Update Programs

Minimal (anti-)explanations can be computed with *update programs* (UPs) [13]. The *update-minimal* (U-minimal) answer sets of a UP describe which rules have to be deleted from the program, and which rules have to be inserted into the program, in order to (un-)explain an observation.

For the given EDP K and a given set of abducibles $\mathcal{A}$, a set of **update rules** UR is devised that describe how entries of K can be changed. This is done with the following three types of rules.

1. [**Abducible rules**] The rules for abducible literals state that an abducible is either true in K or not. For each $L \in \mathcal{A}$, a new atom $\bar{L}$ is introduced that has the same variables as L. The set of abducible rules for each L is
$$\mathit{abd}(L) = \{L \leftarrow \mathit{not}\ \bar{L}.\ ,\ \bar{L} \leftarrow \mathit{not}\ L.\}.$$
2. [**Insertion rules**] Abducible literals that are not contained in K might be inserted into K and hence might occur in the set E of the explanation (E, F). For each $L \in \mathcal{A} \setminus K$, a new atom $+L$ is introduced and the insertion rule is
$$+L \leftarrow L.$$
3. [**Deletion rules**] Abducible literals that are contained in K might be deleted from K and hence might occur in the set F of the explanation (E, F). For each $L \in \mathcal{A} \cap K$, a new atom $-L$ is introduced and the deletion rule is
$$-L \leftarrow \mathit{not}\ L.$$

The **update program** is then defined by replacing abducible literals in K with the update rules; that is, $\mathit{UP} = (K \setminus \mathcal{A}) \cup \mathit{UR}$.

Example 7. Continuing Example 6, from $\langle K^n, \mathcal{A}^n \rangle$ we obtain

$$\begin{aligned}UP = {} & abd(Ill(\mathsf{Mary}, \mathsf{Aids})) \cup abd(Treat(\mathsf{Pete}, \mathsf{Medi1})) \cup abd(n(R)) \cup \\ & \{-Ill(\mathsf{Mary}, \mathsf{Aids}) \leftarrow not\ Ill(\mathsf{Mary}, \mathsf{Aids}).,\\ & -Treat(\mathsf{Pete}, \mathsf{Medi1}) \leftarrow not\ Treat(\mathsf{Pete}, \mathsf{Medi1}).,\\ & -n(R) \leftarrow not\ n(R).,\\ & Ill(x, \mathsf{Aids}); Ill(x, \mathsf{Flu}) \leftarrow Treat(x, \mathsf{Medi1}), not\ Treat(x, \mathsf{Medi2}), n(R).\}\end{aligned}$$

The set of atoms $+L$ is the set $\mathcal{UA}^+$ of positive update atoms; the set of atoms $-L$ is the set $\mathcal{UA}^-$ of negative update atoms. The set of **update atoms** is $\mathcal{UA} = \mathcal{UA}^+ \cup \mathcal{UA}^-$. From all answer sets of an update program UP we can identify those that are **update minimal** (U-minimal): they contain less update atoms than others. Thus, S is U-minimal iff there is no answer set T such that $T \cap \mathcal{UA} \subset S \cap \mathcal{UA}$.

4.4 Ground Observations

It is shown in [9] how in some situations the observation formulas O can be mapped to new positive ground observations. Non-ground atoms with variables can be mapped to a new ground observation. Several positive observations can be conjoined and mapped to a new ground observation. A negative observation (for which an anti-explanation is sought) can be mapped as a NAF-literal to a new positive observation (for which then an explanation has to be found). Moreover, several negative observations can be mapped as a conjunction of NAF-literals to one new positive observation such that its resulting explanation acts as an anti-explanation for all negative observations together. Hence, in extended abduction it is usually assumed that O is a positive ground observation for which an explanation has to be found. In case of finding a skeptical explanation, an inconsistency check has to be made on the resulting knowledge base. Transformations to a ground observation and inconsistency check will be detailed in Sect. 5.1 and applied to confidentiality-preservation.

5 Confidentiality-Preservation with UPs

We now show how to achieve confidentiality-preservation by extended abduction: we define the set of abducibles and describe how a confidentiality-preserving knowledge base can be obtained by computing U-minimal answer sets of the appropriate update program. We additionally distinguish between the case that we allow only deletions of formulas – that is, in the anti-explanation (E, F) the set E of positive anti-explanation formulas is empty – and the case that we also allow insertions of literals.

5.1 Policy Transformation for Credulous and Skeptical Users

Elements of the confidentiality policy will be treated as negative observations for which an anti-explanation has to be found while adding *prior* as invariable knowledge. Accordingly, we will transform policy elements to a set of rules containing new positive observations as sketched in Sect. 4.4. As these rules are distinct for credulous and skeptical users, we call them **policy transformation rules for credulous users** (PTR^{cred}) and **policy transformation rules for skeptical users** (PTR^{skep}), respectively. In the credulous user case, we aim to find a *skeptical* anti-explanation that unexplains all the policy elements at the same time; in other words, no answer set of the resulting knowledge base K^{pub} satisfies any of the policy elements. More formally, assume *policy* contains k elements. For each conjunction $C_i \in \mathit{policy}\ \ (i = 1 \dots k)$, we introduce a new negative ground observation O_i^- and map C_i to O_i^-. As each C_i is a conjunction of (NAF-)literals, the resulting formula is an EDP rule. In the credulous case, as a last policy transformation rule, we add one rule that maps all new negative ground observations O_i^- (in their NAF version) to a positive observation O^+:

$$PTR^{cred} = \{O_i^- \leftarrow C_i. \mid C_i \in \mathit{policy}\} \cup \{O^+ \leftarrow \mathit{not}\ \ O_1^-, \dots, \mathit{not}\ \ O_k^-.\}.$$

In the skeptical case, we have to treat every policy element individually; more precisely, for each single policy element, we have to find a *credulous* anti-explanation. In other words, for every policy element there must be at least one answer set of K^{pub} where it is not satisfied. For different policy elements these answer sets can however be different. For the credulous anti-explanation this has the consequence that each policy element has to be treated independent of others in the update program. That is why we obtain a set of rules PTR_i^{skep}: each policy element alone is mapped to the new positive observation O^+. Hence,

$$PTR_i^{skep} = \{O_i^- \leftarrow C_i. \mid C_i \in \mathit{policy}\} \cup \{O^+ \leftarrow \mathit{not}\ \ O_i^-.\}.$$

Example 8. The sets of policy transformation rules for *policy'* are

$$\begin{aligned} PTR^{cred} &= \{O_1^- \leftarrow \mathit{Ill}(x, \mathsf{Aids}).\ ,\ O_2^- \leftarrow \neg \mathit{AbleToWork}(x).\ , \\ &\qquad O^+ \leftarrow \mathit{not}\ \ O_1^-, \mathit{not}\ \ O_2^-.\} \\ PTR_1^{skep} &= \{\ O_1^- \leftarrow \mathit{Ill}(x, \mathsf{Aids}).,\ \ O^+ \leftarrow \mathit{not}\ \ O_1^-.\ \} \\ PTR_2^{skep} &= \{\ O_2^- \leftarrow \neg \mathit{AbleToWork}(x).,\ O^+ \leftarrow \mathit{not}\ \ O_2^-.\ \} \end{aligned}$$

Lastly, in both cases we consider an additional **goal rule** GR that enforces the single positive observation O^+: $GR = \{\leftarrow \mathit{not}\ \ O^+.\}$.

5.2 Deletions for Credulous Users

As a simplified setting, we first of all assume that in the credulous user case only deletions are allowed to achieve confidentiality-preservation. This setting can informally be described as follows: For a given knowledge base K , if we only allow deletions of rules from K, we have to find a *negative explanation* F that explains the new positive observation O^+ while respecting *prior* as invariable a priori knowledge. The set of abducibles is thus identical to K as we want to choose formulas from K for deletion: $\mathcal{A} = K$. That is, in total we consider the abductive program $\langle K, \mathcal{A}\rangle$. Then, we transform it into normal form $\langle K^n, \mathcal{A}^n\rangle$, and compute its update program UP as described in Sect. 4.3. As for *prior*, we add this set to the update program UP in order to make sure that the resulting answer sets of the update program do not contradict *prior*.

For the credulous user case, we finally add all the policy transformation rules PTR^{cred} and the goal rule GR. The goal rule is then meant as a constraint that only allows those answer sets of $UP \cup \mathit{prior} \cup PTR^{cred}$ in which O^+ is *true*. We thus obtain a new program P^{cred} as

$$P^{cred} = UP \cup \mathit{prior} \cup PTR^{cred} \cup GR$$

and compute its U-minimal answer sets. If S is one of these answer sets, the negative explanation F is obtained from the negative update atoms contained in S: $F = \{L \mid -L \in S\}$.

To obtain a confidentiality-preserving knowledge base for a credulous user, we have to check for inconsistency with the negation of the positive observation O^+ (which makes F a *skeptical* explanation of O^+); and allow only answer sets of P that are U-minimal among those respecting this inconsistency property. More precisely, we check whether

$$(K \setminus F) \cup \mathit{prior} \cup PTR^{cred} \cup \{\leftarrow O^+.\} \text{ is inconsistent.} \tag{1}$$

Example 9. We combine the update program UP of K with *prior* and the policy transformation rules PTR^{cred} and goal rule GR. This leads to the following U-minimal answer sets satisfying inconsistency property (1): $S_1' = \{-\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), -\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), n(R), \overline{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})}, \overline{\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1})}, O^+\}$, as a first, and $S_2' = \{-\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids}), \mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1}), -n(R), \overline{\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})}, \overline{n(R)}, O^+\}$ as a second answer set. These two answer sets correspond to the two minimal solutions with only deletions from Example 5 where $\mathit{Ill}(\mathsf{Mary}, \mathsf{Aids})$ must be deleted from K together with either $\mathit{Treat}(\mathsf{Pete}, \mathsf{Medi1})$ or the rule named R. Note that the two resulting $(K \setminus F)$ indeed satisfy inconsistency property (1), because O^+ is contained in every answer set of $(K \setminus F) \cup \mathit{prior} \cup PTR^{cred}$.

From Theorem 1 and the correspondence between update programs and explanations shown in [13], the following proposition follows for deletions.

Proposition 1 (Correctness for deletions for credulous users). *A knowledge base $K^{pub} = K \setminus F$ preserves confidentiality under the credulous response semantics and changes K subset-minimally iff F is obtained by an answer set of the program P^{cred} that is U-minimal among those satisfying the inconsistency property (1).*

5.3 Deletions for Skeptical Users

For the skeptical user case, we first have to find those abducibles that have to be deleted from K such that confidentiality is preserved for each individual policy element; hence, we find a credulous anti-explanation for each individual negative observation O_i^- (which indeed corresponds to a credulous explanation of O^+) by computing U-minimal answer sets for the following programs:

$$P_i^{skep} = UP \cup prior \cup PTR_i^{skep} \cup GR.$$

If S_i is one of these answer sets, the negative explanation F_i is obtained from the negative update atoms contained in S_i: $F_i = \{L \mid -L \in S_i\}$. We collect all negative explanations of P_i^{skep} in the set $\mathcal{F}_i$:

$$\mathcal{F}_i = \{F_i \mid F_i \text{ is obtained from a U-minimal answer set of } P_i^{skep}\}$$

In order to obtain a publishable knowledge base K^{pub} that preserves confidentiality for all policy elements, we combine the individual explanations $F_i \in \mathcal{F}_i$ in every possible way – and take the subset-minimal ones; that is, we obtain

$$\mathcal{F} = \{F = F_1 \cup \ldots \cup F_k \mid \; F_i \in \mathcal{F}_i \text{ and there is no } F' = F_1' \cup \ldots \cup F_k' \text{ (for } F_i' \in \mathcal{F}_i) \text{ such that } F' \subset F\}$$

Lastly, we choose those sets F from $\mathcal{F}$ that satisfy the following *consistency* check: the resulting knowledge base must be consistent with the negation of each of the policy entries. More formally, we check whether

$$(K \setminus F) \cup prior \cup PTR_i^{skep} \cup \; \{\leftarrow O_i^- .\} \; \text{ is consistent for each } i = 1, \ldots, k. \quad (2)$$

In other words, we verify that the combined explanation set F indeed preserves confidentiality of each single policy element. In sum, we make sure that no policy element can be deduced skeptically from $K^{pub} = (K \setminus F)$ together with the given background knowledge *prior* : for every policy element there is at least one answer set in which it is not true.

Example 10. In our running example for K with $prior$ and $policy'$, for PTR_1^{skep} $S_1'' = \{-Ill(\mathsf{Mary}, \mathsf{Aids}),\; Treat(\mathsf{Pete}, \mathsf{Medi1}),\; n(R), \overline{Ill(\mathsf{Mary}, \mathsf{Aids})}, Ill(\mathsf{Pete}, \mathsf{Flu}), \neg AbleToWork(\mathsf{Pete}), O^+\}$, is the only answer set; whereas for PTR_2^{skep} $S_2'' = \{Ill(\mathsf{Mary}, \mathsf{Aids}),\; Treat(\mathsf{Pete}, \mathsf{Medi1}), n(R), Ill(\mathsf{Pete}, \mathsf{Aids}), O^+\}$ is the only answer set. Only the update atom $-Ill(\mathsf{Mary}, \mathsf{Aids})$ appears in S_1''; and hence $\mathcal{F} = \{Ill(\mathsf{Mary}, \mathsf{Aids})\}$. Which means that we obtain the minimal solution from Example 5 by deleting $Ill(\mathsf{Mary}, \mathsf{Aids})$ from K. Note that the resulting $(K \setminus F)$ indeed satisfies consistency property (2), because each $(K \setminus F) \cup prior \cup PTR_i^{skep}$ has at least on answer set in which O_i^- is not contained.
Note that it is indeed necessary to compute each explanation individually: otherwise, for the example P^{cred} credulous and skeptical explanations coincide and hence would delete more entries from K than necessary.

Similar to Proposition 1, the following result follows for skeptical users.

Proposition 2 (Correctness for deletions for skeptical users). *A knowledge base $K^{pub} = K \setminus F$ preserves confidentiality under the skeptical response semantics and changes K subset-minimally iff F is obtained by combining update atoms of the answer sets of the programs P_i^{skep} that are U-minimal among those satisfying the consistency property (2) for each i.*

5.4 Deletions and Literal Insertions for Credulous Users

To obtain a confidentiality-preserving knowledge base, (incorrect) entries may also be inserted into the knowledge base. To allow for insertions of literals, a more complex set $\mathcal{A}$ of abducibles has to be chosen. We reinforce the point that the subset $\mathcal{A} \cap K$ of abducibles that are already contained in the knowledge base K are those that may be deleted while the subset $\mathcal{A} \setminus K$ of those abducibles that are not contained in K may be inserted. In general, for literal insertions we could take the whole set of atoms that can be obtained by considering predicate symbols from the knowledge base K and the a priori knowledge *prior*, and then instantiating them in all possible ways according to the Herbrand universe of K and *prior*. By taking all atoms and their negations we obtain a set of literals; all those literals that are not contained in K can be used as abducibles for a positive explanation E. In other words, they can potentially be inserted into K to avoid deduction of secrets.

However, we can reduce this number of new abducibles by analyzing which literals have influence on the policy elements at all. First of all, we assume that the policy transformation is applied as described in Sect. 5.1. Then, starting from the atoms in policy elements C_i, we trace back all rules in $K \cup prior$ that influence these policy atoms and collect all atoms in the bodies as well as heads of these rules. In other words, we construct a dependency graph (similar to [17]). However, in contrast to the traditional dependency graph, (as EDPs allow disjunction in rule heads) we do not only consider body atoms but also the head atoms as well as all their negations. More formally, let P_0 be the set of literals that can be obtained from atoms that appear in the policy:

$$P_0 = \{A, \neg A \mid A \text{ is an atom in a literal or NAF-literal in a policy element}\}$$

Next we iterate and collect all the literals that the P_0 literals depend on:

$$\begin{aligned} P_{j+1} = \{A, \neg A \mid\ & A \text{ is an atom in a literal or NAF-literal in the head or body of} \\ & \text{a rule } R \text{ where } R \in K \cup prior \text{ and } head(R) \cap P_j \neq \emptyset\} \end{aligned}$$

and combine all these literals in a set $\mathcal{P} = (\bigcup_{j=0}^{\infty} P_j)$.

As we also want to have the option to delete rules from K (not only the literals in $\mathcal{P}$), we define the set of abducibles as the set $\mathcal{P}$ plus all those rules in K whose head depends on literals in $\mathcal{P}$:

$$\mathcal{A} = \mathcal{P} \cup \{R \mid R \in K \text{ and } head(R) \cap \mathcal{P} \neq \emptyset\}$$

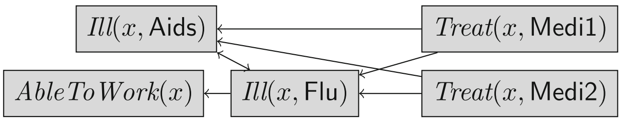


Fig. 2. Dependency graph for literals in *policy* wrt. $K \cup prior$

Example 11. For the example $K \cup prior \cup PTR^{cred}$, the dependency graph on atoms is shown in Fig. 2. We note that the policy atom $Ill(x, \mathsf{Aids})$ directly depends on the atoms $Ill(x, \mathsf{Flu})$, $Treat(x, \mathsf{Medi1})$ and $Treat(x, \mathsf{Medi2})$; the policy atom $AbleToWork(x)$ directly depends on the atom $Ill(x, \mathsf{Flu})$ which again depends on $Ill(x, \mathsf{Aids})$, $Treat(x, \mathsf{Medi1})$ and $Treat(x, \mathsf{Medi2})$. In the end, considering negations of these atoms $\mathcal{P} = \{(\neg)Ill(x, \mathsf{Aids}), (\neg)AbleToWork(x), (\neg)Ill(x, \mathsf{Flu}), (\neg)Treat(x, \mathsf{Medi1}), (\neg)Treat(x, \mathsf{Medi2})\}$ is obtained. Lastly, we also have to add the rule R from K to $\mathcal{A}$ because literals in its head are contained in $\mathcal{P}$.

We obtain the normal form and then the update program UP for K and the new set of abducibles $\mathcal{A}$. The process of finding a skeptical explanation (for the new positive observation O^+) proceeds with finding an answer set of program P^{cred} as in Sect. 5.2 where additionally the positive explanation E is obtained as $E = \{L \mid +L \in S\}$ and S is U-minimal among those satisfying

$$(K \setminus F) \cup E \cup prior \cup PTR^{cred} \cup \{\leftarrow O^+.\} \text{ is inconsistent.} \tag{3}$$

Example 12. For UP from Example 9 the new set of abducibles leads to new insertion rules. The insertion rules for the new abducibles $Treat(\mathsf{Pete}, \mathsf{Medi2})$, $\neg Ill(x, \mathsf{Aids})$ and $Ill(x, \mathsf{Flu})$ are $+Treat(\mathsf{Pete}, \mathsf{Medi2}) \leftarrow Treat(\mathsf{Pete}, \mathsf{Medi2})$, as well as $+\neg Ill(x, \mathsf{Aids}) \leftarrow \neg Ill(x, \mathsf{Aids})$ and $+Ill(x, \mathsf{Flu}) \leftarrow Ill(x, \mathsf{Flu})$. With these new rules included in UP, we also obtain the solutions of Example 5 where the appropriate facts are inserted into K (together with deletion of $Ill(\mathsf{Mary}, \mathsf{Aids})$).

Proposition 3 (Correctness for deletions and literal insertions for credulous users). *A knowledge base $K^{pub} = (K \setminus F) \cup E$ preserves confidentiality and changes K subset-minimally iff (E, F) is obtained by an answer set of program P^{cred} that is U-minimal among those satisfying inconsistency property (3).*

5.5 Deletions and Literal Insertions for Skeptical Users

For skeptical users, we have to obtain the same new set of abducibles as for credulous users by tracing back all dependencies. But in the skeptical case we again have to find an anti-explanation for each policy element individually to avoid changing the knowledge base K more than necessary. Hence, we obtain the update programs UP based on the new set of abducibles and compute U-minimal answer sets of the following individual programs:

$$P_i^{skep} = UP \cup prior \cup PTR_i^{skep} \cup GR.$$

These answer sets may now contain positive explanations E_i as well as negative explanations F_i. If S_i is one of these answer sets, F_i is obtained from the negative update atoms contained in S_i: $F_i = \{L \mid -L \in S_i\}$ whereas E_i is $E_i = \{L \mid +L \in S_i\}$. We collect these explanations of P_i^{skep} in the set Exp_i:

$$Exp_i = \{(E_i, F_i) \mid (E_i, F_i) \text{ is obtained from a U-minimal answer set of } P_i^{skep}\}$$

Similar to the deletion only case, we combine the individual explanations $(E_i, F_i) \in Exp_i$ in every possible way – and take the subset-minimal ones; that is, we obtain

$$\begin{aligned} Exp = \{(E,F) \mid\ & F = F_1 \cup \ldots \cup F_k, E = E_1 \cup \ldots \cup E_k, (E_i, F_i) \in Exp_i \\ & \text{and there is no } F' = F_1' \cup \ldots \cup F_k' \text{ and no } E' = E_1' \cup \ldots \cup E_k' \\ & (\text{for } (E_i', F_i') \in Exp_i) \text{ such that } F' \cup E' \subset F \cup E\} \end{aligned}$$

We choose those sets (E, F) from Exp that satisfy the following *consistency* check: the resulting knowledge base must be consistent with the negation of each of the policy entries. More formally, we check whether

$$(K \setminus F) \cup E \cup prior \cup PTR_i^{skep} \cup \{\leftarrow O_i^-.\} \text{ is consistent for each } i = 1, \ldots, k. \quad (4)$$

That is, we again verify that the combined explanation (E, F) preserves confidentiality of each single policy element, and hence no policy element can be deduced skeptically from $K^{pub} = (K \setminus F) \cup E$ together with the given background knowledge *prior*.

Example 13. To give an example for literal insertions for skeptical users, we consider the given example K and *policy* as well as a new a priori knowledge $prior' = \{\neg Ill(\mathsf{Pete}, \mathsf{Flu})\}$. In this case, from $K \cup prior'$ the secrets $Ill(\mathsf{Mary}, \mathsf{Aids})$ and $Ill(\mathsf{Pete}, \mathsf{Aids})$ can both be deduced skeptically. There is only one *policy* element and hence only one program PTR_1^{skep}. This program has two U-minimal answer sets; one containing $-\neg Ill(\mathsf{Mary}, \mathsf{Aids})$ and $-Treat(\mathsf{Pete}, \mathsf{Medi1})$ and a second one containing $-\neg Ill(\mathsf{Mary}, \mathsf{Aids})$ and $+Treat(\mathsf{Pete}, \mathsf{Medi2})$. Hence we now also have the option to insert $Treat(\mathsf{Pete}, \mathsf{Medi2})$ in order protect the secret $Ill(\mathsf{Pete}, \mathsf{Aids})$.

Proposition 4 (Correctness for deletions and literal insertions for skeptical users). *A knowledge base $K^{pub} = (K \setminus F) \cup E$ preserves confidentiality and changes K subset-minimally iff (E, F) is obtained by combining update atoms of the answer sets of the programs P_i^{skep} that are U-minimal among those satisfying the consistency property (4) for each i.*

6 Discussion and Conclusion

This article showed that when publishing an extended disjunctive logic program, confidentiality-preservation can be ensured by extended abduction; more precisely, we showed that under the credulous and skeptical query response semantics it reduces to finding anti-explanations with update programs. This is an application of data modification, because a user can be misled by the published knowledge base to believe incorrect information; we hence apply dishonesties [12] as a security mechanism. This is in contrast to [17] whose aim is to avoid incorrect deductions while enforcing access control on a knowledge base. Another difference to [17] is that they do not allow disjunctions in rule heads; hence, to the best of our knowledge this article is the first one to handle a confidentiality problem for EDPs. In [3] the authors study databases that may provide users with incorrect answers to preserve security in a multi-user environment. Differently from our approach, they consider a database as a set of formulas of propositional logic and formulate the problem using modal logic. Future work might handle insertion of non-literal rules. Moreover, the whole system could be extended by preferences among the possible solutions. Generally, we can consider preferences such that deleting facts is preferred to deleting rules, or inserting facts with non-confidential predicates is preferred to inserting facts with confidential ones.

References

1. Afrati, F.N., Kolaitis, P.G.: Repair checking in inconsistent databases: algorithms and complexity. ICDT 2009. ACM International Conference Proceeding Series, vol. 361, pp. 31–41. ACM, New York (2009)
2. Biskup, J.: Usability confinement of server reactions: maintaining inference-proof client views by controlled interaction execution. In: Kikuchi, S., Sachdeva, S., Bhalla, S. (eds.) DNIS 2010. LNCS, vol. 5999, pp. 80–106. Springer, Heidelberg (2010)
3. Bonatti, P.A., Kraus, S., Subrahmanian, V.S.: Foundations of secure deductive databases. IEEE Trans. Knowl. Data Eng. **7**(3), 406–422 (1995)
4. Calimeri, F., et al.: The third answer set programming competition: preliminary report of the system competition track. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 388–403. Springer, Heidelberg (2011)
5. Dix, J., Faber, W., Subrahmanian, V.S.: The relationship between reasoning about privacy and default logics. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 637–650. Springer, Heidelberg (2005)
6. Farkas, C., Jajodia, S.: The inference problem: a survey. SIGKDD Explor. **4**(2), 6–11 (2002)
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gen. Comput. **9**(3/4), 365–386 (1991)
8. Grau, B.C., Horrocks, I.: Privacy-preserving query answering in logic-based information systems. In: ECAI 2008. Frontiers in Artificial Intelligence and Applications, vol. 178, pp. 40–44. IOS Press (2008)
9. Inoue, K., Sakama, C.: Abductive framework for nonmonotonic theory change. In: Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95), vol. 1, pp. 204–210. Morgan Kaufmann (1995)
10. Inoue, K., Sakama, C., Wiese, L.: Confidentiality-preserving data publishing for credulous users by extended abduction. Paper appears in the Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011), arXiv:1108.5825 (2011)
11. Ma, J., Russo, A., Broda, K., Lupu, E.: Multi-agent confidential abductive reasoning. In: ICLP (Technical Communications). LIPIcs, vol. 11, pp. 175–186. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2011)

12. Sakama, C.: Dishonest reasoning by abduction. In: 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 1063–1064. IJCAI/AAAI (2011)
13. Sakama, C., Inoue, K.: An abductive framework for computing knowledge base updates. Theor. Pract. Logic Program. **3**(6), 671–713 (2003)
14. Stouppa, P., Studer, T.: Data privacy for $\mathcal{ALC}$ knowledge bases. In: Artemov, S., Nerode, A. (eds.) LFCS 2009. LNCS, vol. 5407, pp. 409–421. Springer, Heidelberg (2008)
15. Toland, T.S., Farkas, C., Eastman, C.M.: The inference problem: Maintaining maximal availability in the presence of database updates. Comput. Secur. **29**(1), 88–103 (2010)
16. Wiese, L.: Horizontal fragmentation for data outsourcing with formula-based confidentiality constraints. In: Echizen, I., Kunihiro, N., Sasaki, R. (eds.) IWSEC 2010. LNCS, vol. 6434, pp. 101–116. Springer, Heidelberg (2010)
17. Zhao, L., Qian, J., Chang, L., Cai, G.: Using ASP for knowledge management with user authorization. Data Knowl. Eng. **69**(8), 737–762 (2010)

INAP Technical Papers III: Semantics

Every Formula-Based Logic Program Has a Least Infinite-Valued Model

Rainer Lüdecke[✉]

Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13, 72076 Tübingen, Germany
luedecke@informatik.uni-tuebingen.de

Abstract. Every definite logic program has as its meaning a least Herbrand model with respect to the program-independent ordering $\subseteq$. In the case of normal logic programs there do not exist least models in general. However, according to a recent approach by Rondogiannis and Wadge, who consider infinite-valued models, every normal logic program does have a least model with respect to a program-independent ordering. We show that this approach can be extended to formula-based logic programs (i.e., finite sets of rules of the form $A \leftarrow \phi$ where A is an atom and ϕ an arbitrary first-order formula). We construct for a given program P an interpretation M_P and show that it is the least of all models of P.

Keywords: Logic programming · Semantics of programs · Negation-as-failure · Infinite-valued logics · Set theory

1 Introduction

It is well-known that every definite logic program P has a Herbrand model and the intersection of all its Herbrand models is also a model of P. We call it the least Herbrand model or the canonical model of P and constitute that it is the intended meaning of the program. If we consider a normal logic program P it is more complicated to state the intended meaning of the program because the intersection of all its models is not necessarily a model. There are many approaches to overcome that problem. The existing approaches are not purely model-theoretic (i.e., there are normal logic programs that have the same models but different intended meanings). However, there is a recent purely model-theoretic approach of P. Rondogiannis and W. Wadge [3]. They prove that every normal logic program has a least infinite-valued model. Their work is based on an infinite set of truth values, ordered as follows:

$$\mathcal{F}_0 < \mathcal{F}_1 < \cdots < \mathcal{F}_\alpha < \cdots < 0 < \cdots < \mathcal{T}_\alpha < \cdots < \mathcal{T}_1 < \mathcal{T}_0$$

Intuitively, $\mathcal{F}_0$ and $\mathcal{T}_0$ are the classical truth values *False* and *True*, 0 is the truth value *Undefined* and α is an arbitrary countable ordinal. The considered ordering of the interpretations is a program-independent ordering on the infinite-valued interpretations and generalizes the classical ordering on the Herbrand interpretations. The intended meaning of a normal logic program is, as in the classical case, stated as the unique minimal infinite-valued model of P. Furthermore, they show that the 3-valued interpretation that results from the least infinite-valued model of P by collapsing all true values to True and all false values to False coincides with the well-founded model of P introduced in [2].

Inspired by [4] we consider in this paper formula-based logic programs. A formula-based logic program is a finite set of rules of the form $A \leftarrow \phi$, where A is an atomic formula and ϕ is an arbitrary first-order formula. We show that the construction methods to obtain the least infinite-valued model of a normal logic program P given in [3] can be adapted to formula-based logic programs. The initial step to carry out this adaption is the proof of two extension theorems. Informally speaking, these theorems state that a complex formula shows the same behavior as an atomic formula. While Rondogiannis and Wadge [3] make use of the fact that the bodies of normal program rules are conjunctions of negative or positive atoms, we instead make use of one of the extension theorems. The second step to achieve the adaption is the set-theoretical fact that the least uncountable cardinal $\aleph_1$ is regular (i.e., the limit of a countable sequence of countable ordinals is in $\aleph_1$). Contrary to the bodies of normal program rules, the bodies of formula-based program rules can refer a ground atom to a countably infinite set of ground atoms. This is the reason why we must use in our approach $\aleph_1$ many iteration steps in the construction of the least model of a given program P in conjunction with the regularity of $\aleph_1$. In [3] ω many iteration steps in conjunction with the fact that the limit of a finite sequence of natural numbers is once again a natural number is sufficient to construct the least model. Towards the end of the paper, we use again the regularity of $\aleph_1$ to show that there is a countable ordinal $\delta_{\max}$ with the property that every least model of a formula-based logic-program refers only to truth values of the form $\mathcal{T}_\alpha$ or $\mathcal{F}_\alpha$ or 0, where $\alpha < \delta_{\max}$. This implies that we only need a very small fragment of the truth values if we consider the meaning of a formula-based logic program. Finally, we show that the 3-valued interpretation that results from the least infinite-valued model of a given formula-based logic program by collapsing all true values to True and all false values to False, is a model of P in the sense of [2]. But compared to the case of normal logic programs, the collapsed least infinite-valued model of a formula-based logic program is not a minimal 3-valued model of P in general. However, there is a simple restriction for the class of formula-based programs such that the collapsed model is minimal in general.

At this point we would like to mention that we did not develop the theory presented in this paper with respect to applied logic. We have a predominantly theoretical interest in extending the notion of *inductive definition* to a wider class of rules.

We make heavy use of ordinal numbers in this paper. Therefore, we included an appendix with a short introduction to ordinal numbers for those readers who are not familiar with this part of set theory. Moreover, one can find the omitted proofs and a detailed discussion of an example within the appendix. It is downloadable at: http://www-ls.informatik.uni-tuebingen.de/luedecke/luedecke.html

2 Infinite-Valued Models

We are interested in logic programs based on a first-order Language $\mathcal{L}$ with finitely many predicate symbols, function symbols, and constants.

Definition 1. The alphabet of $\mathcal{L}$ consists of the following symbols, where the numbers n, m, l, s_1,...,s_n, r_1,...,r_m are natural numbers such that $n, l, r_i \geq 1$ and $m, s_i \geq 0$ hold:

1. Predicate symbols: $P_1, ..., P_n$ with assigned arity $s_1, ..., s_n$
2. Function symbols: $f_1, ..., f_m$ with assigned arity $r_1, ..., r_m$
3. Constants (abbr.: Con): $c_1, ..., c_l$
4. Variables (abbr.: Var): x_k provided that $k \in \mathbb{N}$
5. Connectives: $\wedge, \vee, \neg, \forall, \exists, \perp, \top$
6. Punctuation symbols: '(', ')' and ','

The natural numbers $n, m, l, s_1, ..., s_n, r_1, ..., r_m \in \mathbb{N}$ are fixed and the language $\mathcal{L}$ only depends on these numbers. If we consider different languages of this type, we will write $\mathcal{L}_{n,m,l,(s_i),(r_i)}$ instead of $\mathcal{L}$ to prevent confusion. The following definitions depend on $\mathcal{L}$. However, to improve readability, we will not mention this again.

Definition 2. The set of *terms* Term is the smallest one satisfying:

1. Constants and variables are in Term.
2. If $t_1, ..., t_{r_i} \in$ Term, then $f_i(t_1, ..., t_{r_i}) \in$ Term.

Definition 3. The *Herbrand universe* H_U is the set of ground terms (i.e., terms that contain no variables).

Definition 4. The set of *formulas* Form is the smallest one satisfying:

1. $\perp$ and $\top$ are elements of Form.
2. If $t_1, ..., t_{s_k} \in$ Term, then $P_k(t_1, ..., t_{s_k}) \in$ Form.
3. If $\phi, \psi \in$ Form and $v \in$ Var, then $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\forall v\phi)$, $(\exists v\phi) \in$ Form.

An *atom* is a formula only constructed by means of (1.) or (2.) and a *ground atom* is an atom that contains no variables.

Definition 5. The *Herbrand base* H_B is the set of all ground atoms, except $\perp$ and $\top$.

Definition 6. A *(formula-based) rule* is of the form $A \leftarrow \phi$ where ϕ is an arbitrary formula and A is an arbitrary atom provided that $A \neq \top$ and $A \neq \perp$.

A *(formula-based) logic program* is a finite set of (formula-based) rules. Notice that we write $A \leftarrow$ instead of $A \leftarrow \top$. Remember $A \leftarrow \phi$ is called a *normal rule* (resp. *definite rule*) if ϕ is a conjunction of literals (resp. positive literals). A finite set of normal (resp. definite) rules is a *normal (resp. definite) program.*

Definition 7. The set of truth values W is given by

$$W := \{\langle 0, n\rangle\, ;\ n \in \aleph_1\} \cup \{0\} \cup \{\langle 1, n\rangle\, ;\ n \in \aleph_1\}\, .$$

Additionally, we define a strict linear ordering $<$ on W as follows:

1. $\langle 0, n\rangle < 0$ and $0 < \langle 1, n\rangle$ for all $n \in \aleph_1$
2. $\langle w, x\rangle < \langle y, z\rangle$ iff
   $(w = 0 = y$ and $x \in z)$ or $(w = 1 = y$ and $z \in x)$ or $(w = 0$ and $y = 1)$

We define $\mathcal{F}_i := \langle 0, i\rangle$ and $\mathcal{T}_i := \langle 1, i\rangle$ for all $i \in \aleph_1$. $\mathcal{F}_i$ is said to be a *false* value and $\mathcal{T}_i$ is called a *true* value. The value 0 is the *undefined* value. The following summarizes the situation ($i \in \aleph_1$):

$$\mathcal{F}_0 < \mathcal{F}_1 < \mathcal{F}_2 < \cdots < \mathcal{F}_i < \cdots < 0 < \cdots < \mathcal{T}_i < \cdots < \mathcal{T}_2 < \mathcal{T}_1 < \mathcal{T}_0$$

Definition 8. The *degree* (abbr.: deg) of a truth value is given by $\deg(0) := \infty$, $\deg(\mathcal{F}_\alpha) := \alpha$, and $\deg(\mathcal{T}_\alpha) := \alpha$ for all $\alpha \in \aleph_1$.

Definition 9. An *(infinite-valued Herbrand) interpretation* I is a function from the Herbrand base H_B to the set of truth values W. A *variable assignment* h is a mapping from Var to H_U.

Definition 10. Let I be an interpretation and $w \in W$ be a truth value, then $I\|w$ is defined as the inverse image of w under I (i.e., $I\|w = \{A \in H_B;\ I(A) = w\}$).

Definition 11. Let I and J be interpretations and $\alpha \in \aleph_1$. We write $I =_\alpha J$, if for all $\beta \leq \alpha$, $I\|\mathcal{F}_\beta = J\|\mathcal{F}_\beta$ and $I\|\mathcal{T}_\beta = J\|\mathcal{T}_\beta$.

Definition 12. Let I and J be interpretations and $\alpha \in \aleph_1$. We write $I \sqsubseteq_\alpha J$, if for all $\beta < \alpha$, $I =_\beta J$ and furthermore $J\|\mathcal{F}_\alpha \subseteq I\|\mathcal{F}_\alpha$ & $I\|\mathcal{T}_\alpha \subseteq J\|\mathcal{T}_\alpha$. We write $I \sqsubset_\alpha J$, if $I \sqsubseteq_\alpha J$ and $I \neq_\alpha J$.

Now we define a partial ordering $\sqsubseteq_\infty$ on the set of all interpretations. It is easy to see that this ordering generalizes the classical partial ordering $\subseteq$ on the set of 2-valued Herbrand interpretations.

Definition 13. Let I and J be interpretations. We write $I \sqsubset_\infty J$, if there exists an $\alpha \in \aleph_1$ such that $I \sqsubset_\alpha J$. We write $I \sqsubseteq_\infty J$, if $I \sqsubset_\infty J$ or $I = J$.

Remark 1. To motivate these definitions let us briefly recall the classical 2-valued situation. Therefore, let us pick two (2-valued) Herbrand interpretations $I, J \subseteq H_B$. Considering these, it becomes apparent that $I \subseteq J$ holds if and only if the set of ground atoms that are false w.r.t. J is a subset of the set of ground atoms that are false w.r.t. I and the set of ground atoms that are true w.r.t. I is a subset of the set of ground atoms that are true w.r.t. J.

Definition 14. Let h be a variable assignment. The *semantics of terms* is given by (with respect to h):

1. $[\![c]\!]_h = c$ if c is a constant.
2. $[\![v]\!]_h = h(v)$ if v is a variable.
3. $[\![f_i(t_1, ..., t_{r_i})]\!]_h = f_i([\![t_1]\!]_h, ..., [\![t_{r_i}]\!]_h)$ if $1 \leq i \leq m$ and $t_1, ..., t_{r_i} \in$ Term.

Before we start to talk about the semantics of formulas, we have to show that every subset of W has a *least upper bound* (abbr: sup) and a *greatest lower bound* (abbr: inf). The proof of the following lemma is left to the reader. The proof is using the fact that every nonempty subset of $\aleph_1$ has a least element.

Lemma 1. *For every subset $M \subseteq W$ the least upper bound sup$M$ and the greatest lower bound inf$M$ exist in W. Moreover, sup$M \in \{\mathcal{T}_\alpha;\ \alpha \in \aleph_1\}$ implies that sup$M \in M$ and on the other hand inf$M \in \{\mathcal{F}_\alpha;\ \alpha \in \aleph_1\}$ implies that inf$M \in M$.*

Definition 15. Let I be an interpretation and h be a variable assignment. The *semantics of formulas* is given by (with respect to I and h):

1. If $t_1, ..., t_{s_k} \in$ Term, then $[\![P_k(t_1, ..., t_{s_k})]\!]_h^I = I\left(P_k([\![t_1]\!]_h, ..., [\![t_{s_k}]\!]_h)\right)$. Additionally, the semantics of $\top$ and $\bot$ is given by $[\![\top]\!]_h^I = \mathcal{T}_0$ and $[\![\bot]\!]_h^I = \mathcal{F}_0$.
2. If $\phi, \psi \in$ Form and v an arbitrary variable, then $[\![\phi \wedge \psi]\!]_h^I = \min\{[\![\phi]\!]_h^I, [\![\psi]\!]_h^I\}$, $[\![\phi \vee \psi]\!]_h^I = \max\{\ [\![\phi]\!]_h^I\ ,\ [\![\psi]\!]_h^I\ \}$, $[\![\exists v(\phi)]\!]_h^I = \sup\{[\![\phi]\!]_{h[v\mapsto u]}^I;\ u \in H_U\}$, $$[\![\forall v(\phi)]\!]_h^I = \inf\{[\![\phi]\!]_{h[v\mapsto u]}^I;\ u \in H_U\} \text{ and } [\![\neg(\phi)]\!]_h^I = \begin{cases} \mathcal{T}_{\alpha+1}, & \text{if } [\![\phi]\!]_h^I = \mathcal{F}_\alpha \\ \mathcal{F}_{\alpha+1}, & \text{if } [\![\phi]\!]_h^I = \mathcal{T}_\alpha \\ 0, & \text{otherwise} \end{cases}.$$

Definition 16. Let $A \leftarrow \phi$ be a rule, P a program and I an interpretation. Then I *satisfies* $A \leftarrow \phi$ if for all variable assignment h the property $[\![A]\!]_h^I \geq [\![\phi]\!]_h^I$ holds. Furthermore, I is a *model* of P if I satisfies all rules of P.

Definition 17. Let $A \leftarrow \phi$ be a rule and σ be a variable substitution (i.e., a function from Var to Term with finite support). Then, $A\sigma \leftarrow \phi\sigma$ is a *ground instance* of the rule $A \leftarrow \phi$ if $A\sigma \in H_B$ and all variables in $\phi\sigma$ are in the scope of a quantifier. It is easy to see that $[\![A\sigma]\!]_h^I$ and $[\![\phi\sigma]\!]_h^I$ (with respect to an interpretation I and a variable assignment h) depend only on I. That is why we write also $[\![A\sigma]\!]^I$ and $[\![\phi\sigma]\!]^I$. We denote the *set of all ground instances* of a program P with P_G.

Example 1. Consider the formula-based program P given by the set of rules $\{P(c) \leftarrow,\ R(x) \leftarrow \neg P(x),\ P(Sx) \leftarrow \neg R(x),\ Q \leftarrow \forall x\, P(x)\}$. Then it is easy to prove that the Herbrand interpretation $I = \{P(S^n c) \mapsto \mathcal{T}_{2n};\ n \in \mathbb{N}\} \cup \{R(S^n c) \mapsto \mathcal{F}_{2n+1};\ n \in \mathbb{N}\} \cup \{Q \mapsto \mathcal{T}_\omega\}$ is a model of P. Moreover, using the results of this paper one can show that it is also the least Herbrand model of P.

Remark 2. Before we proceed we want to give a short informal but intuitive description of the semantics given above. Let us consider two rabbits named Bugs Bunny and Roger Rabbit. We know about them, that Bugs Bunny is a grey rabbit and if Roger Rabbit is not a grey rabbit, then he is a white one. This information can be understood as a normal logic program:

$$\begin{aligned} \mathit{grey}(\mathit{Bugs\,Bunny}) &\Leftarrow \\ \mathit{white}(\mathit{Roger\,Rabbit}) &\Leftarrow \mathit{not grey}(\mathit{Roger\,Rabbit}) \end{aligned}$$

There is no doubt that *Bugs Bunny is grey* is true because it is a fact. There is also no doubt that every try to prove that *Roger Rabbit is grey* will fail. Hence, using the negation-as-failure rule, we can infer that *Roger Rabbit is white* is also true. But everybody would agree that there is a difference of quality between the two statements because negation-as-failure is not a sound inference rule. The approach of [3] suggests that the ground atom *grey(Bugs Bunny)* receives the best possible truth value named $\mathcal{T}_0$ because it is a fact of the program. The atom *grey(Roger Rabbit)* receives the worst possible truth value named $\mathcal{F}_0$ because of the negation-as-failure approach. Hence, using the above semantics for negation, *white(Roger Rabbit)* receives only the second best truth value $\mathcal{T}_1$.

3 The Immediate Consequence Operator

Definition 18. Let P be a program, then the *immediate consequence operator* T_P for the program P is a mapping from and into $\{I;\ I \text{ is an interpretation}\}$, where $T_P(I)$ maps an $A \in H_B$ to $T_P(I)(A) := \sup\{[\![\phi]\!]^I; A \leftarrow \phi \in P_G\}$. (Notice that P_G can be infinite and hence we cannot use max instead of sup.)

Definition 19. Let α be an arbitrary countable ordinal. A function T from and into the set of interpretations is called *α-monotonic* iff for all interpretations I and J the property $I \sqsubseteq_\alpha J \Rightarrow T(I) \sqsubseteq_\alpha T(J)$ holds.

We will show that T_P is α-monotonic. Before we will give the proof of this property, we have to prove the first extension theorem.

Theorem 1 (Extension Theorem I). *Let α be an arbitrary countable ordinal and I, J two interpretations provided that $I \sqsubseteq_\alpha J$. The following properties hold for every formula ϕ:*

1. *If $\mathcal{F}_0 \leq w \leq \mathcal{F}_\alpha$ and h an assignment, then $[\![\phi]\!]^J_h = w \Rightarrow [\![\phi]\!]^I_h = w$.*
2. *If $\mathcal{T}_\alpha \leq w \leq \mathcal{T}_0$ and h an assignment, then $[\![\phi]\!]^I_h = w \Rightarrow [\![\phi]\!]^J_h = w$.*
3. *If $deg(w) < \alpha$ and h an assignment, then $[\![\phi]\!]^I_h = w \Leftrightarrow [\![\phi]\!]^J_h = w$.*

Proof. We show these statements by induction on ϕ. Let $I_H(X)$ be an abbreviation for 1. and 2. and 3., where ϕ is replaced by X (induction hypothesis).

Case 1: $\phi = \top$ or $\phi = \bot$. In this case 1., 2., and 3. are obviously true.
Case 2: $\phi = P_k(t_1, ..., t_{s_k})$. 1., 2., and 3. follow directly from $I \sqsubseteq_\alpha J$.
Case 3: $\phi = \neg A$. We assume that $I_H(A)$. We show simultaneously that 1., 2. and 3. also hold. Therefore, we choose an assignment h and a truth value w such that $\mathcal{F}_0 \leq w \leq \mathcal{F}_\alpha$ resp. $\mathcal{T}_\alpha \leq w \leq \mathcal{T}_0$ resp. $\deg(w) < \alpha$. Assume that $[\![\phi]\!]^J_h = w$ resp.

$[\![\phi]\!]_h^I = w$ resp. $[\![\phi]\!]_h^{K_1} = w$ (where $K_1 = I$ and $K_2 = J$ or $K_1 = J$ and $K_2 = I$). Using Definition 15 we get that $\mathcal{T}_{\alpha-1} \leq [\![A]\!]_h^J \leq \mathcal{T}_0$ resp. $\mathcal{F}_0 \leq [\![A]\!]_h^I \leq \mathcal{F}_{\alpha-1}$ resp. $\deg([\![A]\!]_h^{K_1}) < \alpha - 1$. Thus, using the third part of $I_H(A)$, $[\![A]\!]_h^J = [\![A]\!]_h^I$ resp. $[\![A]\!]_h^I = [\![A]\!]_h^J$ resp. $[\![A]\!]_h^{K_1} = [\![A]\!]_h^{K_2}$. Finally, using Definition 15, we get that $[\![\phi]\!]_h^I = w$ resp. $[\![\phi]\!]_h^J = w$ resp. $[\![\phi]\!]_h^{K_2} = w$.

Before we can go on with the next case, we must prove the following technical lemma.

Lemma 2. *We use the same assumptions as in Theorem 1. Let $\mathcal{I}$ be a set of indices, A_i $(i \in \mathcal{I})$ a formula provided that $I_H(A_i)$ and h_i $(i \in \mathcal{I})$ an assignment. We define $inf_K := inf\{[\![A_i]\!]_{h_i}^K; i \in \mathcal{I}\}$ and $sup_K := sup\{[\![A_i]\!]_{h_i}^K; i \in \mathcal{I}\}$ (where $K = I, J$). Then the following holds:*

1. $\inf_J = \mathcal{F}_\gamma \;\Rightarrow\; \inf_I = \mathcal{F}_\gamma$ *(for all $\gamma \leq \alpha$)*
2. $\inf_I = \mathcal{T}_\gamma \;\Rightarrow\; \inf_J = \mathcal{T}_\gamma$ *(for all $\gamma \leq \alpha$)*
3. $\inf_I = w \;\Leftrightarrow\; \inf_J = w$ *(for all w provided that $\deg(w) < \alpha$)*
4. $\sup_J = \mathcal{F}_\gamma \;\Rightarrow\; \sup_I = \mathcal{F}_\gamma$ *(for all $\gamma \leq \alpha$)*
5. $\sup_I = \mathcal{T}_\gamma \;\Rightarrow\; \sup_J = \mathcal{T}_\gamma$ *(for all $\gamma \leq \alpha$)*
6. $\sup_I = w \;\Leftrightarrow\; \sup_J = w$ *(for all w provided that $\deg(w) < \alpha$)*

Proof. 1.: Assume that $\inf_J = \mathcal{F}_\gamma$. Using Lemma 1 we get that there exists an i_0 such that $[\![A_{i_0}]\!]_{h_{i_0}}^J = \mathcal{F}_\gamma$. Thus, from the first part of $I_H(A_{i_0})$, $[\![A_{i_0}]\!]_{h_{i_0}}^I = \mathcal{F}_\gamma$. This implies that $[\![A_{i_0}]\!]_{h_{i_0}}^I \leq [\![A_i]\!]_{h_i}^I$ for all $i \in \mathcal{I}$. (Since otherwise we had that there exists a $j_0 \in \mathcal{I}$ such that $[\![A_{j_0}]\!]_{h_{j_0}}^I < \mathcal{F}_\gamma$. Hence, using the third part of $I_H(A_{j_0})$, we would get $[\![A_{j_0}]\!]_{h_{j_0}}^J < \mathcal{F}_\gamma$. But this contradicts our assumption $\inf_J = \mathcal{F}_\gamma$.) Finally, we get that $\inf_I = \mathcal{F}_\gamma$ must hold true.
2.: Assume now, that $\inf_I = \mathcal{T}_\gamma$. Then, for all $i \in \mathcal{I}$, $\mathcal{T}_\gamma \leq [\![A_i]\!]_{h_i}^I$. Using part two of $I_H(A_i)$, we get that $[\![A_i]\!]_{h_i}^I = [\![A_i]\!]_{h_i}^J$ for all i. This implies $\inf_J = \mathcal{T}_\gamma$.
3.: Due to 1. and 2., it only remains to show $(\inf_J = \mathcal{T}_\gamma \;\Rightarrow\; \inf_I = \mathcal{T}_\gamma)$ and $(\inf_I = \mathcal{F}_\gamma \;\Rightarrow\; \inf_J = \mathcal{F}_\gamma)$ for $\gamma < \alpha$. Assume that $\inf_J = \mathcal{T}_\gamma$ (where $\gamma < \alpha$). Then, for all $i \in \mathcal{I}$, $\mathcal{T}_\gamma \leq [\![A_i]\!]_{h_i}^J$ and this implies, using the third part of $I_H(A_i)$, $[\![A_i]\!]_{h_i}^J = [\![A_i]\!]_{h_i}^I$ for all i. Finally, we get that $\inf_I = \mathcal{T}_\gamma$.
For the latter case assume that $\inf_I = \mathcal{F}_\gamma$ $(\gamma < \alpha)$. Then there exists an i_0 such that $[\![A_{i_0}]\!]_{h_{i_0}}^I = \mathcal{F}_\gamma$ (Lemma 1). Thus, using the third part of $I_H(A_{i_0})$, we get that $[\![A_{i_0}]\!]_{h_{i_0}}^J = \mathcal{F}_\gamma$. This implies that $[\![A_{i_0}]\!]_{h_{i_0}}^J \leq [\![A_i]\!]_{h_i}^J$ for all $i \in \mathcal{I}$. (Since otherwise we had that there exists a $j_0 \in \mathcal{I}$ such that $[\![A_{j_0}]\!]_{h_{j_0}}^I < \mathcal{F}_\gamma$, see proof of statement 1.) Finally, we get that $\inf_J = \mathcal{F}_\gamma$.
We will not give the proofs of 4., 5., and 6. here, because they are similar to 1., 2., and 3.. □

Case 4: $\phi = A \wedge B$. Assume that $I_H(A)$ and $I_H(B)$. Let h be an arbitrary assumption. We define $\mathcal{I} := \{1, 2\}$, $h_1 := h$, $h_2 := h$, $A_1 := A$ and $A_2 := B$. Then $I_H(A_i)$ for $i = 1, 2$, $[\![\phi]\!]_h^J = \min\{[\![A]\!]_h^J, [\![B]\!]_h^J\} = \inf_J$ and $[\![\phi]\!]_h^I = \min\{[\![A]\!]_h^I, [\![B]\!]_h^I\} = \inf_I$. Thus, using 1., 2. and 3. of Lemma 2, we get that 1., 2. and 3. of Theorem 1 hold.

Case 5: $\phi = A \vee B$. Replace min by max and inf by sup in the proof above and use 4., 5. and 6. of Lemma 2 instead of 1., 2. and 3..
Case 6: $\phi = \forall v\, A$. Assume that $I_H(A)$ and let h be an arbitrary assumption.

We define $\mathcal{I} := \{u;\ u \in H_U\}$, $h_u := h[v \mapsto u]$ and $A_u := A$ for all $u \in H_U$. Then $I_H(A_u)$ for all $u \in \mathcal{I}$, $[\![\phi]\!]_h^J = \inf\{[\![A]\!]_{h[v\mapsto u]}^J;\ u \in H_U\} = \inf_J$, and $[\![\phi]\!]_h^I = \inf\{[\![A]\!]_{h[v\mapsto u]}^I;\ u \in H_U\} = \inf_I$. Thus, using 1., 2. and 3. of Lemma 2, we get that 1., 2. and 3. of Theorem 1 hold.

Case 7: $\phi = \exists v\, A$. Replace inf by sup in the proof above and use 4., 5. and 6. of Lemma 2 instead of 1., 2. and 3.. □

Lemma 3. *The immediate consequence operator T_P of a given program P is α-monotonic for all countable ordinals α.*

Proof. The proof is by transfinite induction on α. Assume the lemma holds for all $\beta < \alpha$ (induction hypothesis). We demonstrate that it also holds for α. Let I, J be two interpretations such that $I \sqsubseteq_\alpha J$. Thus, using the induction hypothesis, we get that

$$T_P(I) =_\beta T_P(J) \text{ for all } \beta < \alpha. \tag{1}$$

It remains to show that $T_P(I) \parallel \mathcal{T}_\alpha \subseteq T_P(J) \parallel \mathcal{T}_\alpha$ and that $T_P(J) \parallel \mathcal{F}_\alpha \subseteq T_P(I) \parallel \mathcal{F}_\alpha$. For the first statement assume that $T_P(I)(A) = \mathcal{T}_\alpha$ for some $A \in H_B$. Hence, using Lemma 1, there exists a ground instance $A \leftarrow \phi$ of P such that $[\![\phi]\!]^I = \mathcal{T}_\alpha$. But then, by Theorem 1, $[\![\phi]\!]^J = \mathcal{T}_\alpha$. This implies $\mathcal{T}_\alpha \leq T_P(J)(A)$. But this implies $\mathcal{T}_\alpha = T_P(J)(A)$. (Since $\mathcal{T}_\alpha < T_P(J)(A)$, using (1), would imply $\mathcal{T}_\alpha < T_P(I)(A)$.) For the latter statement assume that $T_P(J)(A) = \mathcal{F}_\alpha$ for some $A \in H_B$. This implies that $[\![\phi]\!]^J \leq \mathcal{F}_\alpha$ for every ground instance $A \leftarrow \phi$ of P. But then, using again Theorem 1, we get that $[\![\phi]\!]^I = [\![\phi]\!]^J$ for every ground instance $A \rightarrow \phi$ of P. Finally, this implies also $T_P(I)(A) = \mathcal{F}_\alpha$. □

Remark 3. The immediate consequence operator T_P is not monotonic with respect to $\sqsubseteq_\infty$. Consider the program $P = \{A \leftarrow \neg A\}$ and the interpretations I_1 and I_2 given by $I_1 := \{A \mapsto \mathcal{F}_0\}$ and $I_2 := \{A \mapsto 0\}$. Obviously, $I_1 \sqsubset_0 I_2$ and hence $I_1 \sqsubseteq_\infty I_2$. Using Definition 18, we get that $T_P(I_1) = sup\{[\![\neg A]\!]^{I_1}\} = \mathcal{T}_1$ and $T_P(I_2) = sup\{[\![\neg A]\!]^{I_2}\} = 0$. This implies $T_P(I_2) \sqsubset_1 T_P(I_1)$ (i.e., $T_P(I_1) \sqsubseteq_\infty T_P(I_2)$ does not hold).

4 Construction of the Least Model

In this section we show how to construct the interpretation M_P of a given formula-based logic program P. We will give the proof that M_P is a model of P and that it is the least of all models of P in the next section. In [3] the authors give a clear informal description of the following construction:

"As a first approximation to M_P, we start (...) iterating the T_P on $\emptyset$ until both the set of atoms that have a $\mathcal{F}_0$ value and the set of atoms having $\mathcal{T}_0$ value, stabilize. We keep all these atoms whose values have stabilized and reset the values of all remaining atoms to the next false value (namely $\mathcal{F}_1$). The procedure is repeated until the $\mathcal{F}_1$ and $\mathcal{T}_1$ values stabilize, and we reset the remaining atoms to a value equal to $\mathcal{F}_2$, and so on. Since the Herbrand Base of P is countable, there exists a countable ordinal δ for which this process will not produce any new atoms having $\mathcal{F}_\delta$ or $\mathcal{T}_\delta$ values. At this point we stop iteration and reset all remaining atoms to the value 0."

Definition 20. Let P be a program, I an interpretation, and $\alpha \in \aleph_1$ such that $I \sqsubseteq_\alpha T_P(I)$. We define by recursion on the ordinal $\beta \in \Omega$ the interpretation $T^\beta_{P,\alpha}(I)$ as follows:
$T^0_{P,\alpha}(I) := I$ and if β is a successor ordinal, then $T^\beta_{P,\alpha} := T_P(T^{\beta-1}_{P,\alpha})$. If $0 < \beta$ is a limit ordinal and $A \in H_B$, then

$$T^\beta_{P,\alpha}(I)(A) := \begin{cases} I(A), & \text{if } \deg(I(A)) < \alpha \\ \mathcal{T}_\alpha, & \text{if } A \in \bigcup_{\gamma \in \beta} T^\gamma_{P,\alpha}(I) \| \mathcal{T}_\alpha \\ \mathcal{F}_\alpha, & \text{if } A \in \bigcap_{\gamma \in \beta} T^\gamma_{P,\alpha}(I) \| \mathcal{F}_\alpha \\ \mathcal{F}_{\alpha+1}, & \text{otherwise} \end{cases}.$$

Lemma 4. *Let P be a program, I an interpretation and $\alpha \in \aleph_1$ such that $I \sqsubseteq_\alpha T_P(I)$. Then the following statements hold:*

1. *For all limit ordinals $0 < \gamma \in \Omega$ and all interpretations M the condition $\forall \beta < \gamma : T^\beta_{P,\alpha}(I) \sqsubseteq_\alpha M$ implies $T^\gamma_{P,\alpha}(I) \sqsubseteq_\alpha M$.*
2. *For all $\beta \leq \gamma \in \Omega$ the property $T^\beta_{P,\alpha}(I) \sqsubseteq_\alpha T^\gamma_{P,\alpha}(I)$ holds.*

Proof. **1.**: The proof follows directly from the above definition.
2.: One can prove the second statement with induction, using the assumption $I \sqsubseteq_\alpha T_P(I)$, the fact that T_P is α-monotonic, the fact that $\sqsubseteq_\alpha$ is transitive and at limit stage the first statement of this lemma. □

At this point, we have to consider a theorem of Zermelo-Fraenkel axiomatic set theory with the Axiom of Choice (ZFC). In the case of normal logic programs this theorem is not necessary, because in the bodies of normal logic programs do not appear "$\forall$" or "$\exists$". One can find the proof of the theorem in [1].

Definition 21. Let $\alpha > 0$ be a limit ordinal. We say that an increasing β-sequence $(\alpha_\zeta)_{\zeta<\beta}$, β limit ordinal, is *cofinal* in α if $\sup\{\alpha_\zeta; \zeta < \beta\} = \alpha$. Similarly, $A \subseteq \alpha$ is *cofinal* in α if $\sup A = \alpha$. If α is an infinite limit ordinal, the *cofinality* of α is $cf(\alpha)$ = "the least limit ordinal β such that there is an increasing β-sequence $(\alpha_\zeta)_{\zeta<\beta}$ with $\sup\{\alpha_\zeta; \zeta < \beta\} = \alpha$". An infinite cardinal $\aleph_\alpha$ is *regular* if $cf(\aleph_\alpha) = \aleph_\alpha$.

Theorem 2. *Every cardinal of the form $\aleph_{\alpha+1}$ is regular. Particularly, $\aleph_1$ is regular.*

Theorem 3 (Extension Theorem II). *Let P be a program, I an interpretation, and $\alpha \in \aleph_1$ such that $I \sqsubseteq_\alpha T_P(I)$. Then for every formuöla $\phi \in Form$ and every assignment h the following hold:*

1. $[\![\phi]\!]_h^{T_{P,\alpha}^{\aleph_1}(I)} = [\![\phi]\!]_h^I$, *if* $\deg([\![\phi]\!]_h^I) < \alpha$ (C1)
2. $[\![\phi]\!]_h^{T_{P,\alpha}^{\aleph_1}(I)} = \mathcal{T}_\alpha$, *if* $[\![\phi]\!]_h^{T_{P,\alpha}^{i}(I)} = \mathcal{T}_\alpha$ *for some* $i \in \aleph_1$ (C2)
3. $[\![\phi]\!]_h^{T_{P,\alpha}^{\aleph_1}(I)} = \mathcal{F}_\alpha$, *if* $[\![\phi]\!]_h^{T_{P,\alpha}^{i}(I)} = \mathcal{F}_\alpha$ *for all* $i \in \aleph_1$ (C3)
4. $\mathcal{F}_\alpha < [\![\phi]\!]_h^{T_{P,\alpha}^{\aleph_1}(I)} < \mathcal{T}_\alpha$ $\Leftrightarrow$ *not* (C1) *and not* (C2) *and not* (C3)

Proof. 1. and 2.: We get this using Lemma 4 and Theorem 1.
3.: We show this by induction on ϕ. We define $I_i := T_{P,\alpha}^i(I)$ and $I_\infty := T_{P,\alpha}^{\aleph_1}(I)$. Moreover, we use $I_H(X)$ as an abbreviation for

"for all assignments g the property $\forall i \in \aleph_1([\![X]\!]_g^{I_i} = \mathcal{F}_\alpha) \Rightarrow [\![X]\!]_g^{I_\infty} = \mathcal{F}_\alpha$ holds".

Case 1: $\phi = P_k(t_1, ..., t_{s_k})$ or $= \top, \bot$. This follows directly from Definition 20 respectively from Definition 15.
Case 2: $\phi = \neg A$. Assuming $\forall i \in \aleph_1$: $[\![\phi]\!]_h^{I_i} = \mathcal{F}_\alpha$ we conclude $\forall i \in \aleph_1 : [\![A]\!]_h^{I_i} = \mathcal{T}_{\alpha-1}$. Thus, by Theorem 1, we get $[\![A]\!]_h^{I_\infty} = \mathcal{T}_{\alpha-1}$ and this implies $[\![\phi]\!]_h^{I_\infty} = \mathcal{F}_\alpha$.
Case 3: $\phi = A \wedge B$ or $\phi = A \vee B$. The following cases are more general than this case. Therefore, we will not give a proof here.
Case 4: $\phi = \exists v\, A$. We assume that $I_H(A)$ and for every $i \in \aleph_1$ we assume that $[\![\phi]\!]_h^{I_i} = \mathcal{F}_\alpha$. This implies $\sup\{[\![A]\!]_{h[v\mapsto u]}^I;\ u \in H_U\} = \mathcal{F}_\alpha$ as well as $\forall i \in \aleph_1 \forall u \in H_U : [\![A]\!]_{h[v\mapsto u]}^{I_i} \le \mathcal{F}_\alpha$. Now we show by case distinction that $\forall u \in H_U : [\![A]\!]_{h[v\mapsto u]}^{I_\infty} = [\![A]\!]_{h[v\mapsto u]}^I$ and this obviously implies $[\![\phi]\!]_h^{I_\infty} = \mathcal{F}_\alpha$. First we consider the case $[\![A]\!]_{h[v\mapsto u]}^I < \mathcal{F}_\alpha$. Hence, using Lemma 4 and Theorem 1, we get that $[\![A]\!]_{h[v\mapsto u]}^{I_\infty} = [\![A]\!]_{h[v\mapsto u]}^I$. At least, we consider the other case $[\![A]\!]_{h[v\mapsto u]}^I = \mathcal{F}_\alpha$. We know that $\forall i \in \aleph_1 : [\![A]\!]_{h[v\mapsto u]}^{I_i} \le \mathcal{F}_\alpha$. But this implies $\forall i \in \aleph_1 : [\![A]\!]_{h[v\mapsto u]}^{I_i} = \mathcal{F}_\alpha$, since $\exists i \in \aleph_1 : [\![A]\!]_{h[v\mapsto u]}^{I_i} < \mathcal{F}_\alpha$ would imply (using Lemma 4 and Theorem 1) $[\![A]\!]_{h[v\mapsto u]}^I < \mathcal{F}_\alpha$, which is a contradiction. Finally, we get, by $I_H(A)$, that $[\![A]\!]_h^{I_\infty} = \mathcal{F}_\alpha = [\![A]\!]_{h[v\mapsto u]}^I$.
Case 5: $\phi = \forall v\, A$. We assume that $I_H(A)$ and for every $i \in \aleph_1$ we assume that $[\![\phi]\!]_h^{I_i} = \mathcal{F}_\alpha$. Then $\forall i \in \aleph_1 : \inf\{[\![A]\!]_{h[v\mapsto u]}^{I_i};\ u \in H_U\} = \mathcal{F}_\alpha$. This implies, using Lemma 1, $\forall i \in \aleph_1 \exists u \in H_U : [\![A]\!]_{h[v\mapsto u]}^{I_i} = \mathcal{F}_\alpha$. Next we choose for every $i \in \aleph_1$ a term $u_i \in H_U$ with $[\![A]\!]_{h[v\mapsto u_i]}^{I_i} = \mathcal{F}_\alpha$ (Remark: We do not need the Axiom of Choice because H_U is countable). Thus, using Lemma 4 and Theorem 1, $\forall i \in \aleph_1 \forall j \le i \in \aleph_1 : [\![A]\!]_{h[v\mapsto u_i]}^{I_j} = \mathcal{F}_\alpha$. This implies that the mapping

$$\zeta : \{u_i; i \in \aleph_1\} \to \aleph_1 \cup \{\aleph_1\} : u \mapsto \begin{cases} \min\{j \in \aleph_1; [\![A]\!]_{h[v\mapsto u]}^{I_j} \neq \mathcal{F}_\alpha\}, & \text{if min exists} \\ \aleph_1, & \text{otherwise} \end{cases}$$

has the properties $\forall i \in \aleph_1 : \zeta(u_i) > i$ and $\sup\{\zeta(u_i); i \in \aleph_1\} = \aleph_1$. We assume now that $\forall u \in H_U \exists j \in \aleph_1 : [\![A]\!]_{h[v\mapsto u]}^{I_j} \neq \mathcal{F}_\alpha$. Then $\zeta(\{u_i; i \in \aleph_1\})$ is a countable subset of $\aleph_1$ and moreover cofinal in $\aleph_1$. But this is a contradiction to Theorem 2.

Therefore we know that there exists an atom $u^* \in H_U$ such that $\forall i \in \aleph_1$: $[\![A]\!]^{I_i}_{h[v\mapsto u^*]} = \mathcal{F}_\alpha$. Thus, using $I_H(A)$, we get that $[\![A]\!]^{I_\infty}_{h[v\mapsto u^*]} = \mathcal{F}_\alpha$. This implies $[\![\phi]\!]^{I_\infty}_h \leq \mathcal{F}_\alpha$ and finally, using $[\![\phi]\!]^{I}_h = \mathcal{F}_\alpha$, Lemma 4 and Theorem 1, we get that $[\![\phi]\!]^{I_\infty}_h = \mathcal{F}_\alpha$.

4.:"$\Rightarrow$": We prove this by the method of contrapositive. We assume that (C1) or (C2) or (C3). Hence, using 1., 2., and 3., we get that not($\mathcal{F}_\alpha < [\![\phi]\!]^{I_\infty}_h < \mathcal{T}_\alpha$) holds.
"$\Leftarrow$": We shall first consider the following Lemma.

Lemma 5. *Under the same conditions as in Theorem 3 for every formula $\phi \in$ Form and every assignment h the following must hold true:*

$$[\![\phi]\!]^{I_\infty}_h = \mathcal{T}_\alpha \quad \Rightarrow \quad [\![\phi]\!]^{I_i}_h = \mathcal{T}_\alpha \ \textit{for some}\ i \in \aleph_1$$

Proof. This proof is similar to the proof of Theorem 3 statement 3 (see Appendix). □

We prove "$\Leftarrow$" also by the method of contrapositive. We assume that $\mathcal{F}_\alpha < [\![\phi]\!]^{I_\infty}_h < \mathcal{T}_\alpha$ does not hold. We consider the three possible cases $\deg([\![\phi]\!]^{I_\infty}_h) < \alpha$, $[\![\phi]\!]^{I_\infty}_h = \mathcal{F}_\alpha$, and $[\![\phi]\!]^{I_\infty}_h = \mathcal{T}_\alpha$. Let us consider the first case (resp. the second case). Then, using Lemma 4 and Theorem 1, (C1) (resp. (C3)) holds. Now, we consider the latter case. Using Lemma 5 we get that (C2) holds. Finally, in every case (C1) or (C2) or (C3) holds. □

Definition 22. Let α be a countable ordinal and for every $\gamma < \alpha$ let I_γ be an interpretation such that $\forall \zeta \leq \gamma : I_\zeta =_\zeta I_\gamma$. Then the union of the interpretations I_γ ($\gamma < \alpha$) is a well-defined interpretation and given by the following definition:

$$\bigsqcup_{\gamma<\alpha} I_\gamma\ (A) := \begin{cases} \mathcal{F}_\zeta, & \text{if } \zeta < \alpha \ \&\ I_\zeta(A) = \mathcal{F}_\zeta \\ \mathcal{T}_\zeta, & \text{if } \zeta < \alpha \ \&\ I_\zeta(A) = \mathcal{T}_\zeta \\ \mathcal{F}_\alpha, & \text{otherwise} \end{cases} \qquad (A \in H_B)$$

Remark 4. Using $\forall \zeta \leq \gamma : I_\zeta =_\zeta I_\gamma$ it is easy to prove that the union $\bigsqcup_{\gamma<\alpha} I_\gamma$ is a well-defined interpretation. Particularly if $\alpha = 0$, then the union is equal to the interpretation that maps all atoms of H_B to the truth value $\mathcal{F}_0$. This interpretation is sometimes denoted by $\emptyset$.

Lemma 6. *Let P be a program, α be a countable ordinal and for all $\gamma < \alpha$ an interpretation I_γ is given such that $\forall \zeta < \gamma : I_\zeta =_\zeta I_\gamma$. Then the following holds:*

$$\forall \gamma < \alpha\, (I_\gamma \sqsubseteq_{\gamma+1} T_P(I_\gamma)) \ \Rightarrow\ \bigsqcup_{\gamma<\alpha} I_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} I_\gamma)$$

Proof. The proof is left to the reader (see appendix). □

Lemma 7. *Let P be a program, α a countable ordinal, and I an interpretation. Then the following holds:*

$$I \sqsubseteq_\alpha T_P(I) \ \Rightarrow\ T^{\aleph_1}_{P,\alpha}(I) \sqsubseteq_{\alpha+1} T_P(T^{\aleph_1}_{P,\alpha}(I))$$

Proof. The proof is left to the reader (see appendix). □

Definition 23. Let P be a program. We define by recursion on the countable ordinal α the approximant M_α of P as follows:

$$M_\alpha := \begin{cases} T^{\aleph_1}_{P,\alpha}(\bigsqcup_{\gamma<\alpha} M_\gamma), & \text{if } \begin{array}{l}\forall\gamma<\alpha\forall\zeta<\gamma\,(M_\zeta =_\zeta M_\gamma)\ \&\\ \bigsqcup_{\gamma<\alpha} M_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} M_\gamma)\end{array} \\ \emptyset, & \text{otherwise} \end{cases}$$

Theorem 4. *Let P be a program, then for all $\alpha \in \aleph_1$ the following holds:*

1. $\forall\gamma<\alpha\,(M_\gamma =_\gamma M_\alpha)$
2. $\bigsqcup_{\gamma<\alpha} M_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} M_\gamma)$
3. $M_\alpha = T^{\aleph_1}_{P,\alpha}(\bigsqcup_{\gamma<\alpha} M_\gamma)$
4. $M_\alpha \sqsubseteq_{\alpha+1} T_P(M_\alpha)$

Proof. We prove this by induction on α. We assume that the theorem holds for all $\beta<\alpha$ (induction hypothesis). We prove that it holds also for α. Using the induction hypothesis, we get that for every $\beta<\alpha$ the following properties hold $\forall\gamma<\beta : M_\gamma =_\gamma M_\beta$ as well as $M_\beta \sqsubseteq_{\beta+1} T_P(M_\beta)$. Hence, using Lemma 6, we get that $\bigsqcup_{\gamma<\alpha} M_\gamma \sqsubseteq_\alpha T_P(\bigsqcup_{\gamma<\alpha} M_\gamma)$ (this is 2.). This together with the above definition imply $M_\alpha = T^{\aleph_1}_{P,\alpha}(\bigsqcup_{\gamma<\alpha} M_\gamma)$ (this is 3.). Thus, using 2. and 3. and Lemma 7, we get that $M_\alpha \sqsubseteq_{\alpha+1} T_P(M_\alpha)$ (this is 4.). It remains to prove the first statement. We know that for all $\gamma<\alpha$ the property $M_\gamma =_\gamma \bigsqcup_{\gamma'<\alpha} M_{\gamma'} \sqsubseteq_\alpha^{(\text{Lemma 4 \& 2.})} T^{\aleph_1}_{P,\alpha}(\bigsqcup_{\gamma'<\alpha} M_{\gamma'}) =^{3.} M_\alpha$ holds. Hence, using that $\sqsubseteq_\alpha$ is stronger than $=_\gamma$, we get that 1. also holds. □

Lemma 8. *Let P be a program. Then there exists an ordinal $\delta \in \aleph_1$ such that*

$$\forall\gamma \geq \delta : M_\gamma \| \mathcal{F}_\gamma = \emptyset \text{ and } M_\gamma \| \mathcal{T}_\gamma = \emptyset. \tag{2}$$

Proof. We define the subset H_B^* of the Herbrand base H_B by $H_B^* := \{A \in H_B;\ \exists\gamma \in \aleph_1 : M_\gamma(A) \in \{\mathcal{F}_\gamma, \mathcal{T}_\gamma\}\}$. Thus, using part one of Theorem 4, we know that for every $A \in H_B^*$ there is exactly one γ_A such that $M_{\gamma_A}(A) \in \{\mathcal{F}_{\gamma_A}, \mathcal{T}_{\gamma_A}\}$. Now let us define the function ζ by $\zeta : H_B^* \to \aleph_1 : A \mapsto \gamma_A$. We know that H_B^* is countable. This implies that $\zeta(H_B^*)$ is also countable. Hence, using Theorem 2, we know that $\zeta(H_B^*)$ is not cofinaöl in $\aleph_1$. This obviously implies that there is an ordinal $\delta \in \aleph_1$ such that $\forall A \in H_B^* : \zeta(A) < \delta$. Finally, this ordinal δ satisfies the property (2). □

Definition 24. Let P be a program. The lemma above justifies the definition $\delta_P := \min\{\delta;\ \forall\gamma \geq \delta : M_\gamma\|\mathcal{F}_\gamma = \emptyset \text{ and } M_\gamma\|\mathcal{T}_\gamma = \emptyset\} \in \aleph_1$. This ordinal δ_P is called the *depth* of the program P.

Definition 25. We define the interpretation M_P of a given formula-based logic program P by

$$M_P(A) := \begin{cases} M_{\delta_P}(A), & \text{if } \deg(M_{\delta_P}(A)) < \delta_P \\ 0, & \text{otherwise} \end{cases}.$$

5 Properties of the Interpretation M_P

Proposition 1. *Let P be a program. The interpretation M_P is a fixed point of T_P (i.e., $T_P(M_P) = M_P$).*

Proof. See Theorem 7.1 in [3]. □

Theorem 5. *Let P be a program. The interpretation M_P is a model of P.*

Proof. See Theorem 7.2 in [3]. □

Proposition 2. *Let P be a program, α a countable ordinal and M an arbitrary model of P. Then the following holds:*

$$\forall \beta < \alpha \, (M_\beta =_\beta M) \Rightarrow M_\alpha \sqsubseteq_\alpha M$$

Proof. We assume that $\forall \beta < \alpha \, (M_\beta =_\beta M)$. Definition 22 implies that

$$\bigsqcup_{\beta<\alpha} M_\beta \sqsubseteq_\alpha M \,. \tag{3}$$

Now we prove that the following holds:

$$T_P(M) \sqsubseteq_\alpha M \tag{4}$$

Therefore, using Lemma 3 and the assumption above, we get that the equation $\forall \beta < \alpha \, (T_P(M_\beta) =_\beta T_P(M))$ holds true. This the assumption above and the fourth part of Theorem 4 imply that $\forall \beta < \alpha : M =_\beta T_P(M)$. But this, together with the fact that M is a model (i.e., $M(A) \geq T_P(M)(A)$ holds for all atoms $A \in H_U$), implies that (4) holds.

We finish the proof by induction on the ordinal $\gamma \in \Omega$. Using Lemma 3 and (4), we get that $T^{\gamma}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$ implies $T^{\gamma+1}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$. Using the first part of Lemma 4, we get (for all limit ordinal γ) that $\forall \beta < \gamma : T^{\beta}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$ implies $T^{\gamma}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$. Thus, using (3) and statement 3. of Theorem 4, $M_\alpha = T^{\aleph_1}_{P,\alpha}(\bigsqcup_{\beta<\alpha} M_\beta) \sqsubseteq_\alpha M$ must hold true. □

Theorem 6. *The interpretation M_P of a given program P is the least of all models of P (i.e., for all models M of P the property $M_P \sqsubseteq_\infty M$ holds).*

Proof. Let M be an arbitrary model of P. Without loss of generality, we assume that $M \neq M_P$. Then let α be the least ordinal such that $M_P \neq_\alpha M$. This implies $\forall \beta < \alpha \, (M_\beta =_\beta M)$. Hence, using Proposition 2, $M_P =_\alpha M_\alpha \sqsubseteq_\alpha M$. The choice of α ensures that $M_P \neq_\alpha M$. Therefore we get that $M_P \sqsubset_\alpha M$ and this finally implies $M_P \sqsubseteq_\infty M$. □

Corollary 1. *Let P be a program. The interpretation M_P is the least of all fixed points of T_P.*

Proof. It is easy to prove that every fixed point of T_P is also a model of P. This together with Proposition 1 and Theorem 6 imply Corollary 1 □

Proposition 3. *There is a countable ordinal $\delta \in \aleph_1$ such that for all programs P of an arbitrary language $\mathcal{L}_{n,m,l,(s_i),(r_i)}$ the property $\delta_P < \delta$ holds. Let δ_{max} be the least ordinal such that the above property holds.*

Proof. We know that the set of all signatures $\langle n, m, l, (s_i)_{1 \leq i \leq n}, (r_i)_{1 \leq i \leq m} \rangle$ is countable. Additionally, we know that the set of all programs of a fixed signature is also countable (Remember that a program is a finite set of rules.). This implies that the set of all programs is countable. Hence we get that the image of the function from the set of all programs to $\aleph_1$ given by $P \mapsto \delta_P$ is countable. Thus, using Theorem 2, the image of $\delta_{(\cdot)}$ is not cofinal in $\aleph_1$ (i.e., there exists an ordinal $\delta \in \aleph_1$ such that for all programs P the property $\delta_P < \delta$ holds). □

Proposition 4. *The ordinal δ_{max} is at least ω^ω.*

Proof. Let $n > 0$ be a natural number. We consider the program P_n consisting of the following rules (where G, H are predicate symbols, f is a function symbol and c is a constant):

$G(x_1, ..., x_{n-1}, f(x_n)) \leftarrow \neg\neg G(x_1, ..., x_{n-1}, x_n)$
For all k provided that $1 \leq k \leq n-1$ the rule:
$G(x_1, ..., x_{k-1}, f(x_k), c, ..., c) \leftarrow \exists x_{k+1}, ..., x_n G(x_1, ..., x_{k-1}, x_k, x_{k+1}, ..., x_n)$
$H \leftarrow \exists x_1, ..., x_n G(x_1, ..., x_n)$

This implies that M_{P_n} maps $G\left(f^{k_1}(c), ..., f^{k_n}(c)\right)$ to $\mathcal{F}_{\sum_{m=1}^{n-1} \omega^{n-m} \bullet k_m + 2 \bullet k_n}$ and H to $\mathcal{F}_{\omega^n}$. □

At the end of this paper we will prove that the 3-valued interpretation $M_{P,3}$ that results from the infinite-valued model M_P by collapsing all true values to *True* (abbr. $\mathcal{T}$) and all false values to *False* (abbr. $\mathcal{F}$) is also a model in the sense of the following semantics:

Definition 26. *The semantics of formulas with respect to 3-valued interpretations is defined as in Definition 15 except that $[\![\top]\!]_h^I = \mathcal{T}$, $[\![\bot]\!]_h^I = \mathcal{F}$ and*

$$[\![\neg(\phi)]\!]_h^I = \begin{cases} \mathcal{T}, & \text{if } [\![\phi]\!]_h^I = \mathcal{F} \\ \mathcal{F}, & \text{if } [\![\phi]\!]_h^I = \mathcal{T} \\ 0, & \text{otherwise} \end{cases}.$$

The Definition 16 is also suitable in the case of 3-valued interpretations. The truth values are ordered as follows: $\mathcal{F} < 0 < \mathcal{T}$

Proposition 5. *Let P be a program and let collapse$(\cdot)$ be the function from W to the set $\{\mathcal{F}, 0, \mathcal{T}\}$ given by $\mathcal{F}_i \mapsto \mathcal{F}$, $0 \mapsto 0$ and $\mathcal{T}_i \mapsto \mathcal{T}$. Moreover, let I be an arbitrary interpretation and let collapse(I) be the 3-valued interpretation given by collapse$(I)(A) :=$ collapse$(I(A))$ (for all $A \in H_B$). Then for all formulas ϕ and all assignments h the following holds:*

$$collapse([\![\phi]\!]_h^I) = [\![\phi]\!]_h^{collapse(I)}$$

Proof. The proof is by induction on ϕ (see appendix). □

Proposition 6. *Let P be a formula-based logic program. Then the 3-valued interpretation $M_{P,3}$ is a 3-valued model of P.*

Proof. We assume that $A \leftarrow \phi$ is a rule of P. Thus, for every assignment h, we get that $[\![\phi]\!]_h^{M_{P,3}} \overset{\text{Proposition 5}}{=} \text{collapse}([\![\phi]\!]_h^{M_P}) \overset{\text{Theorem 6}}{\leq} \text{collapse}([\![A]\!]_h^{M_P}) = [\![A]\!]_h^{M_{P,3}}$ holds. □

Remark 5. The 3-valued model $M_{P,3}$ is not a minimal model in general. Consider the logic program $P = \{P_1 \leftarrow \neg\neg P_1\}$. Thus the infinite-valued model M_P maps P_1 to 0 and this implies $M_{P,3}(P_1) = 0$. But the (2-valued) interpretation $\{\langle P_1, \mathcal{F}\rangle\}$ is a model of P and it is less than $M_{P,3}$. The ordering on the 3-valued interpretations is introduced in [2] page 5.

However, Rondogiannis and Wadge prove in [3] that the 3-valued model $M_{P,3}$ of a given normal program P is equal to the 3-valued well-founded model of P and hence, using a result of Przymusinski (Theorem 3.1 of [2]), it is a minimal model of P. In the context of formula-based logic programs we can prove Theorem 7. Before we start with the proof we have to consider the following definition and a lemma that plays an important role in the proof of the theorem.

Definition 27. The *negation degree* $\deg_\neg(\phi)$ of a formula ϕ is defined recursively on the structure of ϕ as follows:

1. If ϕ is an atom, then $\deg_\neg(\phi) := 0$.
2. If $\phi = \psi_1 \circ \psi_2$, then $\deg_\neg(\phi) := \max\{\deg_\neg(\psi_1), \deg_\neg(\psi_2)\}$. ($\circ \in \{\vee, \wedge\}$)
3. If $\phi = \neg\psi$, then $\deg_\neg(\phi) := \deg_\neg(\psi) + 1$.
4. If $\phi = \boxdot x(\psi)$, then $\deg_\neg(\phi) := \deg_\neg(\psi)$. ($\boxdot \in \{\exists, \forall\}$)

Lemma 9. *Let I be an interpretation and $\gamma, \zeta \in \aleph_1$ such that for all $A \in H_B$ the following holds:*

$$I(A) \in [\mathcal{F}_0, \mathcal{F}_\gamma] \cup \{0\} \cup [\mathcal{T}_\zeta, \mathcal{T}_0]$$

Then for all formulas ϕ such that $deg_\neg(\phi) \leq 1$ and all variable assignments h the following holds:

$$[\![\phi]\!]_h^I \in \begin{cases} [\mathcal{F}_0, \mathcal{F}_\gamma] \cup \{0\} \cup [\mathcal{T}_\zeta, \mathcal{T}_0], & \textit{if } \deg_\neg(\phi) = 0 \\ [\mathcal{F}_0, \mathcal{F}_{\max\{\gamma,\zeta+1\}}] \cup \{0\} \cup [\mathcal{T}_{\max\{\gamma+1,\zeta\}}, \mathcal{T}_0], & \textit{otherwise} \end{cases}$$

Proof. The proof can be found in the appendix. □

Theorem 7. *Let P be a formula-based program such that for every rule $A \leftarrow \phi$ in P the property $deg_\neg(\phi) \leq 1$ holds. Then, the 3-valued model $M_{P,3}$ of the program P is a minimal 3-valued model.*

Proof. Let N_3 be an arbitrary 3-valued model of the program P, such that N_3 is smaller or equal to $M_{P,3}$. This is equivalent to

$$M_{P,3}\|\mathcal{F} \subseteq N_3\|\mathcal{F} \textit{ and } N_3\|\mathcal{T} \subseteq M_{P,3}\|\mathcal{T}. \tag{5}$$

Now we have to prove that N_3 is equal to $M_{P,3}$. Note that this holds if and only if both equations $M_{P,3}\|\mathcal{F} = N_3\|\mathcal{F}$ and $N_3\|\mathcal{T} = M_{P,3}\|\mathcal{T}$ hold.

Firstly, we prove that $N_3\|\mathcal{T} = M_{P,3}\|\mathcal{T}$ by contradiction. We assume that

$$M_{P,3}\|\mathcal{T} \setminus N_3\|\mathcal{T} \neq \emptyset. \tag{6}$$

We know that $M_{P,3}\|\mathcal{T} = \bigcup_{\alpha\in\aleph_1} M_P\|\mathcal{T}_\alpha$ and hence, using (6), there must be at least one ordinal $\alpha \in \aleph_1$ such that $M_P\|\mathcal{T}_\alpha \setminus N_3\|\mathcal{T} \neq \emptyset$. This justifies the definition $\alpha_{\min} := \min\{\alpha \in \aleph_1;\ M_P\|\mathcal{T}_\alpha \setminus N_3\|\mathcal{T} \neq \emptyset\}$. Using Theorem 4 we get that $M_{\alpha_{\min}} = T^{\aleph_1}_{P,\alpha_{\min}}(\bigsqcup_{\beta<\alpha_{\min}} M_\beta)$. To improve readability we define $J := \bigsqcup_{\beta<\alpha_{\min}} M_\beta$. It is obviously that $\alpha_{\min} < \delta_P$, and hence Definition 25, Theorem 4, and Definition 20 imply $M_P\|\mathcal{T}_{\alpha_{\min}} = M_{\delta_P}\|\mathcal{T}_{\alpha_{\min}} = M_{\alpha_{\min}}\|\mathcal{T}_{\alpha_{\min}} = \bigcup_{\gamma\in\aleph_1} T^{\gamma}_{P,\alpha_{\min}}(J)\|\mathcal{T}_{\alpha_{\min}}$. This and the definition of $\alpha_{\min}$ justify the definition $\gamma_{\min} := \min\{\gamma \in \aleph_1;\ T^{\gamma}_{P,\alpha_{\min}}(J)\|\mathcal{T}_{\alpha_{\min}} \setminus N_3\|\mathcal{T} \neq \emptyset\}$. From Definition 22 and Definition 20 we infer that $0 < \gamma_{\min}$ and $\gamma_{\min}$ is not an infinite limit ordinal and hence $\gamma_{\min}$ is a successor ordinal. We assume that $\gamma_{\min} = \gamma^{-}_{\min} + 1$. Thus, using the definition of $\alpha_{\min}$ and $\gamma_{\min}$, we get that $T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{T}_\zeta \subseteq N_3\|\mathcal{T}$ for all $\zeta \leq \alpha_{\min}$. Using statement (5) we infer that $T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_\zeta \subseteq N_3\|\mathcal{F}$ for all $\zeta < \alpha_{\min}$. Hence, the following definition of the infinite-valued interpretation N is well-defined: $(A \in H_B)$

$$N(A) := \begin{cases} \mathcal{F}_\zeta, & \text{if } \zeta < \alpha_{\min}\ \&\ A \in T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_\zeta \\ \mathcal{F}_{\alpha_{\min}}, & \text{if } A \in T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_{\alpha_{\min}} \cap N_3\|\mathcal{F} \\ \mathcal{F}_{\alpha_{\min}+1}, & \text{if } A \in N_3\|\mathcal{F} \setminus \bigcup\limits_{\zeta\leq\alpha_{\min}} T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{F}_\zeta \\ \mathcal{T}_\zeta, & \text{if } \zeta \leq \alpha_{\min}\ \&\ A \in T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{T}_\zeta \\ \mathcal{T}_{\alpha_{\min}+1}, & \text{if } A \in N_3\|\mathcal{T} \setminus \bigcup\limits_{\zeta\leq\alpha_{\min}} T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)\|\mathcal{T}_\zeta \\ 0, & \text{otherwise} \end{cases}$$

It is easy to see that

$$T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J) \sqsubseteq_{\alpha_{\min}} N \text{ and that } N_3 = \text{collapse}(N). \tag{7}$$

Since $T^{\gamma_{\min}}_{P,\alpha_{\min}}(J)\|\mathcal{T}_{\alpha_{\min}} \setminus N_3\|\mathcal{T}$ is not empty, we can pick an A that is contained in this set. Thus, together with Definition 18, we get that $\mathcal{T}_{\alpha_{\min}} = T^{\gamma_{\min}}_{P,\alpha_{\min}}(J)(A) = T_P(T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J))(A) = \sup\{[\![\phi]\!]_I;\ A \leftarrow \phi \in P_G\}$, where $I := T^{\gamma_{\min}-1}_{P,\alpha_{\min}}(J)$. Hence, using Lemma 1, we can pick a rule $A \leftarrow \phi \in P_G$ such that $[\![\phi]\!]_I = \mathcal{T}_{\alpha_{\min}}$. Thus, using statement (7), Theorem 1, and Proposition 5, we get that $[\![\phi]\!]_N = \mathcal{T}_{\alpha_{\min}}$ and $[\![\phi]\!]_{N_3} = [\![\phi]\!]_{\text{collapse}(N)} = \text{collapse}([\![\phi]\!]_N) = \mathcal{T}$. Lastly, the fact that N_3 is a model and $A \leftarrow \phi$ is a ground instance of P imply that $N_3(A) = \mathcal{T}$. But this is a contradiction because we have chosen A to be not contained in $N_3\|\mathcal{T}$. Hence, statement (6) must be wrong (i.e., $M_{P,3}\|\mathcal{T} = N_3\|\mathcal{T}$).

Secondly, we show that $M_{P,3}\|\mathcal{F} = N_3\|\mathcal{F}$. Definition 25 implies that $M_{P,3}\|\mathcal{F} = \bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{F}_\zeta$ and $M_{P,3}\|\mathcal{T} = \bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{T}_\zeta$. Thus, using (5) and the result of the first part of this proof, we get that $\bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{F}_\zeta \subseteq N_3\|\mathcal{F}$ and $\bigcup_{\zeta<\delta_P} M_{\delta_P}\|\mathcal{T}_\zeta = N_3\|\mathcal{T}$. Hence, the following definition of the infinite-valued interpretation N is well-defined and $N_3 = \text{collapse}(N)$.

$$N(A) := \begin{cases} \mathcal{F}_\zeta, & \text{if } \zeta < \delta_P \;\&\; A \in M_{\delta_P}\|\mathcal{F}_\zeta \\ \mathcal{F}_{\delta_P+1}, & \text{if } A \in N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F} \\ \mathcal{T}_\zeta, & \text{if } \zeta < \delta_P \;\&\; A \in M_{\delta_P}\|\mathcal{T}_\zeta \\ 0, & \text{otherwise} \end{cases} \qquad \text{(for all } A \in H_B\text{)}$$

Now we are going to prove by transfinite induction on $\zeta \in \aleph_1$ that $T^{\zeta}_{P,\delta_P+1}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$. Obviously, $T^{\zeta}_{P,\delta_P+1}(M_{\delta_P}) =_{\delta_P} N$ for all $\zeta \in \aleph_1$. The Definition of N, Definition 24, and Theorem 4 imply that $N\|\mathcal{T}_{\delta_P+1} = \emptyset = M_{\delta_P+1}\|\mathcal{T}_{\delta_p+1} = T^{\aleph_1}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{T}_{\delta_P+1} = \bigcup_{\gamma<\aleph_1} T^{\gamma}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{T}_{\delta_P+1}$ and hence we get that for all $\zeta \in \aleph_1$ the relation $T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{T}_{\delta_P+1} \subseteq N\|\mathcal{T}_{\delta_P+1}$ must hold. It remains to show that $N\|\mathcal{F}_{\delta_P+1} \subseteq T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$ for all $\zeta \in \aleph_1$.

Case 1: $\zeta = 0$. It is easy to prove (using Theorem 4, the result of the first part of this proof, and $N_3\|\mathcal{F} \cap N_3\|\mathcal{T} = \emptyset$) that $M_{\delta_P}\|\mathcal{F}_{\delta_P+1} = H_B \setminus (M_{P,3}\|\mathcal{F} \cup M_{P,3}\|\mathcal{T}) \supseteq N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F} = N\|\mathcal{F}_{\delta_P+1}$.

Case 2: ζ is a successor ordinal and $T^{\zeta-1}_{P,\delta_P+1}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$. Hence, using Definition 20 and Lemma 4, we get that

$$T_P(T^{\zeta-1}_{P,\delta_P+1}(M_{\delta_P})) = T^{\zeta}_{P,\delta_P+1}(M_{\delta_P}) \tag{8}$$

and

$$T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_p+1} \subseteq T^{\zeta-1}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}. \tag{9}$$

We will prove that $T^{\zeta-1}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} \setminus T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_p+1}$ and $N\|\mathcal{F}_{\delta_P+1}$ are disjoint. This, together with $T^{\zeta-1}_{P,\delta_P+1}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$ and statement (9), implies that $N\|\mathcal{F}_{\delta_P+1} \subseteq T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$ and we have proven this case. Now we choose an arbitrary $A \in T^{\zeta-1}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} \setminus T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_p+1}$. Hence, using Lemma 4, we get that $\mathcal{F}_{\delta_P+1} < T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})(A)$. This, together with (8) and Definition 18, implies that there must be a rule $A \leftarrow \phi \in P_G$ such that $\mathcal{F}_{\delta_P+1} < [\![\phi]\!]_I$, where I is given by $I := T^{\zeta-1}_{P,\delta_P+1}(M_{\delta_P})$. Thus, using the assumption $I \sqsubseteq_{\delta_P+1} N$ and Theorem 1, we get that $\mathcal{F}_{\delta_P+1} < [\![\phi]\!]_N$. We know that for all atoms $C \in H_B$ the image $N(C)$ is an element of $[F_0, F_{\delta_P+1}] \cup \{0\} \cup [T_{\delta_P}, T_0]$. But then Lemma 9 and the fact that $\deg_\neg(\phi) \leq 1$ imply $0 \leq [\![\phi]\!]_N$. Hence, using Proposition 5, $N_3 = \text{collapse}(N)$ and N_3 is a model of P, we get that $0 \leq [\![\phi]\!]_{N_3} \leq N_3(A)$. Finally, this implies $A \notin N_3\|\mathcal{F} \supseteq N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F} = N\|\mathcal{F}_{\delta_p+1}$.

Case 3: $\zeta > 0$ is a limit ordinal and $T^{\gamma}_{P,\delta_P+1}(M_{\delta_P}) \sqsubseteq_{\delta_P+1} N$ for all $\gamma < \zeta$. This implies $N\|\mathcal{F}_{\delta_P+1} \subseteq T^{\gamma}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$ for all $\gamma < \zeta$. Hence, using Definition 20, we get that $T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} = \bigcap_{\gamma\in\zeta} T^{\gamma}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1} \supseteq N\|\mathcal{F}_{\delta_P+1}$.

The above induction proves that $N\|\mathcal{F}_{\delta_P+1} \subseteq \bigcap_{\zeta\in\aleph_1} T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$ holds true. Thus, using that both the equation $M_{\delta_P+1}\|\mathcal{F}_{\delta_P+1} = \emptyset$ and $M_{\delta_P+1}\|\mathcal{F}_{\delta_P+1} = = \bigcap_{\zeta\in\aleph_1} T^{\zeta}_{P,\delta_P+1}(M_{\delta_P})\|\mathcal{F}_{\delta_P+1}$, we get that $\emptyset = N\|\mathcal{F}_{\delta_P+1} = N_3\|\mathcal{F} \setminus M_{P,3}\|\mathcal{F}$ (see definition of N above). Last of all, using the assumption (5), we get that $M_{P,3}\|\mathcal{F} = N_3\|\mathcal{F}$. □

Corollary 2. *If P is a saturated formula-based program (i.e., $M_P(A) \neq 0$ for all $A \in H_B$), then $M_{P,3}$ is a minimal model of P.*

Proof. A simple conclusion of the first part of the above proof. □

6 Summary and Future Work

We have shown that every formula-based logic program P has a least infinite-valued model M_P with respect to the ordering $\sqsubseteq_\infty$ given on the set of all infinite-valued interpretations. We have presented how to construct the model M_P with the help of the immediate consequence operator T_P and have shown that M_P is also the least of all fixed points of the operator T_P. Moreover, we have considered the 3-valued interpretation $M_{P,3}$ and have proven that it is a 3-valued model of the program P. Furthermore, we have observed a restricted class of formula-based programs such that the associated 3-valued models are even minimal models. There are some aspects of this paper that we feel should be further investigated. Firstly, we believe that the main results of this work also hold in Zermelo-Fraenkel axiomatic set theory without the Axiom of Choice (ZF). For instance, we could use the class of all ordinals Ω instead of the cardinal $\aleph_1$ in Theorem 3. Secondly, we have proven that the ordinal $\delta_{\max}$ is at least ω^ω, but on the other hand we do not know a program P such that $\omega^\omega < \delta_P$. So, one could assume that $\delta_{\max} = \omega^\omega$. Lastly, the negation-as-failure rule is sound for M_P (respectively, $M_{P,3}$) when we are dealing with a normal program P. Within the context of formula-based programs we think it would be fruitful to investigate the rule of definitional reflection presented in [4] instead of negation-as-failure.

Acknowledgements. This work has been financed through a grant made available by the Carl Zeiss Foundation. The author is grateful to Prof. Dr. Peter Schroeder-Heister, Hans-Joerg Ant, M. Comp. Sc., and three anonymous reviewers for helpful comments and suggestions.

References

1. Jech, T.: Set Theory. The Third Millennium Edition, Revised and Expanded. Springer, Heidelberg (2002)
2. Przymusinski, T.: Every logic program has a natural stratification and an iterated least fixed point model. In: Eighth ACM Symposium on Principles of Database Systems, pp. 11–21 (1989)
3. Rondogiannis, P., Wadge, W.: Minimum model semantics for logic programs with negation-as-failure. ACM Trans. Comput. Logic **6**(2), 441–467 (2005)
4. Schroeder-Heister, P.: Rules of definitional reflection. In: Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (Montreal 1993), Los Alamitos, pp. 222–232 (1993)

WLP Technical Papers I: Constraints and Logic Programming

A Declarative Approach for Computing Ordinal Conditional Functions Using Constraint Logic Programming

Christoph Beierle[1(✉)], Gabriele Kern-Isberner[2], and Karl Södler[1]

[1] Department of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
christoph.beierle@fernuni-hagen.de

[2] Department of Computer Science, TU Dortmund, 44221 Dortmund, Germany

Abstract. In order to give appropriate semantics to qualitative conditionals of the form *if A then normally B*, ordinal conditional functions (OCFs) ranking the possible worlds according to their degree of plausibility can be used. An OCF accepting all conditionals of a knowledge base R can be characterized as the solution of a constraint satisfaction problem. We present a high-level, declarative approach using constraint logic programming (CLP) techniques for solving this constraint satisfaction problem. In particular, the approach developed here supports the generation of all minimal solutions; this also holds for different notions of minimality which we discuss and implement in CLP. Minimal solutions are of special interest as they provide a basis for model-based inference from R.

1 Introduction

In knowledge representation, rules play a prominent role. Default rules of the form *If A then normally B* are being investigated in nonmonotonic reasoning, and various semantical approaches have been proposed for such rules. Since it is not possible to assign a simple Boolean truth value to such default rules, a semantical approach is to define when a rational agent accepts such a rule. We could say that an agent accepts the rule *Birds normally fly* if she considers a world with a flying bird to be less surprising than a world with a nonflying bird. At the same time, the agent can also accept the rule *Penguin birds normally do not fly*; this is the case if she considers a world with a nonflying penguin bird to be less surprising than a world with a flying penguin bird.

The informal notions just used can be made precise by formalizing the underlying concepts like default rules, epistemic state of an agent, and the acceptance relation between epistemic states and default rules. In the following, we deal with qualitative default rules and a corresponding semantics modelling the epistemic state of an agent. While a full epistemic state could compare possible worlds

The research reported here was partially supported by the Deutsche Forschungsgemeinschaft – DFG (grants BE 1700/7-2 and KE 1413/2-2).

according to their possibility, their probability, their degree of plausibility, etc. (cf. [9,10,18]), we will use ordinal conditional functions (OCFs), which are also called ranking functions [18]. To each possible world ω, an OCF κ assigns a natural number $\kappa(\omega)$ indicating its degree of surprise: The higher $\kappa(\omega)$, the greater is the surprise for observing ω.

In [12,13] a criterion when a ranking function respects the conditional structure of a set $\mathcal{R}$ of conditionals is defined, leading to the notion of c-representation for $\mathcal{R}$, and it is argued that ranking functions defined by c-representations are of particular interest for model-based inference. In [3] a system that computes a c-representation for any such $\mathcal{R}$ that is consistent is described, but this c-representation may not be minimal. An algorithm for computing a minimal ranking function is given in [5], but this algorithm fails to find all minimal ranking functions if there is more than one minimal one. In [15] an extension of that algorithm being able to compute all minimal c-representations for $\mathcal{R}$ is presented. The algorithm developed in [15] uses a non-declarative approach and is implemented in an imperative programming language. While the problem of specifying all c-representations for $\mathcal{R}$ is formalized as an abstract, problem-oriented constraint satisfaction problem $CR(\mathcal{R})$ in [2], no solving method is given there.

In this paper, we present a high-level, declarative approach using constraint logic programming techniques for solving the constraint satisfaction problem $CR(\mathcal{R})$ for any consistent $\mathcal{R}$. In particular, the approach developed here supports the generation of all minimal solutions; these minimal solutions are of special interest as they provide a preferred basis for model-based inference from $\mathcal{R}$. Moreover, we investigate different notions of minimality and demonstrate the flexibility of our approach by showing how alternative minimality concepts can be taken into account by slight modifications of the CLP implementation.

The rest of this paper is organized as follows: After recalling the formal background of conditional logics as it is given in [1] and as far as it is needed here (Sect. 2), we elaborate the birds-penguins scenario sketched above as an illustration for a conditional knowledge base and its semantics in Sect. 3. The definition of the constraint satisfaction problem $CR(\mathcal{R})$ and its solution set denoting all c-representations for $\mathcal{R}$ is given in Sect. 4. In Sect. 5, a declarative, high-level CLP program `GenOCF` solving $CR(\mathcal{R})$ is developed, observing the objective of being as close as possible to $CR(\mathcal{R})$. Its realization in Prolog is described in detail, as well as the modifications needed for alternative notions of minimality. In Sect. 6, `GenOCF` is evaluated with respect to a series of some first example applications. Section 7 concludes the paper and points out further work.

2 Background

We start with a propositional language $\mathcal{L}$, generated by a finite set Σ of atoms $a, b, c, \ldots$. The formulas of $\mathcal{L}$ will be denoted by uppercase Roman letters $A, B, C, \ldots$. For conciseness of notation, we will omit the logical *and*-connective, writing AB instead of $A \wedge B$, and overlining formulas will indicate negation, i.e.

$\overline{A}$ means $\neg A$. Let Ω denote the set of possible worlds over $\mathcal{L}$; Ω will be taken here simply as the set of all propositional interpretations over $\mathcal{L}$ and can be identified with the set of all complete conjunctions over Σ. For $\omega \in \Omega$, $\omega \models A$ means that the propositional formula $A \in \mathcal{L}$ holds in the possible world ω.

By introducing a new binary operator $|$, we obtain the set $(\mathcal{L} \mid \mathcal{L}) = \{(B|A) \mid A, B \in \mathcal{L}\}$ of *conditionals* over $\mathcal{L}$. $(B|A)$ formalizes "*if A then (normally) B*" and establishes a plausible, probable, possible etc. connection between the *antecedent* A and the *consequence* B. Here, conditionals are supposed not to be nested, that is, antecedent and consequent of a conditional will be propositional formulas.

A conditional $(B|A)$ is an object of a three-valued nature, partitioning the set of worlds Ω in three parts: those worlds satisfying AB, thus *verifying* the conditional, those worlds satisfying $A\overline{B}$, thus *falsifying* the conditional, and those worlds not fulfilling the premise A and so which the conditional may not be applied to at all. This allows us to represent $(B|A)$ as a *generalized indicator function* going back to [7] (where u stands for *unknown* or *indeterminate*):

$$(B|A)(\omega) = \begin{cases} 1 & \text{if } \omega \models AB \\ 0 & \text{if } \omega \models A\overline{B} \\ u & \text{if } \omega \models \overline{A} \end{cases} \tag{1}$$

To give appropriate semantics to conditionals, they are usually considered within richer structures such as *epistemic states.* Besides certain (logical) knowledge, epistemic states also allow the representation of preferences, beliefs, assumptions of an intelligent agent. Basically, an epistemic state allows one to compare formulas or worlds with respect to plausibility, possibility, necessity, probability, etc.

Well-known qualitative, ordinal approaches to represent epistemic states are Spohn's *ordinal conditional functions, OCFs*, (also called *ranking functions*) [18], and *possibility distributions* [4], assigning degrees of plausibility, or of possibility, respectively, to formulas and possible worlds. In such qualitative frameworks, a conditional $(B|A)$ is valid (or *accepted*), if its confirmation, AB, is more plausible, possible, etc. than its refutation, $A\overline{B}$; a suitable degree of acceptance is calculated from the degrees associated with AB and $A\overline{B}$.

In this paper, we consider Spohn's OCFs [18]. An OCF is a function

$$\kappa : \Omega \to \mathbb{N}$$

expressing degrees of plausibility of propositional formulas where a higher degree denotes "less plausible" or "more suprising". At least one world must be regarded as being normal; therefore, $\kappa(\omega) = 0$ for at least one $\omega \in \Omega$. Each such ranking function can be taken as the representation of a full epistemic state of an agent. Each such κ uniquely extends to a function (also denoted by κ) mapping sentences and rules to $\mathbb{N} \cup \{\infty\}$ and being defined by

$$\kappa(A) = \begin{cases} \min\{\kappa(\omega) \mid \omega \models A\} & \text{if } A \text{ is satisfiable} \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

for sentences $A \in \mathcal{L}$ and by

$$\kappa((B|A)) = \begin{cases} \kappa(AB) - \kappa(A) & \text{if } \kappa(A) \neq \infty \\ \infty & \text{otherwise} \end{cases} \tag{3}$$

for conditionals $(B|A) \in (\mathcal{L} \mid \mathcal{L})$. Note that $\kappa((B|A)) \geqslant 0$ since any ω satisfying AB also satisfies A and therefore $\kappa(AB) \geqslant \kappa(A)$.

The belief of an agent being in epistemic state κ with respect to a default rule $(B|A)$ is determined by the satisfaction relation $\models_{\mathcal{O}}$ defined by:

$$\kappa \models_{\mathcal{O}} (B|A) \text{ iff } \kappa(AB) \; < \; \kappa(A\overline{B}) \tag{4}$$

Thus, $(B|A)$ is believed in κ iff the rank of AB (verifying the conditional) is strictly smaller than the rank of $A\overline{B}$ (falsifying the conditional). We say that κ *accepts* the conditional $(B|A)$ iff $\kappa \models_{\mathcal{O}} (B|A)$.

3 Example

In order to illustrate the concepts presented in the previous section, we will use a scenario involving a set of some default rules representing common-sense knowledge.

Example 1. Suppose we have the propositional atoms
f - *flying*, b - *birds*, p - *penguins*, w - *winged* animals, k - *kiwis*.
Let the set $\mathcal{R}$ consist of the following conditionals:

$\mathcal{R}$	r_1:	$(f\|b)$	*birds fly*
	r_2:	$(b\|p)$	*penguins are birds*
	r_3:	$(\overline{f}\|p)$	*penguins do not fly*
	r_4:	$(w\|b)$	*birds have wings*
	r_5:	$(b\|k)$	*kiwis are birds*

Figure 1 shows a ranking function κ that accepts all conditionals given in $\mathcal{R}$. Thus, for any $i \in \{1, 2, 3, 4, 5\}$ it holds that $\kappa \models_{\mathcal{O}} R_i$.

For the conditional $(f|p)$ (*"Do penguins fly?"*) that is not contained in $\mathcal{R}$, we get $\kappa(pf) = 2$and $\kappa(p\overline{f}) = 1$ and therefore

$$\kappa \not\models_{\mathcal{O}} (f|p)$$

so that the conditional $(f|p)$ is not accepted by κ. This is in accordance with the behaviour of a rational agent believing $\mathcal{R}$ since the knowledge base $\mathcal{R}$ used for building up κ explicitly contains the opposite rule $(\overline{f}|p)$.

On the other hand, for the conditional $(w|k)$ (*"Do kiwis have wings?"*) that is also not contained in $\mathcal{R}$, we get $\kappa(kw) = 0$ and $\kappa(k\overline{w}) = 1$ and therefore

$$\kappa \models_{\mathcal{O}} (w|k)$$

i.e., the conditional $(w|k)$ is accepted by κ. Thus, from their superclass *birds*, kiwis inherit the property of having wings.

ω	$\kappa(\omega)$	ω	$\kappa(\omega)$	ω	$\kappa(\omega)$	ω	$\kappa(\omega)$
$pbfwk$	2	$p\overline{b}fwk$	5	$\overline{p}bfwk$	0	$\overline{p}\overline{b}fwk$	1
$pbfw\overline{k}$	2	$p\overline{b}fw\overline{k}$	4	$\overline{p}bfw\overline{k}$	0	$\overline{p}\overline{b}fw\overline{k}$	0
$pbf\overline{w}k$	3	$p\overline{b}f\overline{w}k$	5	$\overline{p}bf\overline{w}k$	1	$\overline{p}\overline{b}f\overline{w}k$	1
$pbf\overline{w}\overline{k}$	3	$p\overline{b}f\overline{w}\overline{k}$	4	$\overline{p}bf\overline{w}\overline{k}$	1	$\overline{p}\overline{b}f\overline{w}\overline{k}$	0
$pb\overline{f}wk$	1	$p\overline{b}\overline{f}wk$	3	$\overline{p}b\overline{f}wk$	1	$\overline{p}\overline{b}\overline{f}wk$	1
$pb\overline{f}w\overline{k}$	1	$p\overline{b}\overline{f}w\overline{k}$	2	$\overline{p}b\overline{f}w\overline{k}$	1	$\overline{p}\overline{b}\overline{f}w\overline{k}$	0
$pb\overline{f}\overline{w}k$	2	$p\overline{b}\overline{f}\overline{w}k$	3	$\overline{p}b\overline{f}\overline{w}k$	2	$\overline{p}\overline{b}\overline{f}\overline{w}k$	1
$pb\overline{f}\overline{w}\overline{k}$	2	$p\overline{b}\overline{f}\overline{w}\overline{k}$	2	$\overline{p}b\overline{f}\overline{w}\overline{k}$	2	$\overline{p}\overline{b}\overline{f}\overline{w}\overline{k}$	0

Fig. 1. Ranking function κ accepting the rule set $\mathcal{R}$ given in Example 1.

4 Specification of Ranking Functions as Solutions of a Constraint Satisfaction Problem

Given a set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of conditionals, a ranking function κ that accepts every R_i represents an epistemic state of an agent accepting $\mathcal{R}$. If there is no κ that accepts every R_i then $\mathcal{R}$ is *inconsistent*. For the rest of this paper, we assume that $\mathcal{R}$ is consistent.

For any consistent $\mathcal{R}$ there may be many different κ accepting $\mathcal{R}$, each representing a complete set of beliefs with respect to every possible formula A and every conditional $(B|A)$. Thus, every such κ inductively completes the knowledge given by $\mathcal{R}$, and it is a vital question whether some κ' is to be preferred to some other κ'', or whether there is a unique "best" κ. Different ways of determining a ranking function are given by *system Z* [9,10] or its more sophisticated extension *system Z** [9], see also [6]; for an approach using rational world rankings see [19]. For quantitative knowledge bases of the form $\mathcal{R}_x = \{(B_1|A_1)[x_1], \ldots, (B_n|A_n)[x_n]\}$ with probability values x_i and with models being probability distributions P satisfying a probabilistic conditional $(B_i|A_i)[x_i]$ iff $P(B_i|A_i) = x_i$, a unique model can be choosen by employing the principle of maximum entropy [11,16,17]; the maximum entropy model is a best model in the sense that it is the most unbiased one among all models satisfying $\mathcal{R}_x$.

Using the maximum entropy idea, in [13] a generalization of system Z* is suggested. Based on an algebraic treatment of conditionals, the notion of *conditional indifference* of κ with respect to $\mathcal{R}$ is defined and the following criterion for conditional indifference is given: An OCF κ is indifferent with respect to $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$ iff $\kappa(A_i) < \infty$ for all $i \in \{1, \ldots, n\}$ and there are rational numbers $\kappa_0, \kappa_i^+, \kappa_i^- \in \mathbb{Q}$, $1 \leqslant i \leqslant n$, such that for all $\omega \in \Omega$,

$$\kappa(\omega) = \kappa_0 + \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i B_i}} \kappa_i^+ + \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i \overline{B_i}}} \kappa_i^- . \tag{5}$$

When starting with an epistemic state of complete ignorance (i.e., each world ω has rank 0), for each rule $(B_i|A_i)$ the values κ_i^+, κ_i^- determine how the rank of each satisfying world and of each falsifying world, respectively, should be changed:

- If the world ω verifies the conditional $(B_i|A_i)$, – i.e., $\omega \models A_iB_i$ –, then κ_i^+ is used in the summation to obtain the value $\kappa(\omega)$.
- Likewise, if ω falsifies the conditional $(B_i|A_i)$, – i.e., $\omega \models A_i\overline{B_i}$ –, then κ_i^- is used in the summation instead.
- If the conditional $(B_i|A_i)$ is not applicable in ω, – i.e., $\omega \models \overline{A_i}$ –, then this conditional does not influence the value $\kappa(\omega)$.

κ_0 is a normalization constant ensuring that there is a smallest world rank 0. Employing the postulate that the ranks of a satisfying world should not be changed and requiring that changing the rank of a falsifying world may not result in an increase of the world's plausibility leads to the concept of a *c-representation* [12,13]:

Definition 1. *Let $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$. Any ranking function κ satisfying the conditional indifference condition (5) and $\kappa_i^+ = 0$, $\kappa_i^- \geqslant 0$ (and thus also $\kappa_0 = 0$ since $\mathcal{R}$ is assumed to be consistent) as well as*

$$\kappa(A_iB_i) < \kappa(A_i\overline{B_i}) \tag{6}$$

for all $i \in \{1, \ldots, n\}$ is called a (special) c-representation *of $\mathcal{R}$.*

Note that for $i \in \{1, \ldots, n\}$, condition (6) expresses that κ accepts the conditional $R_i = (B_i|A_i) \in \mathcal{R}$ (cf. the definition of the satisfaction relation in (4)) and that this also implies $\kappa(A_i) < \infty$.

Thus, finding a c-representation for $\mathcal{R}$ amounts to choosing appropriate values $\kappa_1^-, \ldots, \kappa_n^-$. In [2] this situation is formulated as a constraint satisfaction problem $CR(\mathcal{R})$ whose solutions are vectors of the form $(\kappa_1^-, \ldots, \kappa_n^-)$ determining c-representations of $\mathcal{R}$. The development of $CR(\mathcal{R})$ exploits (2) and (5) to reformulate (6) and requires that the κ_i^- are natural numbers (and not just rational numbers). In the following, we set $\min(\emptyset) = \infty$.

Definition 2. *[$CR(\mathcal{R})$] Let $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$. The constraint satisfaction problem for c-representations of $\mathcal{R}$, denoted by $CR(\mathcal{R})$, is given by the conjunction of the constraints*

$$\kappa_i^- \geqslant 0 \tag{7}$$

$$\kappa_i^- > \min_{\omega \models A_iB_i} \sum_{\substack{j \neq i \\ \omega \models A_j\overline{B_j}}} \kappa_j^- - \min_{\omega \models A_i\overline{B}_i} \sum_{\substack{j \neq i \\ \omega \models A_j\overline{B_j}}} \kappa_j^- \tag{8}$$

for all $i \in \{1, \ldots, n\}$.

A solution of $CR(\mathcal{R})$ is an n-tuple $(\kappa_1^-, \ldots, \kappa_n^-)$ of natural numbers, and with $Sol_{CR}(\mathcal{R})$ we denote the set of all solutions of $CR(\mathcal{R})$.

Proposition 1. *For $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$ let $\vec{\kappa} = (\kappa_1^-, \ldots, \kappa_n^-) \in Sol_{CR}(\mathcal{R})$. Then the function κ defined by*

$$\kappa(\omega) = \sum_{\substack{1 \leqslant i \leqslant n \\ \omega \models A_i \overline{B_i}}} \kappa_i^- \tag{9}$$

in the following denoted by $\kappa_{\vec{\kappa}}$, is an OCF that accepts $\mathcal{R}$.

All c-representations built from (7), (8), and (9) provide an excellent basis for model-based inference [12,13]. However, from the point of view of minimal specificity (see e.g. [4]), those c-representations with minimal κ_i^- yielding minimal degrees of implausibility are most interesting.

While different orderings on $Sol_{CR}(\mathcal{R})$ can be defined, leading to different minimality notions, in the following we will first focus on the ordering on $Sol_{CR}(\mathcal{R})$ induced by taking the sum of the κ_i^-, i.e.

$$(\kappa_1^-, \ldots, \kappa_n^-) \preccurlyeq_+ ({\kappa'}_1^-, \ldots, {\kappa'}_n^-) \quad \text{iff} \quad \sum_{1 \leqslant i \leqslant n} \kappa_i^- \leqslant \sum_{1 \leqslant i \leqslant n} {\kappa'}_i^- . \tag{10}$$

A vector $\vec{\kappa}$ is *sum-minimal* (just called *minimal* in the following) iff there is no vector $\vec{\kappa}'$ such that $\vec{\kappa}' \prec_+ \vec{\kappa}$ where $\prec_+$ is the irreflexive subrelation of $\preccurlyeq_+$. As we are interested in minimal κ_i^--vectors, an important question is whether there is always a unique minimal solution. This is not the case; the following example that is also discussed in [15] illustrates that $Sol_{CR}(\mathcal{R})$ may have more than one minimal element.

Example 2. Let $\mathcal{R}_{birds} = \{R_1, R_2, R_3\}$ be the following set of conditionals:

$R_1 : (f\|b)$	*$\underline{b}$irds $\underline{f}$ly*
$R_2 : (a\|b)$	*$\underline{b}$irds are $\underline{a}$nimals*
$R_3 : (a\|fb)$	*$\underline{f}$lying $\underline{b}$irds are $\underline{a}$nimals*

From (8) we get

$$\begin{aligned} \kappa_1^- &> 0 \\ \kappa_2^- &> 0 - min\{\kappa_1^-, \kappa_3^-\} \\ \kappa_3^- &> 0 - \kappa_2^- \end{aligned}$$

and since $\kappa_i^- \geqslant 0$ according to (7), the two vectors

$$\begin{aligned} sol_1 &= (\kappa_1^-, \kappa_2^-, \kappa_3^-) = (1, 1, 0) \\ sol_2 &= (\kappa_1^-, \kappa_2^-, \kappa_3^-) = (1, 0, 1) \end{aligned}$$

are two different solutions of $CR(\mathcal{R}_{birds})$ with $\sum_{1 \leqslant i \leqslant n} \kappa_i^- = 2$ that are both minimal in $Sol_{CR}(\mathcal{R}_{birds})$ with respect to $\preccurlyeq_+$.

5 A Declarative CLP Program for $CR(\mathcal{R})$

In this section, we will develop a CLP program `GenOCF` solving $CR(\mathcal{R})$. Our main objective is to obtain a declarative program that is as close as possible to the abstract formulation of $CR(\mathcal{R})$ while exploiting the concepts of constraint logic programming. We will employ finite domain constraints, and from (7) we immediately get a lower bound for κ_i^-. Considering that we are interested mainly in minimal solutions, due to (8) we can safely restrict ourselves to n as an upper bound for κ_i^-, yielding

$$0 \leqslant \kappa_i^- \leqslant n \tag{11}$$

for all $i \in \{1, \ldots, n\}$ with n being the number of conditionals in $\mathcal{R}$.

5.1 Input Format and Preliminaries

Since we want to focus on the constraint solving part, we do not consider reading and parsing a knowledge base $\mathcal{R} = \{(B_1|A_1), \ldots, (B_n|A_n)\}$. Instead, we assume that $\mathcal{R}$ is already given as a Prolog code file providing the following predicates `variables/1`, `conditional/3` and `indices/1`:

`variables`($[\mathtt{a}_1, \ldots, \mathtt{a}_m]$)	% list of atoms in Σ
`conditional`($i, \langle A_i\rangle, \langle B_i\rangle$)	% representation of ith conditional $(B_i\|A_i)$
`indices`($[1,\ldots,n]$)	% list of indices $\{1, \ldots, n\}$

If $\Sigma = \{a_1, \ldots, a_m\}$ is the set of atoms, we assume a fixed ordering $a_1 < a_2 < \ldots < a_m$ on Σ given by the predicate `variables`($[a_1,\ldots,a_m]$). The fixed index ordering gven by `indices`($[1,\ldots,n]$) ensures that the conditionals are ordered consecutively from 1 to n. Thus, the i-th conditional can be accessed by `conditional`(i, `A,B`), and in a solution vector $[\mathtt{K}_1,\ldots,\mathtt{K}_n]$, the i-th component $\mathtt{K_i}$ is the κ^--value for the i-th conditional.

In the representation of a conditional, a propositional formula A, constituting the antecedent or the consequence of the conditional, is represented by $\langle A\rangle$ where $\langle A\rangle$ is a Prolog list $[\langle D_1\rangle,\ldots,\langle D_l\rangle]$. Each $\langle D_i\rangle$ represents a conjunction of literals such that $D_1 \vee \ldots \vee D_l$ is a disjunctive normal form of A.

Each $\langle D\rangle$, representing a conjunction of literals, is a Prolog list $[b_1,\ldots,b_m]$ of fixed length m where m is the number of atoms in Σ and with $b_k \in \{\mathtt{0}, \mathtt{1}, \_\}$. Such a list $[b_1,\ldots,b_m]$ represents the conjunctions of atoms obtained from $\dot{a}_1 \wedge \dot{a}_2 \wedge \ldots \wedge \dot{a}_m$ by eliminating all occurrences of $\top$, where

$$\dot{a}_k = \begin{cases} a_k & \text{if} \ \ b_k = 1 \\ \overline{a_k} & \text{if} \ \ b_k = 0 \\ \top & \text{if} \ \ b_k = \_ \end{cases}$$

Example 3. The internal representation of the knowledge base presented in Example 1. is shown in Fig. 2.

```
variables([p,b,f,w,k]).

%                p b f w k    p b f w k
conditional(1,[[_,1,_,_,_]],[[_,_,1,_,_]]).  % (f|b)  birds fly
conditional(2,[[1,_,_,_,_]],[[_,1,_,_,_]]).  % (b|p)  penguins are birds
conditional(3,[[1,_,_,_,_]],[[_,_,0,_,_]]).  % (-f|p) penguins do not fly
conditional(4,[[_,1,_,_,_]],[[_,_,_,1,_]]).  % (w|b)  birds have wings
conditional(5,[[_,_,_,_,1]],[[_,1,_,_,_]]).  % (b|k)  kiwis are birds

indices([1,2,3,4,5]).
```

Fig. 2. Internal representation of the knowledge base from Example 1.

As further preliminaries, using `conditional/3` and `indices/1`, we have implemented the predicates `worlds/1`, `verifying_worlds/2`, `falsifying_worlds/2`, and `falsify/2`, realising the evaluation of the indicator function (1) given in Sect. 2:

`worlds`(Ws)	% *Ws* list of possible worlds
`verifying_worlds`(i, Ws)	% *Ws* list of worlds verifying ith conditional
`falsifying_worlds`(i, Ws)	% *Ws* list of worlds falsifying ith conditional
`falsify`(i, W)	% world W falsifies ith conditional

where worlds are represented as complete conjunctions of literals over Σ, using the representation described above.

Using these predicates, in the following subsections we will present the complete source code of the constraint logic program `GenOCF` solving $CR(\mathcal{R})$.

5.2 Generation of Constraints

The particular program code given here uses the SICStus Prolog system[1] and its clp(fd) library implementing constraint logic programming over finite domains [14].

The main predicate `kappa/2` expecting a knowledge base `KB` of conditionals and yielding a vector `K` of κ_i^- values as specified by (8) is presented in Fig. 3.

After reading in the knowledge base, the constraints for `K` are generated. In `constraints/1`, after getting the list of indices, a list `K` of free constraint variables, one for each conditional, is generated; in the two subsequent subgoals, the constraints corresponding to the formulas (11) and (8) are generated, constraining the elements of `K` accordingly. Finally, `labeling([], K)` yields a list of κ_i^- values. Upon backtracking, this will enumerate all possible solutions with an upper bound of n as in (11) for each κ_i^-. Later on, we will demonstrate how to modify `kappa/2` in order to take minimality into account (Sect. 5.3).

How the subgoal `constrain_K(Is, K)` in `kappa/2` generates a constraint for each index $i \in \{1, \ldots, n\}$ according to (8) is defined in Fig. 4.

[1] http://www.sics.se/isl/sicstuswww/site/index.html

```
kappa(KB, K) :-          % K is kappa vector of c-representation for KB
   consult(KB),
   constraints(K),       % generate constraints for K
   labeling([], K).      % generate solution

constraints(K) :-
   indices(Is),          % get list of indices [1,2,...,N]
   length(Is, N),        % N number of conditionals in KB
   length(K, N),         % generate K = [Kappa_1,...,Kappa_N] of free var.
   domain(K, 0, N),      % 0 <= kappa_I <= N  for all I according to (11)
   constrain_K(Is, K).   % generate constraints according to (8)
```

Fig. 3. Main predicate `kappa/2`

```
constrain_K([],_).                     % generate constraints for
constrain_K([I|Is],K) :-               % all kappa_I as in (8)
   constrain_Ki(I,K), constrain_K(Is,K).

constrain_Ki(I,K) :-             % generate constraint for kappa_I as in (8)
   verifying_worlds(I, VWorlds),      % all worlds verifying I-th condit.
   falsifying_worlds(I, FWorlds),     % all worlds falsifying I-th condit.
   list_of_sums(I, K, VWorlds, VS),   % VS list of sums for verif. worlds
   list_of_sums(I, K, FWorlds, FS),   % FS list of sums for falsif. worlds
   minimum(Vmin, VS),                 % Vmin minium for verifying worlds
   minimum(Fmin, FS),                 % Fmin minium for falsifying worlds
   element(I, K, Ki),                 % Ki constraint variable for kappa_I
   Ki #> Vmin - Fmin.                 % constraint for kappa_I as in (8)
```

Fig. 4. Constraining the vector `K` representing $\kappa_1^-, \dots, \kappa_n^-$ as in (8)

Given an index `I`, `constrain_Ki(I,K)` (cf. Fig. 4) determines all worlds verifying and falsifying the `I`-th conditional; over these two sets of worlds the two min expressions in (8) are defined. Two lists `VS` and `FS` of sums corresponding exactly to the first and the second sum, repectively, in (8) are generated (how this is done is defined in Fig. 5 and will be explained below). With the constraint variables `Vmin` and `Fmin` denoting the minimum of these two lists, the constraint

$$\texttt{Ki \#> Vmin} - \texttt{Fmin}$$

given in the last line of Fig. 4 reflects precisely the restriction on κ_i^- given by (8).

For an index `I`, a kappa vector `K`, and a list of worlds `Ws`, the goal `list_of_sums(I, K, Ws, Ss)` (cf. Fig. 5) yields a list `Ss` of sums such that for each world `W` in `Ws`, there is a sum `S` in `Ss` that is generated by `sum_kappa_j(Js, I, K, W, S)` where `Js` is the list of indices $\{1, \dots, n\}$. In the goal `sum_kappa_j (Js, I, K, W, S)`, `S` corresponds exactly to the respective sum expression in (8), i.e., it is the sum of all `Kj` such that `J` $\neq$ `I` and `W` falsifies the `j`-th conditional.

```
% list_of_sums(I, K, Ws, Ss)  generates list of sums as in (8):
%     I   index from 1,...,N
%     K   kappa vector
%     Ws  list of worlds
%     Ss  list of sums:
%         for each world W in Ws there is S in Ss s.t.
%            S is sum of all kappa_J with
%            J \= I and W falsifies J-th conditional
list_of_sums(_, _, [], []).
list_of_sums(I, K, [W|Ws], [S|Ss]) :-
   indices(Js),
   sum_kappa_j(Js, I, K, W, S),
   list_of_sums(I, K, Ws, Ss).

% sum_kappa_j(Js, I, K, W, S)  generates a sum as in (8):
%     Js  list of indices [1,...,N]
%     I   index from 1,...,N
%     K   kappa vector
%     W   world
%     S   sum of all kappa_J s.t.
%            J \= I and W falsifies J-th conditional
sum_kappa_j([], _, _, _, 0).
sum_kappa_j([J|Js], I, K, W, S) :-
    sum_kappa_j(Js, I, K, W, S1),
    element(J, K, Kj),
    ((J \= I, falsify(J, W)) -> S #= S1 + Kj; S #= S1).
```

Fig. 5. Generating list of sums of κ_i^- as in (8)

Example 4. Suppose that `kb_birds.pl` is a file containing the conditionals of the knowledge base $\mathcal{R}_{birds}$ given in Example 2.. Then the first five solutions generated by the program given in Figs. 3, 4, 5 are:

```
| ?- kappa('kb_birds.pl', K).
K = [0,1,1] ? ;
K = [1,0,2] ? ;
K = [1,0,3] ? ;
K = [1,1,0] ? ;
K = [1,1,1] ?
```

Note that the first and the fourth solution are the minimal solutions.

Example 5. If `kb_penguins.pl` is a file containing the conditionals of the knowledge base $\mathcal{R}$ given in Example 1., the first six solutions generated by `kappa/2` are:

```
| ?- kappa('kb_penguins.pl', K).
K = [1,2,2,1,1] ? ;
K = [1,2,2,1,2] ? ;
K = [1,2,2,1,3] ? ;
```

```
kappa_min_sum(KB, K) :-          % K is minimal vector for KB, one solution
   consult(KB),
   constraints(K),               % generate constraints for K
   sum(K, #=, S),                % constraint variable S for sum of kappa_I
   minimize(labeling([],K), S). % generate single minimal solution
```

Fig. 6. Predicate `kappa_min_sum/2` generating a minimal solution

```
kappa_min_sum_all(KB, K) :-     % K is minimal vector for KB, all solutions
   consult(KB),
   constraints(K),              % generate constraints for K
   sum(K, #=, S),               % constraint variable S for sum of kappa_I
   min_sum_kappas(K, S),        % determine minimal value for S
   labeling([], K).             % generate all minimal solutions

min_sum_kappas(K, Min) :-       % Min is sum of a minimal solution for K
   once((labeling([up],[Min]),
         \+ \+ labeling([],K))).
```

Fig. 7. Predicate `kappa_min_all/2` generating exactly all minimal solutions

```
K = [1,2,2,1,4] ? ;
K = [1,2,2,1,5] ? ;
K = [1,2,2,2,1] ?
```

5.3 Generation of Minimal Solutions

The enumeration predicate `labeling/2` of SICStus Prolog allows for an option that minimizes the value of a cost variable. Since we are aiming at minimizing the sum of all κ_i^-, the constraint `sum(K, #=, S)` introduces such a cost variable `S`. Thus, exploiting the SICStus Prolog minimization feature, we can easily modify `kappa/2` to generate a minimal solution: We just have to replace the last subgoal `labeling([], K)` in Fig. 3 by the two subgoals given in Fig. 6.

With this modification, the obtained predicate `kappa_min_sum/2` returns a single minimal solution (and fails on backtracking). Hence calling `?- kappa_min_sum('kb_birds.pl', K).` similar as in Example 4. yields the minimal solution `K = [0,1,1]`.

However, as pointed out in Sect. 4, there are good reasons for considering not just a single minimal solution, but all minimal solutions. We can achieve the computation of all minimal solutions by another slight modification of `kappa/2`. This time, the enumeration subgoal `labeling([], K)` in Fig. 3 is preceded by two new subgoals as in `kappa_min_sum_all/2` in Fig. 7.

The first new subgoal `sum(K, #=, S)` introduces a constraint variable `S` just as in `kappa_min_sum/2`. In the subgoal `min_sum_kappas(K, S)`, this variable `S` is constrained to the sum of a minimal solution as determined by `min_sum_kappas(K, Min)`. These two new subgoals ensure that in the generation caused by the final subgoal `labeling([], K)`, exactly all minimal solutions are enumerated.

```
rank(W, K, R) :-                   % the rank of world W under K is R
   findall(C, falsify(C,W), Cs),
   sum_kappa(Cs, K, R).            % determine R according to (9)

sum_kappa([], _, 0).               % sum of kappa-i-minus values for
sum_kappa([C|Cs], K, R) :-         % falsifying conditionals according
   sum_kappa(Cs, K, R1),           % to equation (9)
   element(C, K, Kj),
   R #= R1 + Kj.
```

Fig. 8. Predicate `rank/3` determining the rank of a world for given

Example 6. Continuing Example 4., calling

```
| ?- kappa_min_sum_all('kb_birds.pl', K).
K = [0,1,1] ? ;
K = [1,1,0] ? ;
no
```

yields the two minimal solutions for $\mathcal{R}_{birds}$.

Example 7. For the situation in Example 5., `kappa_min_sum_all/2` reveals that there is a unique minimal solution:

```
| ?- kappa_min_sum_all('kb_penguins.pl', K).
K = [1,2,2,1,1] ? ;
no
```

The predicate `rank/3` given in Fig. 8 can be used for determining the OCF $\kappa_{\vec{\kappa}}$ induced by the vector $\vec{\kappa} = (\kappa_1^-, \kappa_2^-, \kappa_3^-, \kappa_4^-, \kappa_5^-) = (1, 2, 2, 1, 1)$ according to (9), yielding the ranking function given in Fig. 1.

5.4 Alternative Notions of Minimality

In general, it is still an open problem how to strengthen the requirements defining a c-representation so that a unique minimal solution is guaranteed to exist. Such a strengthening could use alternative minimality criteria. In this subsection, we illustrate the realization of an alternative minimality criterion in our constraint logic programming approach.

Instead of ordering the vectors $\vec{\kappa}$ by the sum of their components as done by $\preccurlyeq_+$ in (10), we could consider a componentwise ordering $\preccurlyeq_{cw}$

$$(\kappa_1^-, \ldots, \kappa_n^-) \preccurlyeq_{cw} ({\kappa'}_1^-, \ldots, {\kappa'}_n^-) \quad \text{iff} \quad \kappa_i^- \leqslant {\kappa'}_i^- \text{ for all } \ i \in \{1, \ldots, n\} \quad (12)$$

yielding a partial order $\preccurlyeq_{cw}$ on $Sol_{CR}(\mathcal{R})$.

Let $\prec_{cw}$ denote the irreflexive subrelations of $\preccurlyeq_{cw}$, respectively. A vector $\vec{\kappa}$ is *componentwise minimal* (or *cw-minimal*) iff there is no vector $\vec{\kappa}'$ with $\vec{\kappa}' \prec_{cw} \vec{\kappa}$. In order to demonstrate the flexibility of the high-level CLP implementation

```
kappa_min_cw_all(KB, K) :-        % all cw-minimal solutions K for KB
   kappa(KB, K),                  % K is kappa vector for KB
   minimal_cw(K).                 % K is minimal componentwise

minimal_cw(K) :-                  % K is cw-minimal
   constraints(K2),               % if there is no kappa vector K2 that
   \+ once((lt_cw(K2, K),         % is componentwise strictly less than K
            labeling([],K2))).

lt_cw([K1|K1s], [K2|K2s]) :-      % one vector component in first
   (K1 #< K2, leq_cw(K1s, K2s));  % argument is strictly less than the
   (K1 #= K2, lt_cw(K1s, K2s)).   % corresponding component in second
                                  % argument, all other are less or equal

leq_cw([], []).                   % every vector component in
leq_cw([K1|K1s], [K2|K2s]) :-     % first argument less or equal to
   K1 #=< K2,                     % corresponding component in
   leq_cw(K1s, K2s).              % second argument
```

Fig. 9. Predicate `kappa_min_cw_all` computing all componentwise minimal solutions

realized in GenOCF, we will show how slight modifications of the program realize these alternative notions of minimality.

The predicate `kappa_min_cw_all/2` as given in Fig. 9 computes all componentwise minimal solution for a knowledge base. After consulting the knowledge base `KB`, the subgoal `kappa(KB, K)` says that `K` is a solution for `KB`, while `minimal_cw(K)` ensures that `K` is cw-minimal. The predicate `minimal_cw/1` enforces that there is no solution vector `K2` for the given knowledge base that is strictly less than `K`: If `constraints(K2)` succeeds where `constraints/1` is given as in Fig. 3, then there is no labeling for `K2` under the constraint `lt_cw(K2,K)`. The predicate `lt_cw/2` takes two vectors of the same length and succeeds if there is at least one position where the component at that position in the first vector is strictly less than the corresponding component in the second vector (ensured by the constraint with `#<` in `lt_cw/2` in Fig. 9), and all other corresponding vector components are less or equal (ensured by the constraints with `#=` in `lt_cw/2` and with `#=<` in `leq_cw/2`).

Example 8. Continuing Example 6., calling

```
| ?- kappa_min_cw_all('kb_birds.pl', K).
K = [1,0,1] ? ;
K = [1,1,0] ? ;
no
```

reveals that in this simple example the set of sum-minimal solutions coincides with the set of cw-minimal solutions.

In our further invstigations, we will extend GenOCF to be able to take into account more alternative minimality criteria. For instance, as illustrated in the previous example, GenOCF determines that both $\kappa_1 = [\mathtt{1,0,1}]$ and $\kappa_2 = [\mathtt{1,1,0}]$ are minimal with respect to $\preccurlyeq_+$ and also with respect to $\preccurlyeq_{cw}$ for $\mathcal{R}_{birds}$. However, we could also compare the full OCFs induced by κ_1 and κ_2 according to (9). These induced OCFs are given by the following table:

ω	$\kappa_1(\omega)$	$\kappa_2(\omega)$	ω	$\kappa_1(\omega)$	$\kappa_2(\omega)$
fba	0	0	$\overline{f}ba$	1	1
$fb\overline{a}$	1	1	$\overline{f}b\overline{a}$	1	2
$f\overline{b}a$	0	0	$\overline{f}\,\overline{b}a$	0	0
$f\overline{b}\overline{a}$	0	0	$\overline{f}\,\overline{b}\overline{a}$	0	0

Note that under κ_1, the world $\overline{f}b\overline{a}$ has a smaller rank than under κ_2, while all other worlds have the same rank under both OCFs. Further theoretical and experimental studies are needed for this and still other minimality criteria.

6 Example Applications and First Evaluation

Although the objective in developing GenOCF was on being as close as possible to the abstract formulation of the constraint satisfaction problem $CR(\mathcal{R})$, we will present the results of some first example applications we have carried out.

For $n \geqslant 1$, we generated synthetic knowledge bases kb_synth<n>_c<2n $-$ 1>.pl according to the following schema: Using the variables $\{f\} \cup \{a_1, \ldots, a_n\}$, kb_synth<$n$>_c<2n $-$ 1>.pl contains the $2 * n - 1$ conditionals given by::

$$\begin{array}{lll}
(f|a_i) & \text{if } i & \text{is odd, } i \in \{1, \ldots, n\} \\
(\overline{f}|a_i) & \text{if } i & \text{is even, } i \in \{1, \ldots, n\} \\
(a_i|a_{i+1}) & \text{if } i & \in \{1, \ldots, n-1\}
\end{array}$$

For instance, kb_synth4_c7.pl uses the five variables $\{f, a_1, a_2, a_3, a_4\}$ and contains the seven conditionals:

$$\begin{array}{c}
(f|a_1) \\
(\overline{f}|a_2) \\
(f|a_3) \\
(\overline{f}|a_4) \\
(a_1|a_2) \\
(a_2|a_3) \\
(a_3|a_4)
\end{array}$$

The basic idea underlying the construction of these synthetic knowledge bases kb_synth<n>_c<2n$-$1>.pl is to establish a kind of subclass relationship between a_{i+1} and a_i for each $i \in \{1, \ldots, n-1\}$ on the one hand, and to state that every a_{i+1} is exceptional to a_i with respect to its behaviour regarding f, again for each $i \in \{1, \ldots, n-1\}$. This sequence of pairwise exceptional elements will force any minimal solution of CR(kb_synth<n>_c<2n$-$1>.pl) to have at least one κ_i^- value of size greater or equal to n.

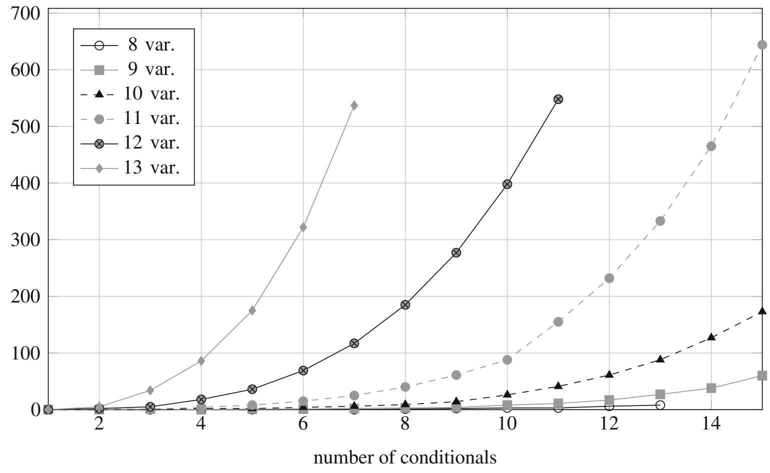


Fig. 10. Execution times (given in seconds) of `GenOCF` under SICStus Prolog for computing all sum-minimal solutions for various synthesized knowledge bases

From `kb_synth<`n`>_c<`m`>.pl`, the knowledge bases `kb_synth<`n`>_c<`m–j`>.pl` are generated for $j \in \{1, \ldots, m-1\}$ by removing the last j conditionals. For instance, `kb_synth4_c5.pl` is obtained from `kb_synth4_c7.pl` by removing the two conditionals $\{(a_2|a_3),\ (a_3|a_4)\}$.

Figure 10 shows the time needed by `GenOCF` for computing all sum-minimal solutions for some of these synthesized knowledge bases with different numbers of variables and conditionals. Execution times are given in seconds for measurements taken in the following environment: SICStus 4.0.8 (x86-linux-glibc2.3), Intel Core 2 Duo E6850 3.00GHz.

Of course, these knowledge bases are by no means representative, and further evaluation is needed. In particular, investigating the complexity depending on the number of variables and conditionals and determining an upper bound for worst-case complexity has still to be done; the graphical illustration in Fig. 10 clearly indicates an exponential increase. However, it should be noted that the high-level, declarative approach taken here provides many opportunities for improving run-time efficiency. For instance, it suffices to compute the verifying and the falsifying worlds for each conditional only once instead of multiple times when generating the constraints for a solution vector `K` as done in Fig. 4. Furthermore, while the code for `GenOCF` given above uses SICStus Prolog, we also have a variant of `GenOCF` for the SWI Prolog system[2] [20]. In our further investigations, we want to evaluate `GenOCF` also using SWI Prolog, to elaborate the changes required and the options provided when moving between SICStus and SWI Prolog, and to study whether there are any significant differences in execution that might depend on the two different Prolog systems and their options.

[2] http://www.swi-prolog.org/index.html

7 Conclusions and Further Work

While for a set of probabilistic conditionals $(B_i|A_i)[x_i]$ the principle of maximum entropy yields a unique model, for a set $\mathcal{R}$ of qualitative default rules $(B_i|A_i)$ there may be several minimal ranking functions. In this paper, we developed a CLP approach for solving $CR(\mathcal{R})$, realized in the Prolog program `GenOCF`. The solutions of the constraint satisfaction problem $CR(\mathcal{R})$ are vectors of natural numbers $\vec{\kappa} = (\kappa_1^-, \ldots, \kappa_n^-)$ that uniquely determine an OCF $\kappa_{\vec{\kappa}}$ accepting all conditionals in $\mathcal{R}$. `GenOCF` is also able to generate exactly all minimal solutions of $CR(\mathcal{R})$ for different notions of minimality. Minimal solutions of $CR(\mathcal{R})$ are of special interest for model-based inference.

In general, it is an open problem how to strengthen the requirements defining a c-representation so that a unique solution is guaranteed to exist. The declarative nature of constraint logic programming supports easy constraint modification, enabling the experimentation and practical evaluation of different notions of minimality for $Sol_{CR}(\mathcal{R})$ and of additional requirements that might be imposed on a ranking function. Furthermore, in [8] the framework of default rules concidered here is extended by allowing not only default rules in the knowledge base $\mathcal{R}$, but also strict knowledge, rendering some worlds completely impossibe. This can yield a reduction of the problem's complexity, and it will be interesting to see which effects the incorporation of strict knowledge will have on the CLP approach presented here.

References

1. Beierle, C., Kern-Isberner, G.: A verified AsmL implementation of belief revision. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 98–111. Springer, Heidelberg (2008)
2. Beierle, C., Kern-Isberner, G.: On the computation of ranking functions for default rules - a challenge for constraint programming. In: Heiß, H.-U., Pepper, P., Schlingloff, H., Schneider, J. (eds.) Informatik 2011: Informatik schafft Communities, Beiträge der 41. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 4.-7.10.2011, Berlin (Abstract Proceedings), volume P-192 of LNI. GI (2011)
3. Beierle, C., Kern-Isberner, G., Koch, N.: A high-level implementation of a system for automated reasoning with default rules (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 147–153. Springer, Heidelberg (2008)
4. Benferhat, S., Dubois, D., Prade, H.: Representing default rules in possibilistic logic. In: Proceedings 3th International Conference on Principles of Knowledge Representation and Reasoning KR'92, pp. 673–684 (1992)
5. Bourne, R.A.: Default reasoning using maximum entropy and variable strength defaults. Ph.D. thesis, Univ. of London (1999)
6. Bourne, R.A., Parsons, S.: Maximum entropy and variable strength defaults. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI'99, pp. 50–55 (1999)
7. DeFinetti, B.: Theory of Probability, vol. 1,2. Wiley, New York (1974)
8. Eiter, T., Lukasiewicz, T.: Complexity results for structure-based causality. Artif. Intell. **142**(1), 53–89 (2002)

9. Goldszmidt, M., Morris, P., Pearl, J.: A maximum entropy approach to non-monotonic reasoning. IEEE Trans. Pattern Anal. Mach. Intell. **15**(3), 220–232 (1993)
10. Goldszmidt, M., Pearl, J.: Qualitative probabilities for default reasoning, belief revision, and causal modeling. Artif. Intell. **84**, 57–112 (1996)
11. Kern-Isberner, G.: Characterizing the principle of minimum cross-entropy within a conditional-logical framework. Artif. Intell. **98**, 169–208 (1998)
12. Kern-Isberner, G.: Conditionals in nonmonotonic reasoning and belief revision. LNCS (LNAI), vol. 2087. Springer, Heidelberg (2001)
13. Kern-Isberner, G.: Handling conditionals adequately in uncertain reasoning and belief revision. J. Appl. Non-Class. Logics **12**(2), 215–237 (2002)
14. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P.H., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997)
15. Müller, C.: Implementing default rules by optimal conditional ranking functions. B.Sc. Thesis, Department of Computer Science, FernUniversität in Hagen, Germany (2004) (in German)
16. Paris, J.B.: The uncertain reasoner's companion - A mathematical perspective. Cambridge University Press, Cambridge (1994)
17. Paris, J.B., Vencovska, A.: In defence of the maximum entropy inference process. Int. J. Approximate Reasoning **17**(1), 77–103 (1997)
18. Spohn, W.: Ordinal conditional functions: a dynamic theory of epistemic states. In: Harper, W.L., Skyrms, B. (eds.) Causation in Decision, Belief Change, and Statistics, vol. II, pp. 105–134. Kluwer Academic Publishers, Dordrecht (1988)
19. Weydert, E.: System JZ - How to build a canonical ranking model of a default knowledge base. In: Proceedings KR'98. Morgan Kaufmann, San Mateo (1998)
20. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. CoRR, abs/1011.5332, (2010) (to appear in Theory and Practice of Logic Programming)

WLP Technical Papers II: Answer-Set Programming and Model Expansion

A Descriptive Approach to Preferred Answer Sets

Ján Šefránek[✉] and Alexander Šimko

Comenius University, Bratislava, Slovakia
{sefranek, simko}@fmph.uniba.sk

Abstract. We are aiming at a semantics of logic programs with preferences defined on rules, which always selects a preferred answer set, if there is a non-empty set of (standard) answer sets of the given program.

It is shown in a seminal paper by Brewka and Eiter that the goal mentioned above is incompatible with their second principle and it is not satisfied in their semantics of prioritized logic programs. Similarly, also according to other established semantics, based on a prescriptive approach, there are programs with standard answer sets, but without preferred answer sets.

According to the standard prescriptive approach no rule can be fired before a more preferred rule, unless the more preferred rule is blocked. This is a rather imperative approach, in its spirit. According to our background intuition, rules can be blocked by more preferred rules, but the rules which are not blocked are handled in a more declarative style, independent on the given preference relation on the rules.

An argumentation framework (different from Dung's framework) is proposed in this paper. Some argumentation structures are assigned to the rules of a given program. Other argumentation structures are derived using a set of derivation rules. Some of the derived argumentation structures correspond to answer sets. An attack relation on derivations of argumentation structures is defined. Preferred answer sets correspond to complete argumentation structures, which are not blocked by other complete argumentation structures.

Keywords: Extended logic program · Answer set · Preference · Preferred answer set · Argumentation structure

1 Introduction

The meaning of a nonmonotonic theory is often characterized by a set of *(alternative) belief sets*. It is appropriate to select sometimes some of the belief sets as more *preferred*.

We are focused in this paper on *extended logic programs* with a preference relation on rules, see, e.g., [1,2,10,18]. Such programs are denoted by the term *prioritized* logic programs in this paper.

It is suitable to require that some preferred answer sets can be *selected* from a non-empty set of the standard answer sets of a (prioritized) logic program.

Unfortunately, there are prioritized logic programs with standard answer sets, but without preferred answer sets according to the semantics of [1] and also of [2] or [18]. This feature is a consequence of the *prescriptive* approach to preference handling [4]. According to that approach no rule can be fired before a more preferred rule, unless the more preferred rule is blocked. This is a rather imperative approach, in its spirit.

An investigation of basic *principles* which should be satisfied by any system containing a preference relation on defeasible rules is of fundamental importance. This type of research has been initialized in the seminal paper [1]. Two basic principles are accepted in the paper.

The second of the principles is violated, if a function is assumed which always selects a non-empty subset of preferred answer sets from a non-empty set of all standard answer sets of a prioritized logic program.

We believe that the possibility to select always a preferred answer set from a non-empty set of standard answer sets is of critical importance. This goal requires to accept a *descriptive* approach to preference handling. The approach is characterized in [3,4] as follows: The order in which rules are applied, reflects their "desirability" – it is desirable to apply the most preferred rules.

Our basic intuition is that rules can be *blocked* by more preferred rules, but the rules which are not blocked are handled in a more declarative style. If we express this in terms of desirability, it is desirable *to apply all (applicable) rules which are not blocked by a more preferred rule.* The execution of non-blocked rules does not depend on their order. Dependencies of more preferred rules on less preferred rules do not prevent the execution of non-blocked rules. However, this approach is computationally more demanding than the prescriptive approach.

A formal elaboration of this intuition resulted in our approach into attack and blocking relations between sets of generating rules (expressed in terms of derivations of complete argumentation structures).

Our goal is:

– to modify the principles proposed by Brewka and Eiter in [1] in such a way that they do not contradict a selection of a non-empty set of preferred answer sets from the underlying non-empty set of standard answer sets, and
– to introduce a notion of preferred answer sets that satisfies the above mentioned modification.

The proposed method is inspired by [8]. While there defeasible rules are treated as (defeasible) arguments, here (defeasible) assumptions, sets of default negations, are considered as arguments. Reasoning about preferences in a logic program is here understood as a kind of argumentation. Sets of default literals can be viewed as defeasible arguments, which may be contradicted by consequences of some applicable rules. The preference relation on rules is used in order to ignore the attacks of less preferred arguments against more preferred arguments. The core problem is to transfer the preference relation defined on rules to a blocking relation between answer sets.[1]

The basic argumentation structures correspond to the rules of a given program. Derivation rules, which enable derivation of argumentation structures from the basic ones are defined. That derivation leads from the basic argumentation structures to argumentation structures corresponding to answer sets of the given program (we use a method of [5]). The argumentation structures, which correspond to answer sets, are called *complete* in this paper.

Derivations of complete argumentation structures play a crucial role in our approach. Attacks of more preferred rules against the less preferred rules are transferred to attacks between derivations of complete argumentation structures. Preferred answer sets are defined over that background. They correspond to complete and non-blocked (warranted) argumentation structures.

The contributions of this paper are summarized as follows. A modified set of principles for preferred answer set specification is proposed. A new type of argumentation framework is constructed, which enables a selection of preferred answer sets. There are basic arguments (argumentation structures) and attacks in the framework. Rules for derivation of argumentation structures are introduced. After that attacks between derivations of complete argumentation structures, acceptable derivations and, finally, warranted and blocked complete argumentation structures are defined. Preferred answer sets are defined in terms of complete and non-blocked (warranted) argumentation structures. Each program with a non-empty set of answer sets has a preferred answer set in our approach.

A preliminary version of the presented research has been published in [12]. This is more than an extended version of [13]. The main differences between the versions are summarized in the Conclusions.

2 Preliminaries

The language of extended logic programs is used in this paper.

Let At be a set of atoms. The set of *objective literals* is defined as $Obj = At \cup \{\neg A : A \in At\}$. If L is an objective literal then an expression of the form $not\ L$ is called *default* literal. Notation: $Def = \{not\ L \mid L \in Obj\}$. The set of literals Lit is defined as $Obj \cup Def$.

A convention: $\neg\neg A = A$, where $A \in At$. If X is a set of objective literals, then $not\ X = \{not\ L \mid L \in X\}$.

A *rule* is each expression of the form $L \leftarrow L_1, \ldots, L_k$, where $k \geq 0$, $L \in Obj$ and $L_i \in Lit$. If r is a rule of the form as above, then L is denoted by $head(r)$ and $\{L_1, \ldots, L_k\}$ by $body(r)$. If R is a set of rules, then $head(R) = \{head(r) \mid r \in R\}$ and $body(R) = \{body(r) \mid r \in R\}$. A finite set of rules is called *extended logic program* (program hereafter).

[1] Our intuitions connected to the notion of argumentation structure and also the used constructions are different from Dung's arguments or from arguments of [8]. This paper does not present a contribution to argumentation theory.

The set of *conflicting literals* is defined as $CON = \{(L_1, L_2) \mid L_1 = not\ L_2 \vee L_1 = \neg L_2\}$. A set of literals S is *consistent* if $(S \times S) \cap CON = \emptyset$. An *interpretation* is a consistent set of literals. A *total* interpretation is an interpretation I such that for each objective literal L either $L \in I$ or $not\ L \in I$. A literal L is *satisfied* in an interpretation I iff $L \in I$ (notation: $I \models L$). A set of literals S is satisfied in I iff $S \subseteq I$ (notation: $I \models S$). A rule r is satisfied in I iff $I \models head(r)$ whenever $I \models body(r)$, notation $I \models r$. An interpretation I is a *model* of a program P, notation $I \models P$, if for each $r \in P$ holds $I \models r$.

If S is a set of literals, then we denote $S \cap Obj$ by S^+ and $S \cap Def$ by S^-. Symbols $(body(r))^-$ and $(body(r))^+$ are used here in that sense (notice that the usual meaning of $body^-(r)$ is different). If $X \subseteq Def$ then $pos(X) = \{L \in Obj \mid not\ L \in X\}$. Hence, $not\ pos((body(r))^-) = (body(r))^-$. If r is a rule, then the rule $head(r) \leftarrow (body(r))^+$ is denoted by r^+.

An answer set of a program can be defined as follows (only consistent answer sets are defined).

A total interpretation S is an *answer set* of a program P iff S^+ is the least model[2] of the program $P^+ = \{r^+ \mid S \models (body(r))^-\}$. Note that an answer set S is usually represented by S^+ (this convention is sometimes used also in this paper).

The set of all answer sets of a program P is denoted by $AS(P)$. A program is called *coherent* iff it has an answer set.

A strict partial order is a binary relation, which is irreflexive, transitive and, consequently, asymmetric.

A *prioritized logic program* is defined in this paper as a pair $(P, \prec)$, where P is a program and $\prec$ a strict partial order on rules of P. Let be $r_1, r_2 \in P$. If $r_1 \prec r_2$ it is said that r_2 is more preferred than r_1.

3 Argumentation Structures

Our aim is to transfer a preference relation defined on rules to a preference relation on answer sets and, finally, to a notion of preferred answer sets. To that end argumentation structures are introduced. The basic argumentation structures correspond to rules. Some more general types of argumentation structures are derived from the basic argumentation structures. A special type of argumentation structures corresponds to answer sets.

Definition 1 ($\ll_P$, [11]). *An objective literal L* depends *on a set of default literals $W \subseteq Def$* with respect to a program P *($L \ll_P W$) iff there is a sequence of rules $\langle r_1, \dots, r_k \rangle$, $k \geq 1$, $r_i \in P$ such that*

- *$head(r_k) = L$,*
- *$W \models body(r_1)$,*
- *for each i, $1 \leq i < k$, $W \cup \{head(r_1), \dots, head(r_i)\} \models body(r_{i+1})$.*

The set $\{L \in Lit \mid L \ll_P W\} \cup W$ is denoted by $Cn_{\ll_P}(W)$.[3]

[2] P^+ is treated as definite logic program, i.e., each objective literal of the form $\neg A$, where $A \in At$, is considered as a new atom.

[3] $Cn_{\ll_P}(W)$ could be defined as $T_P^\omega(W)$ and $L \ll_P W$ as $L \in T_P^\omega(W)$.

$W \subseteq Def$ is self-consistent *w.r.t. a program P iff $Cn_{\ll_P}(W)$ is consistent.* $\square$

If $Z \subseteq Obj$, we will sometimes use the notation $Cn_{\ll_{P \cup Z}}(W)$, assuming that the program P is extended by the set of facts Z.

Definition 2 (Dependency structure). *Let P be a program.*

A self-consistent set $X \subseteq Def$ is called an argument *w.r.t. P for a consistent set of objective literals Y, given a set of objective literals Z iff*

1. *$pos(X) \cap Z = \emptyset$,*
2. *$Y \subseteq Cn_{\ll_{P \cup Z}}(X)$.*

We will use the notation $\langle Y \hookleftarrow X; Z \rangle$ and the triple denoted by it is called a dependency structure *(w.r.t. P).*[4] $\square$

If $Z = \emptyset$ also a shortened notation $\langle Y \hookleftarrow X \rangle$ can be used. We will sometimes omit the phrase "w.r.t. P" and speak simply about dependency structures and arguments, if the corresponding program is clear from the context.

We are going to define basic argumentation structures, while using the same notation as for dependency structures. It is justified by Proposition 1., saying that basic argumentation structures comply with Definition 2 of dependency structures, if some conditions are satisfied.

Definition 3 (Basic argumentation structure). *Let $r \in P$ be a rule such that*

- *$(body(r))^-$ is self-consistent and*
- *$pos((body(r))^-) \cap (body(r))^+ = \emptyset$.*

Then $\mathcal{A} = \langle \{head(r)\} \hookleftarrow (body(r))^-; (body(r))^+ \rangle$ is called a basic *argumentation structure.* $\square$

Proposition 1. *Each basic argumentation structure is a dependency structure.*

Proof. Let $\mathcal{A} = \langle \{head(r)\} \hookleftarrow (body(r))^-; (body(r))^+ \rangle$ be a basic argumentation structure for a rule $r \in P$. We show that$\{head(r)\} \subseteq Cn_{\ll_{P \cup (body(r))^+}}((body(r))^-)$.

Assume the program $P \cup (body(r))^+$. Let $(body(r))^+ = \{L_1, \ldots, L_k\}$. We introduce a new rule $r_{L_i} = L_i \leftarrow$ for every $L_i \in (body(r))^+$. Then we create a sequence of rules $\langle r_1, r_2, \ldots, r_n \rangle$ such that

- $n = |(body(r))^+| + 1$,
- $r_n = r$,
- $r_i = r_{L_i}$ where $L_i \in (body(r))^+$, for $1 \leq i < n$,
- $r_i \neq r_j$ for $1 \leq i, j < n$ and $i \neq j$.

[4] This notation does not refer to P explicitly, but the condition $Y \subseteq Cn_{\ll_{P \cup Z}}(X)$ relates a dependency structure to P. Moreover, we will use only a kind of dependency structures, called argumentation structures, derived from a given program P.

This sequence satisfies conditions from Definition 1 for assumption $(body(r))^-$, hence $head(r) \in Cn_{\ll_{P \cup (body(r))^+}}((body(r))^-)$. That is, we have that $\mathcal{A}$ is a dependency structure. □

We emphasize that only *self-consistent* arguments for *consistent* sets of objective literals are considered in this paper. Hence, programs as $P = \{p \leftarrow not\ p\}$ or $Q = \{p \leftarrow not\ q; \neg p \leftarrow not\ q\}$ are irrelevant for our constructions.

Some dependency structures can be derived from the basic argumentation structures. Only the dependency structures derived from the basic argumentation structures using derivation rules from Definition 4 are of interest in the rest of this paper. We will use the term *argumentation structure* for dependency structures derived from basic argumentation structures using derivation rules.

Derivation rules are motivated later in Example 1.

Definition 4 (Derivation rules and argumentation structures). *Let P be a program. An argumentation structure is inductively defined as follows. Each basic argumentation structure is an argumentation structure.*

Other argumentation structures are obtained using derivation rules R1, R2, and R3:

R1 (Unfolding) *Let* $r_1, r_2 \in P$, $\mathcal{A}_1 = \langle \{head(r_1)\} \hookleftarrow X_1; Z_1 \rangle$ *and* $\mathcal{A}_2 = \langle \{head(r_2)\} \hookleftarrow (body(r_2))^-; (body(r_2))^+ \rangle$ *be argumentation structures,* $head(r_2) \in Z_1$, $X_1 \cup (body(r_2))^- \cup Z_1 \cup (body(r_2))^+ \cup \{head(r_1)\}$ *be consistent and* $X_1 \cup (body(r_2))^-$ *be self-consistent. Then also* $\mathcal{A}_3 = \langle head(r_1) \hookleftarrow X_1 \cup (body(r_2))^-; (Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+ \rangle$ *is an argumentation structure. We also write* $\mathcal{A}_3 = u(\mathcal{A}_1, \mathcal{A}_2)$.

R2 *Let* $\mathcal{A}_1 = \langle Y_1 \hookleftarrow X_1 \rangle$ *and* $\mathcal{A}_2 = \langle Y_2 \hookleftarrow X_2 \rangle$ *be argumentation structures and* $X_1 \cup X_2$ *be self-consistent. Then* $\mathcal{A}_3 = \langle Y_1 \cup Y_2 \hookleftarrow X_1 \cup X_2 \rangle$ *is an argumentation structure. We also write* $\mathcal{A}_3 = \mathcal{A}_1 \cup \mathcal{A}_2$.

R3 *Let* $\mathcal{A}_1 = \langle Y_1 \hookleftarrow X_1 \rangle$ *be an argumentation structure and* $W \subseteq Def$. *If* $X_1 \cup W$ *is self-consistent, then* $\mathcal{A}_2 = \langle Y_1 \hookleftarrow X_1 \cup W \rangle$ *is an argumentation structure. We also write* $\mathcal{A}_2 = \mathcal{A}_1 \cup W$.□

Example 1 ([1]). Let the following program P be given as follows (P is used as a running example in this paper):

$$\begin{array}{ll} r_1 & b \leftarrow a, not\ \neg b \\ r_2 & \neg b \leftarrow not\ b \\ r_3 & a \leftarrow not\ \neg a. \end{array}$$

Suppose that $\prec = \{(r_2, r_1)\}$.

Consider the rule r_2. The default negation *not b* plays the role of a *defeasible argument*. If the argument can be consistently evaluated as true with respect to a program containing r_2, then also $\neg b$ can (and must) be evaluated as true.

As regards the rule r_1, default negation *not* $\neg b$ can be treated as an argument for b, if a is true, it is an example of a "conditional argument".

The following basic argumentation structures correspond to the rules of P: $\langle\{b\} \hookleftarrow \{not\ \neg b\}; \{a\}\rangle$,$\langle\{\neg b\} \hookleftarrow \{not\ b\}\rangle$, $\langle\{a\} \hookleftarrow \{not\ \neg a\}\rangle$. Let us denote them by $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$, respectively.

An example of a derived argumentation structure: $\mathcal{A}_3$ enables to "unfold" the condition a in $\mathcal{A}_1$, the resulting argumentation structure can be expressed as $\mathcal{A}_4 = u(\mathcal{A}_1, \mathcal{A}_3) = \langle\{b\} \hookleftarrow \{not\ \neg b, not\ \neg a\}\rangle$.

Similarly, $\mathcal{A}_5 = \mathcal{A}_3 \cup \mathcal{A}_4 = \langle\{a, b\} \hookleftarrow \{not\ \neg b, not\ \neg a\}\rangle$ can be derived from $\mathcal{A}_3$ and $\mathcal{A}_4$ using the rule R2.

Observe that some argumentation structures correspond to the answer sets. $\mathcal{A}_5$ corresponds to the answer set $\{a, b\}$ and $\mathcal{A}_6 = \langle\{a, \neg b\} \hookleftarrow \{not\ b, not\ \neg a\}\rangle$ to $\{a, \neg b\}$. Notice that $\mathcal{A}_6 = \mathcal{A}_2 \cup \mathcal{A}_3$. The attack relation enables to select the preferred answer set. This will be discussed later. □

Proposition 2. *Each argumentation structure is a dependency structure.*

Proof. We have to show that an application of R1, R2 and R3 preserves the properties of dependency structures.

R1 Since $S_1 = X_1 \cup (body(r_2))^- \cup Z_1 \cup (body(r_2))^+ \cup \{head(r_1)\}$ is consistent then $S_2 = X_1 \cup (body(r_2))^- \cup (Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+ \subseteq S_1$ is also consistent. This means that $pos(X_1 \cup (body(r_2))^-) \cap ((Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+) = \emptyset$.
Let $Q = P \cup (Z_1 \setminus \{head(r_2)\}) \cup (body(r_2))^+$ and $w = head(r_2) \leftarrow$.
From $head(r_2) \in Cn_{\ll_{P \cup (body(r_2))^+}}((body(r_2))^-)$ we have a sequence R_2 of rules, where $R_2 = \langle q_1, q_2, \ldots, q_m\rangle$, $m > 0$ and $q_m = r_2$.
From $head(r_1) \in Cn_{\ll_{P \cup Z_1}}(X_1)$ we have the sequence $R_1 = \langle p_1, p_2, \ldots, p_n\rangle$ where $n > 0$ and $p_n = r_1$. We assume there is at most one occurrence of w in R_1. Otherwise we can remove all but the leftmost one. Note that since $r_2 \in P$ there is a possibility to satisfy $body(r_1)$ in a different way from using w.
If $w \in R_1$ then we have $p_i = w$ for some $1 \leq i < n$. We construct the sequence $R_3 = \langle q_1, q_2, \ldots, q_m, p_1, p_2, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n\rangle$. If $w \notin R_1$ we construct the sequence $R_3 = \langle q_1, q_2, \ldots, q_m, p_1, p_2, \ldots, p_n\rangle$. In both cases R_3 satisfy the conditions from Definition 1 for the assumption $X_1 \cup (body(r_2))^-$.

R2 The condition $pos(X) \cap Z = \emptyset$ is satisfied both for R2 and R3, because $Z = \emptyset$.
It is supposed that $Y_1 \hookleftarrow X_1$ and $Y_2 \hookleftarrow X_2$ are argumentation structures and $X_1 \cup X_2$ is self-consistent. We have to show that $Y_1 \cup Y_2 \subseteq Cn_P(X_1 \cup X_2)$. Let $L \in Y_1 \cup Y_2$. Then $L \in Cn_P(X_1)$ or $L \in Cn_P(X_2)$, hence $L \in Cn_P(X_1 \cup X_2)$.

R3 Now, we assume that $Y \hookleftarrow X$ is an argumentation structure, i.e., $Y \subseteq Cn_P(X)$ and $X \cup W$ is self-consistent. Clearly, $Cn_P(X) \subseteq Cn_P(X \cup W)$, hence $Y \subseteq Cn_P(X \cup W)$.

Definition 5 (Derivations). *A* derivation *of an argumentation structure $\mathcal{A}$ (w.r.t. P) is a minimal sequence $\langle\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\rangle$ of argumentation structures (w.r.t. P) such that $\mathcal{A}_1$ is a basic argumentation structure, $\mathcal{A} = \mathcal{A}_k$, and each*

$\mathcal{A}_i$, $1 < i \leq k$, is either a basic argumentation structure or it is obtained by R1 or R2 or R3 from preceding argumentation structures.

An extraordinary attention is devoted to derivations of complete argumentation structures – they correspond to answer sets.

Definition 6 (Complete argumentation structures). *An argumentation structure $\langle Y \hookleftarrow X \rangle$ is called* complete *iff for each literal $L \in Obj$ it holds that $L \in Y$ or $not\ L \in X$.* $\square$

A set of basic argumentation structures is *assigned* to an arbitrary program P.

Proposition 3. *A complete argumentation structure $\langle Y \hookleftarrow X \rangle$ is derived from a set of basic argumentation structures assigned to a program P iff $X \cup Y$ is an answer set of P.*

A proof is based on the method of [5] and on a correspondence between derivations of argumentation structures and $Cn_{\ll_P}(X)$.

We are interested in attacks against derivations of complete argumentation structures.

4 Attacks and Warranted Derivations

Our approach to preferred answer sets is based on a solution of conflicts between complete argumentation structures. We distinguish three steps towards that goal.

Contradictions between argumentation structures represent the elementary step.

Rule preference and contradictions between basic argumentation structures are used to form an *attack* relation on basic argumentation structures. Consider two basic argumentation structures $\mathcal{A}_1$ and $\mathcal{A}_2$. If $\mathcal{A}_1$ contradicts $\mathcal{A}_2$ and corresponds to a more preferred rule, then it *attacks* $\mathcal{A}_2$.

Attacks between derived argumentation structures depend on how argumentation structures are derived, see Example 3 below. Hence, we will introduce an *attack relation on derivations.* The notion of warranted and blocked complete argumentation structures and of preferred answer set is based on this basis.

Definition 7. *Consider the argumentation structures $\mathcal{A} = \langle Y_1 \hookleftarrow X_1; Z_1 \rangle$ and $\mathcal{B} = \langle Y_2 \hookleftarrow X_2; Z_2 \rangle$.*

If there is a literal $L \in Y_1$ such that $not\ L \in X_2$, it is said that the argument X_1 contradicts *the argument X_2 and the argumentation structure $\mathcal{A}$* contradicts *the argumentation structure $\mathcal{B}$.*

It is said that X_1 is a counterargument *to X_2.* $\square$

The basic argumentation structures corresponding to the facts of the given program are not contradicted.

Let $r_1 = a \leftarrow$ be a fact and $not\ a \in (body(r_2))^-$. Then any $W \subseteq Def$ s.t. $(body(r_2))^- \subseteq W$ is not self-consistent and, therefore, it is not an argument.

Example 2. In Example 1, $\mathcal{A}_1$ contradicts $\mathcal{A}_2$ and $\mathcal{A}_2$ contradicts $\mathcal{A}_1$.

Only some counterarguments are interesting: the rule r_1 is more preferred than the rule r_2, therefore the counterargument of $\mathcal{A}_2$ against $\mathcal{A}_1$ should not be "effectual". We are going to introduce a notion of *attack* in order to denote "effectual" counterarguments. □

Similarly as for the case of argumentation structures, the basic attacks are defined first. A terminological convention: if $\mathcal{A}_1$ attacks $\mathcal{A}_2$, it is said that the pair $(\mathcal{A}_1, \mathcal{A}_2)$ is an attack.

Definition 8. *Let r_1, r_2 be rules, and $\mathcal{A}_1 = \langle \{head(r_1)\} \hookleftarrow (body(r_1))^-; (body(r_1))^+ \rangle$ and $\mathcal{A}_2 = \langle \{head(r_2)\} \hookleftarrow (body(r_2))^-; (body(r_2))^+ \rangle$ be basic argumentation structures such that $r_2 \prec r_1$ and $\mathcal{A}_1$ contradicts $\mathcal{A}_2$.*

Then $\mathcal{A}_1$ attacks *$\mathcal{A}_2$ and it is said that this attack is* basic. □

Next step could be to transfer basic attacks to attacks between derived argumentation structures. However, it is not a straightforward task. Example 3 shows our intuitions. An argumentation structure $\mathcal{B}$ attacks another argumentation structure $\mathcal{A}$ w.r.t. a derivation, but not w.r.t. another derivation.

Example 3. Let P be

r_1	$a \leftarrow not\ b$	r_3	$a \leftarrow not\ c$
r_2	$b \leftarrow not\ a$	r_4	$c \leftarrow b.$

$\prec = \{(r_1, r_2)\}$.

There are two answer sets of P: $S_1 = \{a\}$ and $S_2 = \{b, c\}$. The corresponding argumentation structures are $\mathcal{A} = \langle \{a\} \hookleftarrow \{not\ b, not\ c\} \rangle$ and $\mathcal{B} = \langle \{b, c\} \hookleftarrow \{not\ a\} \rangle$, respectively.

Let $\mathcal{A}_1$ be $\langle \{a\} \hookleftarrow \{not\ b\} \rangle$, $\mathcal{A}_2 = \langle \{b\} \hookleftarrow \{not\ a\} \rangle$, $\mathcal{A}_3 = \langle \{a\} \hookleftarrow \{not\ c\} \rangle$, $\mathcal{A}_4 = \langle \{c\} \hookleftarrow \emptyset; \{b\} \rangle$.

There are two derivations of $\mathcal{A}$: the sequences $\sigma_1 = \langle \mathcal{A}_1, \mathcal{A} \rangle$ and $\sigma_2 = \langle \mathcal{A}_3, \mathcal{A} \rangle$ (remind the minimality condition). They start from a basic argumentation structure and R3 is used.

On the other hand, there is only one[5] derivation of $\mathcal{B}$: $\tau = \langle \mathcal{A}_2, \mathcal{A}_4, \mathcal{B} \rangle$.

The only basic attack is $(\mathcal{A}_2, \mathcal{A}_1)$ ($\mathcal{A}_2$ attacks $\mathcal{A}_1$). Hence, it is intuitive to accept that τ attacks σ_1. However, there is no reason to consider σ_2 as attacked.

A rather credulous approach is accepted in this paper: if there is a derivation of a complete argumentation structure, which is not attacked, then the complete argumentation structure is preferred. However, this is only a rough idea, a more subtle solution is presented below. □

We are going to define attacks between derivations. It is a simple task, but not sufficient for our goals.

[5] If we abstract from the order of argumentation structures in the derivation. This does not influence the attack relation between derivations.

Definition 9. *Let σ be a derivation of an argumentation structure $\mathcal{A}$ and τ a derivation of an argumentation structure $\mathcal{B}$. Suppose that a basic argumentation structure $\mathcal{A}_1$ belongs to σ and a basic argumentation structure $\mathcal{B}_1$ belongs to τ.*

It is said that σ attacks *τ, if $(\mathcal{A}_1, \mathcal{B}_1)$ is a basic attack.*

It is obvious that a derivation σ may attack a derivation τ and τ may attack σ, i.e. mutual attacks are possible. Similarly cyclic attacks are possible.

We intend to define preferred answer sets in terms of preferred complete argumentation structures. A first approximation is to select complete argumentation structures with non-attacked derivations. However, we need to handle the case of mutual or cyclic attacks (i.e., to consider a kind of reinstatement). To this end we borrow a technique from abstract argumentation frameworks [6].

Definition 10. *Consider an argumentation framework (A, α), where A, the set of arguments, is the set of all derivations of all complete argumentation structures and α is the attack relation defined in Definition 9.*

A derivation σ of a complete argumentation structure $\mathcal{A}$ is acceptable *w.r.t. a set $S \subseteq A$ iff for each $\tau \in A$ s.t. $(\tau, \sigma) \in \alpha$ there is some $\sigma' \in S$ s.t. $(\sigma', \tau) \in \alpha$.* $\square$

Notice that acceptable derivations may be attacked by derivations of non-complete argumentation structures.

Fact

If there is a derivation σ of a complete $\mathcal{A}$, which is not attacked, then σ is acceptable w.r.t. the empty set of derivations.

Example 4. Let P be $\{r_1 : \ a \leftarrow not\ b;\ r_2 : \ b \leftarrow not\ a\}$. If $\prec = \emptyset$ then both the derivation of $\langle \{a\} \hookleftarrow \{not\ b\}\rangle$ and the derivation of $\langle \{b\} \hookleftarrow \{not\ a\}\rangle$ are acceptable w.r.t. the empty set of derivations.

Suppose that $r_1 \prec r_2$. Then $\langle \{b\} \hookleftarrow \{not\ a\}\rangle$ is acceptable w.r.t. the empty set of derivations, but there is no set S of derivations s.t. $\langle \{a\} \hookleftarrow \{not\ b\}\rangle$ is acceptable w.r.t. S.

Let R be $P \cup \{r_3 : c \leftarrow a;\ r_4 : \ d \leftarrow b\}$, $r_1 \prec r_2, r_4 \prec r_3$. Then each derivation σ of $\langle \{a, c\} \hookleftarrow \{not\ b, not\ d\}\rangle$ is acceptable w.r.t. $S = \{\sigma\}$ and each derivation τ of $\langle \{b, d\} \hookleftarrow \{not\ a, not\ c\}\rangle$ is acceptable w.r.t. $S = \{\tau\}$.

Example 5. Consider a program P:

r_1	$a_1 \leftarrow not\ a_3, not\ d_2$
r_2	$d_1 \leftarrow not\ a_3, not\ d_2$
r_3	$a_2 \leftarrow not\ a_1, not\ d_3$
r_4	$d_2 \leftarrow not\ a_1, not\ d_3$
r_5	$a_3 \leftarrow not\ a_2, not\ d_1$
r_6	$d_3 \leftarrow not\ a_2, not\ d_1$

$\prec = \{(r1, r4), (r3, r5), (r6, r2)\}$.

We have three complete argumentation structures:
$\mathcal{A}_1 = \langle \{a_1, d_1\} \hookleftarrow \{not\ a_3, not\ d_2\}\rangle$, $\mathcal{A}_2 = \langle \{a_2, d_2\} \hookleftarrow \{not\ a_1, not\ d_3\}\rangle$,
$\mathcal{A}_3 = \langle \{a_3, d_3\} \hookleftarrow \{not\ a_2, not\ d_1\}\rangle$.

We have a cycle of attacks. Each derivation of $\mathcal{A}_2$ attacks each derivation of $\mathcal{A}_1$, each derivation of $\mathcal{A}_3$ attacks each derivation of $\mathcal{A}_2$, each derivation of $\mathcal{A}_1$ attacks each derivation of $\mathcal{A}_3$.

Let $\sigma_1, \sigma_2, \sigma_3$ be derivations of $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$, respectively. It holds that σ_1 is acceptable w.r.t. $\{\sigma_3\}$, σ_2 is acceptable w.r.t. $\{\sigma_1\}$, and σ_3 is acceptable w.r.t. $\{\sigma_2\}$. □

Definition 11 (Warranted and blocked argumentation structures). *Let $\mathcal{A}$ be a complete argumentation structure. If there is an acceptable derivation of $\mathcal{A}$ w.r.t. a set S of some derivations of some complete argumentation structures, then $\mathcal{A}$ is called* warranted*, otherwise it is called* blocked*.* □

5 Preferred Answer Sets

Definition 12 (Preferred answer set). *A complete argumentation structure is* preferred *iff it is warranted.*

$Y \cup X$ is a preferred answer set *iff $\langle Y \hookleftarrow X\rangle$ is a preferred argumentation structure.* □

Notice that our notion of preferred answer set is rather a credulous one.

Example 6. Consider our running Example 1, where we have complete argumentation structures $\mathcal{A}_5 = \langle\{b, a\} \hookleftarrow \{not\ \neg b, not\ \neg a\}\rangle, \mathcal{A}_6 = \langle\{\neg b, a\} \hookleftarrow \{not\ \neg a, not\ b\}\rangle$ and basic argumentation structures $\mathcal{A}_1 = \langle\{b\} \hookleftarrow \{not\ \neg b\}; \{a\}\rangle$, $\mathcal{A}_2 = \langle\{\neg b\} \hookleftarrow \{not\ b\}\rangle$, $\mathcal{A}_3 = \langle\{a\} \hookleftarrow \{not\ \neg a\}\rangle$.

The only basic attack is $(\mathcal{A}_1, \mathcal{A}_2)$, $\mathcal{A}_1$ attacks $\mathcal{A}_2$. Therefore, the derivation $\sigma = \langle\mathcal{A}_1, \mathcal{A}_3, \mathcal{A}_4 = \langle\{b\} \hookleftarrow \{not\ \neg b, not\ \neg a\}\rangle, \mathcal{A}_5\rangle$ attacks $\tau = \langle\mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_6\rangle$.

There is no derivation of $\mathcal{A}_6$ which is not attacked by σ and no derivation of $\mathcal{A}_6$ counterattacks the derivation σ. $\mathcal{A}_6$ is blocked, on the other hand, $\mathcal{A}_5$ is warranted. Hence, we prefer $\mathcal{A}_5$ over $\mathcal{A}_6$.

Consequently, $\{a, b\}$ is a preferred answer set of the given prioritized logic program. □

The following example shows that the argumentation structure corresponding to the only answer set of a program is preferred, even if each its derivation is attacked by a derivation of an argumentation structure which is not complete. The example demonstrates also that attacks between derivations can not be implemented via conventional attacks on arguments. Anyway, a goal of our future research is to find a method how to minimize comparisons of derivations.

Example 7. Consider the program

r_1	$b \leftarrow not\ a$	r_3	$c \leftarrow a$
r_2	$a \leftarrow not\ b$	r_4	$c \leftarrow not\ c$

$\prec = \{(r_2, r_1)\}$.

Let the basic argumentation structures be denoted by $\mathcal{A}_i$, $i = 1, \ldots, 4$. $(\mathcal{A}_1, \mathcal{A}_2)$ is the only basic attack.

The derivation $\langle \mathcal{A}_1 \rangle$ attacks the derivation $\sigma = \langle \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_5, \mathcal{A}_6 \rangle$, where $\mathcal{A}_5 = \langle \{c\} \hookleftarrow \{not\ b\} \rangle$ and $\mathcal{A}_6 = \langle \{c, a\} \hookleftarrow \{not\ b\} \rangle$.

However, $\mathcal{A}_1$ is not a member of a derivation of a complete argumentation structure. Hence, σ is acceptable w..r.t. the empty set according to Definition 10. Therefore, the complete argumentation structure $\mathcal{A}_6$ is warranted and, consequently, it is the preferred argumentation structure. $\Box$

We distinguish between attacking and blocking. If an argumentation structure is blocked then there is no its derivation which counterattacks the attacks of derivations of other *complete* argumentation structures.

Theorem 1. *If S is a preferred answer set of $(P, \prec)$, then S is an answer set of P.*

Proof. If S is a preferred answer set then there is a preferred complete argumentation structure $\mathcal{A} = \langle S^+ \hookleftarrow S^- \rangle$. Hence, S is total. We have to show that $S^+ = Cn_{\ll_P}(S^-) \cap Obj$ using a result of [5].

Clearly, $S^+ \subseteq Cn_{\ll_P}(S^-)$ according to the definition of dependency structure. Let be $L \in Obj$ and $L \in Cn_{\ll_P}(S^-)$. It holds that S^- is self-consistent and S is total. Hence, $not\ L \notin S^-$ and $L \in S^+$. $\Box$

Our next goal is to evaluate the presented approach to preferred answer sets selection. To this end some principles and their (un)satisfaction are discussed in the next section.

6 Evaluation

We start with a discussion of principles proposed by [1]. A new principle requiring selection of a preferred answer set from the non-empty set of standard answer sets is added. After that it is proved that the new principle is satisfied by our approach. Finally, an informal and tentative proposal of some new principles, characterizing the descriptive approach to selection of preferred answer sets is presented.

6.1 Principles

The principles (partially) specify what it means that an order on answer sets corresponds to the given order on rules. Let us start with principles proposed in [1] for arbitrary prioritized theories.

Principle I.
Let B_1 and B_2 be two belief sets of a prioritized propositional theory $(T; \prec)$ generated by the rules $R \cup \{d_1\}$ and $R \cup \{d_2\}$, where $d_1, d_2 \notin R$, respectively. If d_1 is preferred over d_2, then B_2 is not a (maximally) preferred belief set of T. $\Box$

Principle II.
Let B be a preferred belief set of a prioritized propositional theory $(T; \prec)$ and r a rule such that at least one prerequisite of r is not in B. Then B is a preferred belief set of $(T \cup \{r\}; \prec')$ whenever $\prec'$ agrees with $\prec$ on priorities among rules in T. $\square$

We believe that the possibility to select always a preferred answer set from a non-empty set of standard answer sets is of critical importance. Principle III, accepted in this paper, reproduces the idea of Proposition 6.1 from [1].

Principle III.
Let $\mathcal{B} \neq \emptyset$ be the set of all belief sets of a prioritized theory $(T, \prec)$. Then there is a selection function Σ s.t. $\Sigma(\mathcal{B})$ is the set of all preferred belief sets of $(T, \prec)$, where $\emptyset \neq \Sigma(\mathcal{B}) \subseteq \mathcal{B}$. $\square$

We consider and discuss below only a specific case of prioritized theories, prioritized logic programs. Principle I specifies an attack of a belief set B_1 against a belief set B_2. The attack is based on the preference of the rule d_1 over the rule d_2 (they cannot be applied together for generating a preferred answer set). But Principle I is not appropriate for an approach which considers mutual attacks of preferred answer sets and its main goal is to select at least one preferred answer set – existence of an attack against a candidate for a preferred answer set is not sufficient for its elimination. The attacked answer set can be defended by a counterattack of another answer set. In order to summarize, Principle I is not appropriate for an approach which distinguishes between attacking and blocking.

It was shown in [1], Proposition 6.1, that Principle II is incompatible with the existence of a function which selects a non-empty set of preferred answer sets from a non-empty set of standard answer sets of a given logic program, if the notion of preferred answer set from [1] is accepted. Moreover, we have a basic problem with this principle. First an example.

Example 8 ([1]). Suppose that we accept both Principle II and Principle III.

Consider program P, whose single standard answer set is $S = \{b\}$ and the rule (1) is preferred over the rule (2).

$$c \leftarrow \mathit{not}\ b \tag{1}$$

$$b \leftarrow \mathit{not}\ a \tag{2}$$

S is not a preferred answer set in the framework of [1].

Assume that S, the only standard answer set of P, is selected – according to Principle III – as the preferred answer set of $(P, \prec)$.[6] Let P' be $P \cup \{a \leftarrow c\}$ and $a \leftarrow c$ be preferred over the both rules (1) and (2). P' has two standard answer sets, S and $T = \{a, c\}$.

Note that $\{c\} \not\subseteq S^+$. Hence, S should be the preferred answer set of P' according to the Principle II . However, in the framework of [1] the only preferred answer set of $(P', \prec')$ is T. This selection of preferred answer set satisfies clear

[6] Observe that the only derived complete argumentation structure is $\langle \{b\} \hookleftarrow \{\mathit{not}\ a, \mathit{not}\ c\} \rangle$. Hence, $\{b\}$ is a preferred answer set in our framework.

intuitions – T is generated by the two most preferred rules. A consequence, accepted in]1] is that Principle III is refused.

In our approach the complete argumentation structure $\langle \{a, c\} \hookleftarrow \{not\ b\} \rangle$ is preferred and $\{a, c\}$ is the preferred answer set of P'.

Principle III is of crucial value according to our view, therefore we do not accept Principle II. This example is not the only reason for it. A more fundamental reason is expressed in Sect. 6.2 as a principle called *Nonmonotonicity of selection constraints*. We selected in this example preferred answer sets of P' from a broader variety of possibilities. Consequently, a selection of a preferred answer set from the extended set of possibilities should not be limited to a subset of those possibilities.

A more detailed justification of our decision not to accept Principle II is presented in [12]. □

Principle II is not accepted also in [9]. According to [4] descriptive approaches do not satisfy this principle in general.

In the rest of this subsection satisfaction of the Principle III (more precisely, its specialization for prioritized logic programs) is proved.

Lemma 1. *The attack relation between derivations of complete argumentation structures is irreflexive.* □

Proof. Let $\sigma = \langle \mathcal{A}_1, \ldots, \mathcal{A}_k \rangle$ be a derivation of a complete argumentation structure $\mathcal{A}_k$. Suppose to the contrary that σ attacks itself, i.e., there are basic argumentation structures $\mathcal{A}_i, \mathcal{A}_j$ s.t. $\mathcal{A}_i = \langle \{head(r)\} \hookleftarrow \{body^-(r)\}; \{body^+(r)\} \rangle$ attacks $\mathcal{A}_j = \langle \{head(q)\} \hookleftarrow \{body^-(q)\}; \{body^+(q)\} \rangle$, where r, q are rules. It follows that $not\ head(r) \in body^- q$. Contradiction: $\mathcal{A}_k$ is consistent and σ is a minimal sequence with the last member $\mathcal{A}_k$. □

Theorem 2. *Principle III is satisfied.*

Let $\mathcal{P} = (P, \prec)$ be a prioritized logic program and $AS(P) \neq \emptyset$. Then there is a preferred answer set of $\mathcal{P}$ in our approach.

Proof. **Case 1** Assume that P has only one answer set S. if there is only one derivation of $\mathcal{A} = \langle S^+ \hookleftarrow S^- \rangle$, then no complete argumentation structure blocks it (from Lemma 1.). If there are more derivations of $\langle S^+ \hookleftarrow S^- \rangle$, then the argument from the proof of the lemma is applied: $\mathcal{A}$ is consistent and all derivations are minimal sequences with the last member $\mathcal{A}$.

Case 2 Suppose that P has only two answer sets S_1 and S_2. Let the corresponding complete argumentation structures be $\mathcal{A}_1 = \langle S_1^+ \hookleftarrow S_1^- \rangle$ and $\mathcal{A}_2 = \langle S_2^+ \hookleftarrow S_2^- \rangle$, respectively.

Without loss of generality assume that there is a derivation of $\mathcal{A}_1$ which is not attacked by a derivation of $\mathcal{A}_2$. Then P has at least one preferred answer set.

Suppose now that each derivation of $\mathcal{A}_1$ is attacked by a derivation of $\mathcal{A}_2$ and vice versa. Consider a derivation σ of $\mathcal{A}_1$. Let $\{\tau_1, \ldots, \tau_k\}$ be the set of all derivations of $\mathcal{A}_2$ attacking σ. Recall that each τ_i is attacked by a derivation of

$\mathcal{A}_1$. Let S be the set of all derivations of $\mathcal{A}_1$ attacking at least one τ_i. It holds that σ is acceptable w.r.t. S, hence $\mathcal{A}_1$ is warranted.

Case 3 Let be $AS(P) = \{S_1, \ldots, S_k\}, k \geq 3$. Assume that the corresponding complete argumentation structures are $\mathcal{A}_i, i = 1, \ldots, k$.

If there is a derivation of some $\mathcal{A}_i$, which is not attacked, then the corresponding answer set is preferred.

Otherwise, all derivations of all complete argumentation structures are attacked by a derivation of a complete argumentation structure. By a generalization of the argument of Case 2 we have that each derivation of each complete argumentation structure is defended by a set of derivations. $\Box$

6.2 Discussion – Descriptive Approach

Finally, a discussion of a tentative proposal of some possible principles appropriate for a descriptive approach to preferred answer sets selection is presented. The principles are expressed in a more or less informal way and represent a very preliminary attempt. All the principles are inspired by our definitions and constructions, but they are not intended solely for the framework presented in this paper. A general and more detailed discussion of the postulates is postponed for a future paper.

The following principle represents a more careful, but less deep version of Principle I:

Principle – Defeated answer sets.
Let S_1, S_2 be answer sets of a program P, and let S_2 be not defeated. If S_1 is defeated by S_2, then S_1 cannot be a preferred answer set.[7] $\Box$

The principle above and the following one can be considered as expressing a little bit more accurately the main intuitive idea of our stance w.r.t. descriptive approach to preferred answer sets selection: it is desirable to apply all rules of an undefeated set of generating rules. There is a difference between attacking and blocking (defeating).

Below is the other side of this intuitive idea: rules can be blocked by more preferred rules but other rules are handled in a declarative style.

Principle of blocking.
If a standard answer set A is generated by a set R of rules, where no rule is attacked by a more preferred rule then A is a preferred answer set. $\Box$

Next principle is inspired by the problem of Example 8. The problem was as follows: a program P with a set $\mathcal{S}$ of answer sets was given together with a program R s.t. $P \subset R$. $\mathcal{M} \neq \mathcal{S}$ is the set of all answer sets of R. According to our view conditions expressed for a selection of preferred members of $\mathcal{S}$ may not constrain a selection of preferred members of the different $\mathcal{M}$. If we select

[7] Of course, there are different possible ways how to specify the notion of defeat. A definition of defeated generating sets of rules can be obtained in a straightforward way from the notion of defeat presented in this paper.

a preferred answer set from $\mathcal{M}$ then we can not limit (constrain) the selection to $\mathcal{S}$.

Principle – Nonmonotonicity of selection constraints.
If $P \subset R$ are programs, then a selection of preferred answer sets of R should not be limited to the set of preferred answer sets of P. $\square$

Attacks of rules, which do not contribute to a generation of a standard answer set, are irrelevant w.r.t. a selection of preferred answer sets:

Principle – Irrelevant attacks.
Let $r_1, r_2 \in P$, r_1 attacks r_2, but r_1 is not a member of a set of generating rules of a standard answer set and $r_2 \in R$, where R is a set of rules generating a standard answer set A.

If there is no other attack against rules of R, then A is a preferred answer set. $\square$

As regards a choice of principles, we accept the position of [1]: even if somebody does not accept a set of principles for preferential reasoning, those (and similar) principles are still of interest as they may be used for classifying different patterns of reasoning.

Of course, some of principles proposed in this subsection may be refused, or some new may be suggested. Different sets of such principles provide different conceptions of a descriptive approach to preferred answer sets selection.

Finally, notice that our descriptive approach can be expressed without argumentation structures using a translation to generating sets of rules.

7 Related Work

D-preference [2], W-preference [18], and B-preference [1]. D/W/B-preferences are representatives of prescriptive approaches. They are based on the view that preference specifies the order in which rules have to be applied. A preferred rule is forced to be applied first. If a more preferred rule has in its body a literal, which is the head of a less preferred rule, then the more preferred rule is not applicable. As a consequence, there are programs with standard answer sets, but without preferred answer sets (hence, Principle III is not satisfied in those approaches).

Our approach enables to select at least one preferred answer set from the non-empty set of standard answer sets of a program. Not all preferences are effective, i.e. not all preferences are transformed to attacks between derivations of argumentation structures.

Therefore D/W/B-preferences are not in direct hierarchic (subset) relation to our semantics.

A fundamental difference between our approach and D/W/B-preference is that testing D/W/B-preference is local. When testing whether an answer set X is preferred, it is not needed to know other answer sets. The computational complexity of those approaches remains within NP. On the other hand, in our approach, all the attacks between derivations of all complete argumentation structures (they correspond to answer sets) are considered. Hence, our conjecture is that the decision problem, whether a complete argumentation structure (an answer set) is a preferred one, is in our approach beyond the class NP.

Sakama and Inoue [9]. Sakama and Inoue have defined an approach that selects preferred answer sets given the preference on literals. A preference relation on literals is transferred to a preference relation on sets of literals. Preferred answer sets are then the maximal (with respect to a preference relation) answer sets. They also provide a way to transform preference on rules to preference on literals. However, structure of the rules, i.e. which rule is blocked by which rule, is not considered during the transformation.

Wakaki [17]. Wakaki has extended Dung's abstract argumentation framework in order to work with preferences. She has introduced preference relation on arguments. Selection of a preferred extension (in a sense of preference on arguments) is done in a similar manner that Sakama and Inoue use to select preferred answer sets. Wakaki then defines a non-abstract logic programming based argumentation framework. Rules of a logic program are transformed to arguments. Preference on literals is transferred to preference on arguments via heads of rules. Wakaki's and our argumentation framework are principally different. Wakaki's goal is to extend Dung's abstract argumentation framework. When selecting a preferred extension in an abstract framework, there is no information about the structure of arguments. On the other hand, approaches for preference handling that work with preference on rules depend on the structure of the rules. The non-abstract argumentation framework proposed by Wakaki deals with preference on literals, which we do not address by our framework. Just to note, Wakaki's non-abstract framework is equivalent with Sakama and Inoue's approach to preference on literals.

Gabaldon [7]. Gabaldon works with extended logic programs and preference (called priority) on rules. His goal it to develop a semantics that always selects (i) a preferred answer set when a standard one exists, and (ii) the only preferred answer set for fully prioritized programs when a standard one exists. The semantics is defined in three steps. First, a partially prioritized program is fully prioritized. Second, a program is transformed to a prerequisite-free program using the unfolding operator. Third, a test is defined to test whether an answer set is preferred. The rules of a program are applied one at the time. A rule is applicable if all its prerequisites were already derived. The order, in which rules are applied, does not have to correspond to priorities. Priorities are used when there are rules with satisfied prerequisites that block each other via default assumptions. Then the preferred rule is used. An answer set is preferred if it can be generated in the aforementioned way.

The main difference between Gabaldon's and our approach is that Gabaldon's semantics does not satisfy Principle III. It guarantees existence of a preferred answer set when a standard one exists only for a subclass of programs (head-cycle-free and head-consistent). We guarantee it for every logic program. The considered subclass comprises the programs without integrity constraints that are encoded as rules that form a negative odd cycle. Gabaldon motivates this focus by complexity concerns. If integrity constraints are allowed we need to know whether there are other preferred answer sets when testing whether an answer set is a preferred one. An answer set that should be preferred from the view of preference can be ruled out by an integrity constraint. A deeper analysis of the relation between our and Gabaldon's semantics is a subject of our future mutual cooperation.

8 Conclusions

An argumentation framework has been constructed, which enables transferring attacks of rules to attacks between derivations of argumentation structures and, consequently, to warranted complete argumentation structures. Preferred answer sets correspond to warranted complete argumentation structures. This construction enables a selection of a preferred answer set whenever there is a non-empty set of standard answer sets of a program. This feature is paid by an increasing computational complexity. The representative approaches based on the prescriptive approach remain in the class NP, but our approach is beyond that class.

We did not accept the second principle from [1] and we needed to modify their first principle.

Among goals for our future research are a development of the set of principles, characterizing a descriptive approach and a continuation of our approach without the transfer to argumentation structures. A consideration of attacks between generating sets of rules represents a natural solution. Preliminary results of this research are published in [14,15] and also in [16]. A more detailed comparison of our approach(es) to other approaches is needed. Also approaches not referenced in this paper are of interest.

Finally, we have to mention the main differences between the preliminary version [12], the version presented at WLP 2011 [13], and this paper.

Both in [12,13] were used attack derivation rules. They were proposed in order to derive attacks between argumentation structures from the basic attacks. However, we did not find a proper version of the rules. The rules of [12] were too liberal, they did not derive all intuitive attacks and, consequently, the set of preferred answer sets was too broad. Moreover, a dependency of attacks against argumentation structures on derivations of argumentation structures was not explicitly stated. As regards attack derivation in [13], a more subtle set of derivation rules is introduced, a superset of attacks was derivable and attacks of argumentation structures were explicitly relativized to derivations of argumentation structures. However, derivation rules Q2 and Q3 were sensitive to arbitrary attacks and, as a consequence, they did not ignore irrelevant attacks (in the sense of a Principle in Sect. 6.2).

A claim that Principle I holds, was in both papers, the proof in [13] was not correct. Principle I does not hold in the current paper and we consider this as an important feature of our descriptive approach.

Attack derivation rules are omitted in this paper. Attacks between derivations of argumentation structures were defined directly. An important new notion is an acceptable derivation. The notion enables a correct handling of mutual and cyclic attacks and a clear characterization of a difference between attacking and blocking. Omitting of attack derivation rules simplifies our approach, enables a more clear, more transparent exposition of our semantics and a more reliable characterizations of its properties.

Acknowledgements. We are grateful to anonymous referees for careful reading, for very valuable, detailed and helpful comments and proposals. This paper was supported by the grants 1/0689/10 and 1/1333/12 of VEGA.

References

1. Brewka, G., Eiter, T.: Preferred answer sets for extended logic programs. Artif. Intell. **109**(1–2), 297–356 (1999)
2. Delgrande, J., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory Pract. Logic Program. **3**(2), 129–187 (2003)
3. Delgrande, J., Schaub, T.: Expressing preferences in default logic. Artif. Intell. **123**(1–2), 41–87 (2000)
4. Delgrande, J., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Comput. Intell. **20**(2), 308–334 (2004)
5. Dimopoulos, Y., Torres, A.: Graph theoretical structures in logic programs and default theories. Theoret. Comput. Sci. **170**(1–2), 209–244 (1996)
6. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. **77**, 321–357 (1995)
7. Gabaldon, A.: A selective semantics for logic programs with preferences. In: Proceedings of the International Workshop on Nonmonotonic Reasoning, Action and Change (NRAC 11), Barcelona, Spain (2011)
8. García, A.J., Simari, G.R.: Defeasible logic programming: an argumentative approach. Theory Pract. Logic Program. **4**(1–2), 95–138 (2004)
9. Sakama, C., Inoue, K.: Prioritized logic programming and its application to commonsense reasoning. Artif. Intell. **3**(1–2), 185–222 (2000)
10. Schaub, T., Wang, K.: A comparative study of logic programs with preference. In: IJCAI 2001, pp. 597–602 (2001)
11. Šefránek, J.: Rethinking semantics of dynamic logic programs. In: Proceedings of the Workshop NMR (2006)
12. Šefránek, J.: Preferred answer sets supported by arguments. In: Proceedings of the Workshop NMR (2008)
13. Šefránek, J., Šimko, A.: Warranted derivation of preferred answer sets. In: Proceedings of WLP (2011)
14. Šimko, A.: Preferred answer sets - banned generating set. In: Proceedings of the Student Conference, Bratislava 2011, Faculty of Mathematics, Physics and Informatics, Comenius University, pp. 326–333 (2011)

15. Šimko, A.: Selection of a preferred answer sets as a decision between generating sets. In: 6th Workshop on Intelligent and Knowledge Oriented Technologies, pp. 51–56 (2011)
16. Šimko, A.: Accepting the natural order of rules in a logic program with preferences. In: Proceedings of the Doctoral Consortium at ICLP 2011 (2011)
17. Wakaki, T.: Preference-based argumentation handling dynamic preferences built on prioritized logic programming. In: Kinny, D., Hsu, J.-I., Governatori, G., Ghose, A.K. (eds.) PRIMA 2011. LNCS, vol. 7047, pp. 336–348. Springer, Heidelberg (2011)
18. Wang, K., Zhou, L.-Z., Lin, F.: Alternating fixpoint theory for logic programs with priority. In: Palamidessi, C., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, p. 164. Springer, Heidelberg (2000)

Solving Modular Model Expansion: Case Studies

Shahab Tasharrofi, Xiongnan (Newman) Wu, and Eugenia Ternovska(✉)

Simon Fraser University, Burnaby, Canada
{sta44, xwa33, ter}@cs.sfu.ca

Abstract. Model expansion task is the task of representing the essence of search problems where we are given an instance of a problem and are searching for a solution satisfying certain properties. Such tasks are common in AI planning, scheduling, logistics, supply chain management, etc., and are inherently modular. Recently, the model expansion framework was extended to deal with multiple modules to represent e.g. the task of constructing a logistics service provider relying on local service providers. In the current paper, we study existing systems that operate in a modular way in order to obtain general principles of solving modular model expansion tasks. We introduce a general algorithm to solve model expansion tasks for modular systems. We demonstrate, through several case studies, that our algorithm closely corresponds to what is done in practice in different areas such as Satisfiability Modulo Theories (SMT), Integer Linear Programming (ILP), and Answer Set Programming (ASP). We make our framework language-independent through a model-theoretic development.

1 Introduction

In [1], the authors formalize search problems as the logical task of *model expansion* (*MX*), the task of expanding a given (mathematical) structure with new relations. They started a research program of finding common underlying principles of various approaches to specifying and solving search problems, finding appropriate mathematical abstractions, and investigating complexity-theoretic and expressiveness issues. The next step in the development of the MX-based framework is adding modularity concepts. The following example clarifies our goals.

Example 1 (*Factory*). Figure 1 shows a modular representation of a factory. It consists of an office and a workshop.[1] The office takes a list of goods needed by consumers and the workshop takes a list of raw materials. These two entities, i.e., the office and the workshop, can communicate with each other in order to plan the production of customers' orders according to both their internal constraints (such as their maximum throughput) and their external constraints (such as the cost of raw materials). Modularity is incorporated through representing each part in the most suitable language. For example, the office is more easily specified in

[1] A more realistic example contains many more modules.

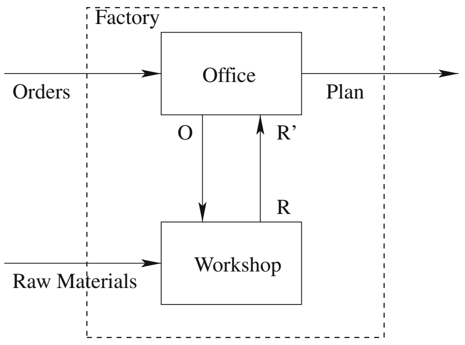


Fig. 1. Modular representation of a factory

extended first-order logic, while the complex operation of the workshop module is perhaps most easily specified using ASP (answer set programming) in order to handle exceptions. In this paper, we take initial steps towards solving the underlying computationally complex task.

In a recent work [2], a subset of the authors extended the MX framework to represent a modular system. Under a model-theoretic view, an MX module can be viewed as a set (or class) of structures satisfying some axioms. An abstract algebra on MX modules was developed, and it allows one to combine modules on abstract model-theoretic level, independently from the languages used for describing them. Perhaps the most important operation in the algebra is the loop (or feedback) operation, since iteration underlies many solving methods. The authors show that the power of the loop operator is such that the combined modular system can capture all of the complexity class NP even when each module is deterministic and polytime. In general, adding loops gives a jump in the polynomial time hierarchy, one step from the highest complexity of the components.

To develop the framework further, we need a method for "solving" modular MX systems, i.e., finding structures which are in the modular system, where the system is viewed as a function of individual modules. *Our goal is to come up with a general algorithm which takes a modular system as its input and generates its solutions.* The main challenge is to come up with an appropriate mathematical abstraction of "combined" solving. Since we aim at developing the foundations of a language-independent problem solving, we tackle the problem model-theoretically.

We take our inspiration in how "combined" solvers are constructed in the general field of declarative problem solving. The field consists of many areas such as Integer Linear Programming (ILP), Answer Set Programming (ASP), Satisfiability Modulo Theories (SMT), Satisfiability (SAT), and Constraint Programming (CP), and each of these areas has developed multitudes of solvers, including "combined" solvers such as SMT solvers. Moreover, SMT-like techniques are needed in the ASP community [3]. These "combined" solvers are very powerful, but are not general enough to find solutions to arbitrary modular systems. On the other hand, we aim at solving systems with arbitrary modules, in particular those representing NP-complete problems. For example, Travelling Salesman and Knapsack modules may both be used by a logistics service provider to construct plans of how to pack shipments and deliver them to their destinations[2]; and the two modules may interact with each other during solving just as SAT and theory modules interact within an SMT solver. Thus, our goal is not to compete with existing systems, and not to replace them, but to develop a new approach for finding solutions to modular systems. In order to active this general goal, we study existing "combined" solvers. Our contributions are as follows.

1. We formalize common principles of "combined" solving in different communities in the context of modular model expansion. The main novelty of our formalization (and thus of our analysis of existing systems) is in clear separation of problem instance and problem specification. Instances are viewed as first-order structures, and expansions of those structures (i.e., solutions) are constructed during solving. [3] Just as in [2], we use a combination of a model-theoretic, algebraic and operational view of modular systems.
2. We design an abstract algorithm that given a modular system, computes the models of that modular system iteratively, and we formulate conditions on languages of individual modules to participate in the iterative solving. Correctness of our algorithm is proven model-theoretically.
3. We introduce abstractions for many ideas in practical systems such as the concept of a *valid acceptance procedure* that abstractly represents unit propagation in SAT, well-founded model computation in ASP, arc-consistency checking in CP, etc.
4. Using the proposed framework, we perform several case studies of existing systems from different communities. For each system with a problem specification, we design a compound modular system which takes a problem instance as input, and outputs the solution, such that the set of structures in the modular system corresponds to the set of solutions for the given problem specification.
   We show that, for the task of model expansion, our algorithm generalizes the work of those systems in a unifying and abstract way. In particular, we show that DPLL(T) framework [4], branch-and-cut based ILP solver [5] and the state-of-the-art combination of ASP and CP [6] are all specializations of our algorithm. In this way, we show the feasibility of our algorithm for solving arbitrary modular systems.
5. We show how solving modular systems can benefit from the techniques used in practical solver constructions. For example, we show how unit propagation techniques in ASP and constraint propagation techniques in CP can be used

[2] We are equally interested in representing multi-module processes that do not require more than a polynomial number of steps to solve, as is common in some business processes such as signing a document.

[3] This view follows the research program started in [1].

to speed up solving of modular systems described in a combination of the two languages.

2 Background

2.1 Model Expansion

In [1], the authors formalize combinatorial search problems as the task of *model expansion* (MX), the logical task of expanding a given (mathematical) structure with new relations. Formally, the user axiomatizes the problem in some logic $\mathcal{L}$. This axiomatization relates an instance of the problem (a *finite structure*, i.e., a universe together with some relations and functions), and its solutions (certain *expansions* of that structure with new relations or functions). Logic $\mathcal{L}$ corresponds to a specification/modelling language. It could be an extension of first-order logic such as FO(ID), or an ASP language, or a modelling language from the CP community such as ESSENCE [7].

Recall that a vocabulary is a set of non-logical (predicate and function) symbols. An interpretation for a vocabulary is provided by a *structure*, which consists of a set, called the domain or universe and denoted by $dom(.)$, together with a collection of relations and (total) functions over the universe. A structure can be viewed as an *assignment* to the elements of the vocabulary. An expansion of a structure $\mathcal{A}$ is a structure $\mathcal{B}$ with the same universe, and which has all the relations and functions of $\mathcal{A}$, plus some additional relations or functions.

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

Model Expansion for logic $\mathcal{L}$
Given: *1.* An $\mathcal{L}$-formula ϕ with vocabulary $\sigma \cup \varepsilon$
2. A structure $\mathcal{A}$ for σ
Find: an expansion of $\mathcal{A}$, to $\sigma \cup \varepsilon$, that satisfies ϕ.

Thus, we expand the structure $\mathcal{A}$ with relations and functions to interpret ε, obtaining a model $\mathcal{B}$ of ϕ. We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary.[4]

Example 2. The following logic program ϕ constitutes an MX specification for Graph 3-colouring:

$$\begin{aligned} 1\{R(x), B(x), G(x)\}1 &\leftarrow V(x). \\ \bot &\leftarrow R(x), R(y), E(x,y). \\ \bot &\leftarrow B(x), B(y), E(x,y). \\ \bot &\leftarrow G(x), G(y), E(x,y). \end{aligned}$$

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

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

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

The interpretations of ε, for structures $\mathcal{B}$ that satisfy ϕ, are exactly the proper 3-colourings of $\mathcal{G}$.

Example 3 (*Factory as Model Expansion*). In Fig. 1, both the office box and the workshop box can be viewed as model expansion tasks. For example, the box labeled with "Workshop" can be abstractly viewed as an MX task with instance vocabulary $\sigma = \{RawMaterials\}$ and expansion vocabulary $\varepsilon = \{R\}$.

Moreover, in Fig. 1, the bigger box with dashed borders can also be viewed as an MX task with instance vocabulary $\sigma' = \{Orders, RawMaterials\}$ and expansion vocabulary $\varepsilon' = \{Plan\}$. This task is a compound MX task whose result depends on the internal work of the office and the workshop.

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

2.2 Modular Systems

This section reviews the concept of a modular system defined in [2] based on the initial development in [8]. As in [2], *each modular system abstractly represents an MX task*, i.e., a set (or class) of structures over some instance and expansion vocabulary. A modular system is formally described as a set of primitive modules (individual MX tasks) combined using the operations of:

1. Projection ($\pi_\tau(M)$) which restricts the vocabulary of a module,
2. Composition ($M_1 \rhd M_2$) which connects outputs of M_1 to inputs of M_2,
3. Union ($M_1 \cup M_2$),
4. Feedback ($M[R = S]$) which connects output S of M to its inputs R and,
5. Intersection ($M_1 \cap M_2$).

Formal definitions of these operations are not essential for understanding this paper, thus, we refer the reader to [2] for details. We illustrate these operations by giving the following algebraic specification for the modular system in Example 1.

$$\text{Factory} := \pi_{\{\text{Goods},\text{RawMaterials},\text{Plan}\}}(\text{Office} \rhd \text{Workshop})[R' = R]). \quad (1)$$

Considering Fig. 1, symbol "Factory" refers to the whole modular system denoted by the box with dotted borders. The only important vocabulary symbols outside this box are "Goods", "RawMaterials" and "Plan". All other symbols are projected out. There is also a feedback from R to R'. In this paper, we only consider modular systems which do not use the union operator.

A description of a modular system (1) looks like a formula in some logic. One can define a satisfaction relation for that logic, however it is not needed here. Still, since each modular system is a set of structures, we call the structures in a modular system *models* of that system. We are looking for models of a modular system M which expand a given instance structure $\mathcal{A}$. We call them *solutions of M for $\mathcal{A}$*.

3 Computing Models of Modular Systems

This section gives an algorithm which takes a modular system M and a structure $\mathcal{A}$ and expands $\mathcal{A}$ to $\mathcal{B} \in M$. The algorithm uses a tool external to the modular system (a solver) and the modules of a modular system (to "assist" the solver in finding a model if one exists). The algorithm gradually extends the current structure by interacting with the modules of the given modular system until either an expansion $\mathcal{B}$ of $\mathcal{A}$ is found in M or it concludes that no such $\mathcal{B}$ exists.

3.1 Partial Structures

Recall that a structure is a domain together with an interpretation of a vocabulary. A partial structure, however, may contain unknown values. Partial structures deal with gradual accumulation of knowledge.

Definition 1 (Partial Structure). *We say $\mathcal{B}$ is a τ_p-partial structure over vocabulary τ if: 1. $\tau_p \subseteq \tau$, 2. $\mathcal{B}$ gives a total interpretation to symbols in $\tau \backslash \tau_p$ and, 3. for each n-ary symbol R in τ_p, $\mathcal{B}$ interprets R using two sets R^+ and R^- such that $R^+ \cap R^- = \emptyset$, and $R^+ \cup R^- \neq (dom(\mathcal{B}))^n$. We say that τ_p is the partial vocabulary of $\mathcal{B}$. If $\tau_p = \emptyset$, then we say $\mathcal{B}$ is total. For two partial structures $\mathcal{B}$ and $\mathcal{B}'$ over the same vocabulary and domain, we say that $\mathcal{B}'$* extends *$\mathcal{B}$ if all unknowns in $\mathcal{B}'$ are also unknowns in $\mathcal{B}$.*

Example 4. Consider a structure $\mathcal{B}$ with domain $\{0, 1, 2\}$ for vocabulary $\{I, R\}$, where I and R are unary relations, and $I^{\mathcal{B}} = \{\langle 0 \rangle, \langle 1 \rangle\}$, $\langle 0 \rangle \in R^{\mathcal{B}}$, and $\langle 1 \rangle \notin R^{\mathcal{B}}$, but it is unknown whether $\langle 2 \rangle \in R^{\mathcal{B}}$ or $\langle 2 \rangle \notin R^{\mathcal{B}}$. Then $\mathcal{B}$ is a $\{R\}$-partial structure over vocabulary $\{I, R\}$ where ${R^+}^{\mathcal{B}} = \{\langle 0 \rangle\}$ and ${R^-}^{\mathcal{B}} = \{\langle 1 \rangle\}$.

A τ_p-partial structure $\mathcal{B}$ (over $\sigma \cup \varepsilon$ may have enough information to satisfy a formula ϕ ($\mathcal{B} \models \phi$) or falsify it ($\mathcal{B} \models \neg\phi$). Note that $\mathcal{B}$ may also neither satisfy nor falsify ϕ. $\mathcal{B}$ is called the *empty expansion* of σ-structure $\mathcal{A}$ if $\mathcal{B}$ and $\mathcal{A}$ agree over σ, ${R^+}^{\mathcal{B}} = {R^-}^{\mathcal{B}} = \emptyset$ for all $R \in \varepsilon$, and $\varepsilon = \tau_p$. Here, a structure is always means a total structure. We also talk about "bad" partial structures which, informally, are the ones without a total extension in M. Furthermore, we always assume that $\tau_p \subseteq \varepsilon$.

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

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

Definition 2 (Good Partial Structures). *For a set of structures S and partial structure $\mathcal{B}$, we say $\mathcal{B}$ is a* good partial structure wrt *S if there is $\mathcal{B}' \in S$ which extends $\mathcal{B}$.*

3.2 Requirements on the Modules

As expressed in the introduction, there is practical urge to solve complex computational tasks in a modular way so that full access to a complete axiomatization of the module is not assumed, i.e., the module is treated as a black box and accessed via controlled methods. However, clearly, as the solver has no information about internals of the modules, modules have to assist it. Therefore, the question is "how can the modules assist the solver in its search for a solution?" Intuitively, modules should be able to tell if the solver is on the "right" path or not, i.e., decide if the current partial structure is bad, and if so, tell the solver not to develop in this direction any further. We accomplish this goal by letting a module accept or reject a partial structure. Moreover, in the case of rejection, modules provide a "reason" which prevents the solver from producing the same model over and over. Furthermore, a module may "know" some extra information, and thus, they may pass hints about the right "path" to the solver. Our algorithm models such hints using "advices" to the solver.

Definition 3 (Advice). *Let Pre and $Post$ be formulas in a language $\mathcal{L}$. Formula $\phi := Pre \supset Post$ is an* advice *wrt a partial structure $\mathcal{B}$ and a set of structures M if: 1. $\mathcal{B} \models Pre$, 2. $\mathcal{B} \not\models Post$ and, 3. for every total structure $\mathcal{B}'$ in M, we have $\mathcal{B}' \models \phi$.*

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

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

Definition 4 (Solver Language). *Language $\mathcal{L}$ is a* solver language *if (1–6) below hold:*

1. *All first-order atomic sentences (i.e., $R(t_1, \cdots, t_n)$ with $t_1, \cdots, t_n$ variable-free terms) are in $\mathcal{L}$. Also, if $\phi_1, \phi_2 \in \mathcal{L}$ then $\neg\phi_1 \in \mathcal{L}$ and $(\phi_1 \supset \phi_2) \in \mathcal{L}$.*
2. *$\mathcal{L}$ has a decidable satisfiability relation, i.e., given partial str. $\mathcal{B}$ and $\phi \in \mathcal{L}$, it can be decided if (1) $\mathcal{B} \models_L \phi$, (2) $\mathcal{B} \models_L \neg\phi$, or (3) $\mathcal{B}$ neither satisfies nor falsifies ϕ.*
3. *$\mathcal{L}$'s satisfiability relation respects partial structures extension, i.e., for partial str. $\mathcal{B}$ and $\mathcal{B}'$, if $\mathcal{B} \sqsubseteq \mathcal{B}'$ and $\mathcal{B} \models_L \phi$ then $\mathcal{B}' \models_L \phi$.*
4. *Semantics of $\mathcal{L}$ for connectives $\neg$ (negation) and $\supset$ (implication) is classical.*
5. *$\mathcal{L}$ is monotone, i.e., for sets of axioms Γ, Γ': $\Gamma \subseteq \Gamma' \Rightarrow Con_L(\Gamma) \subseteq Con_L(\Gamma')$.*
6. *$\mathcal{L}$ has resolution and deduction theorems, i.e., $\Gamma \models_L A \supset B$ iff $\Gamma \cup \{A\} \models_L B$.*

The presence of the resolution theorem in Definition 4 guarantees that, once an advice of form *Pre* $\supset$ *Post* is added to the solver, and when the solver has deduced *Pre* under some assumptions, it can also deduce *Post* under the same assumptions. From now on, we assume that our advices and reasons are expressed in a language as above, i.e., a solver language.

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

Definition 5 (Valid Acceptance Procedure). *Let S be a set of τ-structures. We say that P is a valid acceptance procedure for S if for all τ_p-partial structures $\mathcal{B}$, we have:*

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

The procedure above is called valid as it never rejects any good partial structures. However, it is a weak acceptance procedure because it may accept some bad partial structures. This kind of weak acceptance procedures are abundant in practice, e.g., Unit Propagation in SAT, Arc-Consistency Checks in CP, and computation of Founded and Unfounded Sets in ASP. As these examples show, such weak notions of acceptance can usually be implemented efficiently as they only look for local inconsistencies. Informally, oracles accept/reject a partial structure through a valid acceptance procedure for a set containing all possible instances of a problem and their solutions. We call this set a Certificate Set. Before giving its formal definition, we should however point out one difference to the readers who are not accustomed to the logical approach to complexity: In theoretical computer science, a problem is a subset of $\{0, 1\}^*$. However, in descriptive complexity, the equivalent definition of a problem being a set of structures is adopted. Now, we give the formal definition of the Certificate Set.

Definition 6 (Certificate Set). *Let σ and ε be instance and expansion vocabularies. Let $\mathcal{P}$ be a problem, i.e., a set of σ-structures, and C be a set of $(\sigma \cup \varepsilon)$-structures. Then, C is a $(\sigma \cup \varepsilon)$-*certificate set for *$\mathcal{P}$ if for all σ-structures $\mathcal{A}$: $\mathcal{A} \in \mathcal{P}$ iff there is a structure $\mathcal{B} \in C$ that expands $\mathcal{A}$.*

Example 5 (*Graph 3-colouring: Certificates*). Consider Example 2 of graph 3-colouring. There, $\sigma = \{E\}$ and $\varepsilon = \{R, G, B\}$. The problem P is the set of graphs $\mathcal{G} = (V^{\mathcal{G}}; E^{\mathcal{G}})$ which are 3-colorable. A certificate set C for problem P of graph 3-colouring is, as one might expect, the same as 3-colouring certificates in complexity theory, i.e., a partitioning of vertices into three sets R, G and B such that no edge of the graph connects vertices of the same color together. The certificate set C, as expected, should be so that $\mathcal{A} \in P$ (i.e., $\mathcal{A}$ is 3-colorable) iff C has at least one 3-colouring for $\mathcal{A}$ (i.e., there is at least one expansion $\mathcal{B}$ of $\mathcal{A}$ in C which interprets R, G and B correctly).

Recall that each module is associated with an oracle to accept/reject a partial structure and give reasons and advices accordingly. Oracles are the interfaces between our algorithm and our modules. Next, we present conditions that oracles should satisfy so that their corresponding modules can contribute to our algorithm.

Definition 7 (Oracle Properties). *Let $\mathcal{L}$ be a solver language. Let $\mathcal{P}$ be a problem, and let O be an oracle. We say that O is:*

- Complete and Constructive (CC) wrt $\mathcal{L}$ *if O returns a reason $\psi_{\mathcal{B}}$ in $\mathcal{L}$ for each partial structure $\mathcal{B}$ that it rejects such that: (1) $\mathcal{B} \models \neg\psi_{\mathcal{B}}$ and, (2) all total structures accepted by O satisfy $\psi_{\mathcal{B}}$.*
- Advising (A) wrt $\mathcal{L}$ *if O gives a set of advices in $\mathcal{L}$ wrt $\mathcal{B}$ for all partial str. $\mathcal{B}$.*
- Verifying (V) *if O is a valid acceptance procedure for some certificate set C for P.*

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

Example 6 (*Graph 3-colouring: Reasons and Advices*). Consider the graph 3-colouring example of Example 2. We want to describe some possible scenarios for an oracle O of graph 3-colouring. Consider graph $\mathcal{G} = (V^{\mathcal{G}}; E^{\mathcal{G}})$ with $V^{\mathcal{G}} = \{a, b, c, d\}$ and $E^{\mathcal{G}} = \{(a, b), (b, a), (a, c), (c, a), (a, d), (d, a), (c, d), (d, c)\}$. Also consider a partial expansion $\mathcal{B} = (V^{\mathcal{G}}; E^{\mathcal{G}}, R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})$ of $\mathcal{G}$ to $\{R, B, G\}$ which assigns color red to vertices a and b, color green to vertex c and (yet) no color to vertex d. Obviously, $\mathcal{B}$ is a bad partial 3-colouring and no matter what color we assign to d, we will not obtain a valid 3-colouring. Therefore, one scenario for oracle O is to reject this partial colouring and give a reason like: $\neg(R(a) \wedge R(b))$.

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

Implementation of Oracles. When modules are described using some well studied languages, we often have existing efficient Valid Acceptance Procedures used in solver constructions, e.g., Well-Founded Model computation for ASP, Arc-Consistency checking for CP, Theory Propagation for various SMT theories, a lifted version of Unit Propagation [9] for FO, etc. In these cases, corresponding techniques can be used to implement oracles to accept/reject partial structures and to provide reasons and advices accordingly. More examples of Valid Acceptance Procedures used in practice are given in Sect. 4.

3.3 Requirements on the Solver

In this section, we discuss properties that a solver has to satisfy. Although the solver can be realized by many practical systems, for them to work in an orderly fashion and for algorithm to converge to a solution fast, it has to satisfy certain properties. First, the solver has to be online since the oracles keep adding reasons and advices to it. Furthermore, to guarantee termination, the solver has to guarantee progress, which means it either reports a proper extension of the previous partial structure or, if not, the solver is guaranteed to never return any extension of that previous partial structure later on. Now, we give the requirements on the solver formally.

Definition 8 (Complete Online Solver). *A solver S is* complete and online *if the following conditions are satisfied by S:*

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

A solver as above is (1) Sound: it returns partial structures that at least do not falsify any of the constraints, and (2) Complete: it reports unsatisfiability only when unsatisfiability is detected and not when, for example, some heuristic has failed to find an answer or some time limit is reached. Also, for finite structures, such a solver guarantees that our algorithm either reports unsatisfiability or finds a solution to modular system M and instance structure $\mathcal{A}$. Proposition 1 gives the exact correspondence in this regard.

3.4 Lazy Model Expansion Algorithm

In this section, we present an iterative algorithm to solve model expansion tasks for modular systems. Algorithm 1 takes an instance structure and a modular system (and its CCAV oracles) and integrates them with a complete online solver to solve a model expansion task in an iterative fashion. The algorithm works by accumulating reasons and advices from oracles and gradually converging to a solution to the problem.

The role of the reasons is to prevent some bad structures and their extensions from being proposed more than once, i.e., when a model is deducted to be bad by an oracle, a new reason is provided by the oracle and added to the solver such that all models of the system satisfy that reason but the "bad" structure does not. The role of an advice is to provide useful information to the solver (satisfied by all models) but not yet satisfied by partial structure $\mathcal{B}$. Informally, an advice is in form "if Pre then Post", where "Pre" corresponds to something already satisfied by current partial structure $\mathcal{B}$ and "Post" is something that is always satisfied by all models of the modular system satisfying the "Pre" part, but not yet satisfied by the partial structure $\mathcal{B}$. It essentially tells the solver that "Post" part is satisfied by all intended structures (models of the system) extending $\mathcal{B}$, thus helping the solver to accelerate its computation in its current direction.

The role of the solver is to provide a possibly good partial structure to the oracles, and if none of the oracles reject the partial structure, keep extending it until we find a solution or conclude none exists. If the partial structure is rejected by some oracle, the solver gets a reason from that oracle for rejection and tries some other partial structure. The solver also gets advices from oracles to accelerate the search.

Proposition 1 (Correctness). *Algorithm 1 is sound and complete for finite structures, i.e., given a modular system M with CCAV oracles, a complete online solver S and a finite instance structure $\mathcal{A}$:*

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

Algorithm 1: Lazy Model Expansion Algorithm

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

4 Case Studies: Existing Frameworks

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

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

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

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

4.1 Modelling DPLL(T)

DPLL(T) [4] system is an abstract framework to model the lazy SMT approach. It is based on a general DPLL(X) engine, where X can be instantiated with a

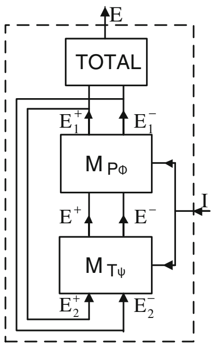


Fig. 2. Modular System Representing the DPLL(T) System on Input Formula $\phi \wedge \psi$

theory T solver. DPLL(T) engine extends the Decide, UnitPropagate, Backjump, Fail and Restart actions of the classic DPLL framework with three new actions: (1) **TheoryPropagate** gives literals that are T-consequences of current partial assignment, (2) T**-Learn** learns T-consistent clauses, and (3) T**-Forget** forgets some previous lemmas of theory solver.

To participate in the DPLL(T) solving architecture, a theory solver provides three operations: (1) taking literals that have been set true, (2) checking if setting these literals true is T-consistent and, if not, providing a subset of them that causes inconsistency, (3) identifying some currently undefined literals that are T-consequences of current partial assignment and providing a justification for each. More details can be found in [4].

The modular system representing the DPLL(T) system on the input formula $\phi \wedge \psi$ is shown in Fig. 2, where $\sigma = I$, $\varepsilon = E$, and $E^+ \cup E^- \cup E_1^+ \cup E_1^- \cup E_2^+ \cup E_2^-$ is the internal vocabulary of the module. Also, there are feedbacks from E_1^+ to E_2^+ and from E_1^- to E_2^-. The set of symbols in E^+ and E^- (same for E_1^+ and E_1^-, E_2^+ and E_2^-) semantically represents a partial interpretation of the symbols in the expansion vocabulary, i.e., E^+ (resp. E^-) represents the positive (resp. negative) part of the partial interpretation.

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

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

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

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

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

where D is as before and (R^+, R^-) is the result of Theory Propagation on ψ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup E_2^{+\mathcal{B}} \cup \neg E_2^{-\mathcal{B}}$, and $R \models_T \psi$ denotes that ψ is T-satisfiable under the set of facts R. Note that the satisfiability test is not necessarily complete. It can be done in different degrees depending on the complexity of different theories.

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

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

$$DPLL(T)_{\phi\wedge\psi} := \pi_{\{I,E\}}(((M_{T_\psi} \rhd M_{P_\phi})[E_1^+ = E_2^+][E_1^- = E_2^-]) \rhd TOTAL). \quad (2)$$

To show that the combined module $DPLL(T)_{\phi\wedge\psi}$ is correct, consider any model of the modular system. Note that for both modules M_{P_ϕ} and M_{T_ψ}, the outputs always contain all the information that the inputs have, i.e., for any structure $\mathcal{B}$ in the module M_{P_ϕ}, we have $E_1^{+\mathcal{B}} \supseteq E^{+\mathcal{B}}$ and $E_1^{-\mathcal{B}} \supseteq E^{-\mathcal{B}}$, and for any structure $\mathcal{B}$ in M_{T_ψ}, we have $E^{+\mathcal{B}} \supseteq E_2^{+\mathcal{B}}$ and $E^{-\mathcal{B}} \supseteq E_2^{-\mathcal{B}}$. Furthermore, from the semantics of the feedback operator, we know that $E_1^{+\mathcal{B}} = E_2^{+\mathcal{B}}$ and $E_1^{-\mathcal{B}} = E_2^{-\mathcal{B}}$. Thus, we have $E^{+\mathcal{B}} = E_1^{+\mathcal{B}} = E_2^{+\mathcal{B}}$ and $E^{-\mathcal{B}} = E_1^{-\mathcal{B}} = E_2^{-\mathcal{B}}$. Moreover, from the definition of module $TOTAL$, we know that $(E_1^{+\mathcal{B}}, E_1^{-\mathcal{B}})$ represents a total interpretation of the symbols in E and $E^{\mathcal{B}} = E_1^{+\mathcal{B}}$. Finally, from the definitions of M_{P_ϕ} and M_{T_ψ} on encodings of total interpretations, we can conclude that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$. On the other hand, it is easy to see that for any structure $\mathcal{B}$ such that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$, $\mathcal{B}$ is in $DPLL(T)_{\phi\wedge\psi}$.

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

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

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

Oracle O_{TOTAL} accepts a partial structure $\mathcal{B}$ iff $E_1^{+\mathcal{B}} \cap E_1^{-\mathcal{B}} = \emptyset$, $E_1^{+\mathcal{B}} \cup E_1^{-\mathcal{B}} = D$, and $E^{\mathcal{B}} = E^{+\mathcal{B}}$. If $\mathcal{B}$ is rejected, O_{TOTAL} returns $\bigwedge_{\omega\in\Omega'} \omega$ as the reason, where Ω' is any non-empty subset of the set $\Omega = \{E_1^+(d) \Leftrightarrow \neg E_1^-(d) \mid d \in D, \mathcal{B} \not\models E_1^+(d) \Leftrightarrow \neg E_1^-(d)\} \cup \{E(d) \Leftrightarrow E_1^+(d) \mid d \in D, \mathcal{B} \not\models E(d) \Leftrightarrow E_1^+(d)\}$. O_{TOTAL} returns the set Ω as the set of advices when $\mathcal{B}$ is the empty expansion of the instance structure, and the empty set otherwise.[5] Clearly, O_{TOTAL} is a CCAV oracle.

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

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

By the definition of M_{T_ψ}, we know that $\mathcal{B}$ falsifies the reason and all models of M_{T_ψ} satisfy the reason. Thus, O_T is complete and constructive. O_T may also return some advices in the same form as any ω above such that $\mathcal{B}$ satisfies the left hand side of the implication, but not the right hand side. Also, since the outputs of M_{T_ψ} always subsume the inputs, O_T may also return the set $\{E_2^+(d) \supset E^+(d) \mid d \in D, \mathcal{B} \models E_2^+(d), \mathcal{B} \not\models E^+(d)\} \cup \{E_2^-(d) \supset E^-(d) \mid d \in D, \mathcal{B} \models E_2^-(d), \mathcal{B} \not\models E^-(d)\}$ as the set of advices.[6] Clearly, all the structures in M_{T_ψ} satisfy all sets of advices. Hence, O_T is an advising oracle. Finally, O_T always makes the correct decision for a total structure and rejects a partial structure only when it falsifies the constraints for M_{T_ψ}. O_T never rejects any good partial structure $\mathcal{B}$ (although it may accept some bad non-total structures). Therefore, O_T is a verifying oracle.

Proposition 2. *1. Modular system $DPLL(T)_{\phi\wedge\psi}$ is the set of structures $\mathcal{B}$ such that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$.*
2. Solver S is complete and online.
3. O_P, O_T, and O_{TOTAL} are CCAV oracles.
4. Algorithm 1 on modular system $DPLL(T)_{\phi\wedge\psi}$ associated with oracles O_P, O_T, O_{TOTAL}, and the solver S models the solving procedure of the DPLL(T) system on input formula $\phi \wedge \psi$.

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

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

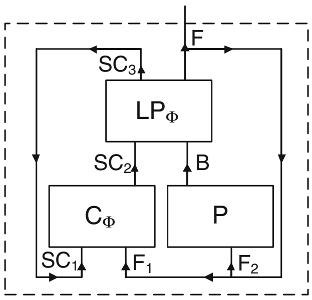


Fig. 3. Modular System Representing an ILP Solver

DPLL(T) architecture is known to be very efficient and many solvers are designed to use it, including most SMT solvers [10]. The DPLL(Agg) module [11] is suitable for all DPLL-based SAT, SMT and ASP solvers to check satisfiability of aggregate expressions in DPLL(T) contexts. All these systems are representable in our modular framework.

4.2 Modelling ILP Solvers

Integer Linear Programming solvers solve optimization problems. In this paper, we model ILP solvers which use general branch-and-cut method to solve *search* problems instead, i.e., when the target function is constant. We show that Algorithm 1 models such ILP solvers. ILP solvers with other methods and Mixed Integer Linear Programming solvers use similar architectures and, thus, can be modelled similarly.

The search version of general branch-and-cut algorithm [5] is as follows:

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

We use the modular system shown in Fig. 3 to represent the ILP solver. The module C_ϕ takes a set of variable assignments F_1 and a set of cutting planes SC_1 as inputs and returns another set of cutting planes SC_2. When all the assignments in F_1 are integral, SC_2 is equal to SC_1, and if not, SC_2 is the union of SC_1 and a cutting plane violated by F_1 w.r.t. the set of linear constraints $SC_1 \cup \phi$. The module P takes a set of assignments F_2 as input and outputs a set of range constraints $B = \{B_x \mid F_2(x) \notin \mathcal{Z}\}$, where B_x is non-deterministically chosen from the set $\{x \leq \lfloor F_2(x) \rfloor,\ x \geq \lceil F_2(x) \rceil\}$. The module LP_ϕ takes the set of cutting planes SC_2 and the set of range constraints B as inputs and outputs the set of cuttings planes SC_3 and the set of assignments F in a deterministic way such that SC_3 is the union of SC_2 and B, and F is a total assignment satisfying $SC_2 \cup B \cup \phi$. LP_ϕ is undefined when $SC_2 \cup B \cup \phi$ is inconsistent. We define the compound module ILP_ϕ to be:

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

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

So there is one-to-one correspondence between the solutions of the ILP problem with input ϕ, and the models of the modular system ILP_ϕ. We compute a model of this modular system by associating modules with oracles (O_c, O_p and O_{lp}) and introducing a solver S that interacts with those oracles. Each oracle rejects a partial structure $\mathcal{B}$ if it contradicts the corresponding module definition and in this case, the reason for the rejection is provided. For example, when $F_2^{\mathcal{B}}$ is non-integral, O_p rejects $\mathcal{B}$ and gives the reason $\lfloor F_2(x)^{\mathcal{B}} \rfloor < F_2(x) < \lceil F_2(x)^{\mathcal{B}} \rceil \supset B(\text{“}F_2(x) \leq \lfloor F_2(x)^{\mathcal{B}} \rfloor\text{”}) \vee B(\text{“}F_2(x) \geq \lceil F_2(x)^{\mathcal{B}} \rceil\text{”})$, for some non-integral variable x, and O_c rejects $\mathcal{B}$ with the reason $(\bigwedge_{l \in L} SC_1(l)) \wedge (\bigwedge_x F_1(x) = F_1(x)^{\mathcal{B}}) \supset SC_2(c)$, where c is the cutting plane that violates F_1, and L is a subset of SC_1 such that F_1 is the intersection of some constraints in $L \cup \phi$. Full details of the oracles are omitted due to space consideration. The solver S accepts the full propositional language with atomic formulas being either boolean variables or range constraints. In addition, S can assign numerical values (for F) according to the set of derived range constraints.

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

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

4.3 Modelling Constraint Answer Set Solvers

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

In [12], the authors extend the language of ASP and its reasoning method to avoid grounding of variables with large domains by using constraint solving techniques. The algorithm uses ASP and CP solvers as black boxes and non-deterministically extends a partial solution to the ASP part and checks it with the CP solver. Paper [13] presents another integration of answer set generation and constraint solving in which a traditional DPLL-like backtracking algorithm is used to embed the CP solver into the ASP solving.

Recently, the authors of [6] developed an improved hybrid solver which supports advanced backjumping and conflict-driven nogood learning (CDNL) techniques. They show that their solver's performance is comparable to state-of-the-art SMT solvers. Paper [6] applies a partial grounding before running its algorithm, thus, it uses an algorithm on propositional level. A brief description of this algorithm follows: Starting from an empty set of assignments and nogoods, the algorithm gradually extends the partial assignments by both unit propagation in ASP and constraint propagation in CP. If a conflict occurs (during either unit propagation or constraint propagation), a nogood containing the corresponding unique implication point (UIP) is learnt and the algorithm backjumps to the decision level of the UIP. Otherwise, the algorithm decides on the truth value of one of the currently unassigned atoms and continues to apply the propagation. If the assignment becomes total, the CP oracle queries to check whether this is indeed a solution for the corresponding constraint satisfaction problem (CSP). This step is necessary because simply performing constraint propagation on the set of constraints, i.e., arc-consistency checking, is not sufficient to decide the feasibility of constraints.

The modular model of this solver is very similar to the one in Fig. 2, except that we have module ASP_ϕ instead of SAT_ϕ and CP_ψ instead of ILP_ψ. The compound module $CASP_{\phi\wedge\psi}$ is defined as:

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

As a CDNL-like technique is also used in SMT solvers, the above algorithm is modelled similarly to Sect. 4.1. We define a solver S to be a CDNL-based ASP solver. We also define modules ASP_ϕ and CP_ψ to deal with the ASP part and the CP part. They are both associated oracles similar to those described in

Sect. 4.1. We do not include the details here as they are similar to the ones in Sect. 4.1.

Note that one can add reasons and advices to an ASP solver safely in the form of conflict rules because stable model semantics is monotonic with respect to such rules. Also, practical CP solvers do not provide reasons for rejecting partial structures. This issue is dealt with in [6] by wrapping CP solvers with a conflict analysis mechanism to compute nogoods based on the first UIP scheme.

5 Related Work and Conclusion

There are many papers on modularity in declarative programming, we only review the most relevant ones. The authors of [8] proposed a multi-language framework for constraint modelling. That work was the initial inspiration of [2], but the authors extended the ideas significantly by developing a model-theoretic framework and introducing a feedback operator that adds a significant expressive power.

An early work on adding modularity to logic programs is [14]. The authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. The ideas are further generalized in [15] by considering the concept of modules in declarative programming and introducing modular equivalence in normal logic programs under the stable model semantics. This line of work is continued in [16] to define modularity for disjunctive logic programs. These ideas are further generalized in [17] to allow (mutually) recursive calls between modules. There are also other approaches to adding modularity to ASP languages and ID-Logic as described in [18–20].

The works mentioned earlier focus on the theory of modularity in declarative languages. However, there are also papers that focus on the practice of modular declarative programming and, in particular, solving. These generally fall into one of the two following categories. The first category consists of practical modelling languages which incorporate other modelling languages. For example, X-ASP [21] and ASP-PROLOG [22] extend Prolog with ASP, CP techniques are incorporated into ASP solving in [12,13] and [6]. Also, ESSENCE [7] and Zinc [23] are CP languages extended with features from other languages. However, these approaches give priority to the host language while our modular setting gives equal weight to all modelling languages that are involved. It is important to note that, even in the presence of this distinction, such works have been very important in the development of this paper because they provide guidelines on how a practical solver deals with efficiency issues. The second category is related to multi-context systems. In [24], the authors introduce non-monotonic bridge rules to the contextual reasoning and originated an interesting and active line of research followed by many others for solving or explaining inconsistencies in non-monotonic multi-context systems [25–27]. However, these papers do not consider the model expansion task. Moreover, the motivations of these works originate from distributed or partial knowledge, e.g., when agents interact or when trust or privacy issues are important. Despite these differences, the field of multi-context systems is very relevant to our research. Investigating this connection is an important future research direction.

Conclusion. We took a language-independent view towards modular problem solving and designed an algorithm to solve search problems described as modular systems. Our model-theoretic approach allows us to abstract away from particular languages of the modules. We performed several case studies of our algorithm in relation to existing systems such as DPLL(T), ILP and ASP+CP. We demonstrated that, for the task of model expansion, our algorithm generalizes the work of these solvers. We also demonstrated how Valid Acceptance Procedures from different communities could be used to implement oracles for modules to achieve efficient solving. For example, the procedures of Well-Founded Model computation and Arc-Consistency checking can be used to implement oracles for the ASP and CP languages to construct an efficient combined solver (corresponding to the state-of-the-art combination of ASP and CP [6]).

Our general approach for solving modular systems is applicable to systems such as Business Process Planners and their variants including Logistics Service Provider, Manufacturer Supply Chain Management, and Mid-size Businesses Relying on External Web Services and Cloud Computing. With the increasing use of service-oriented architecture, such modular systems will become increasingly more applicable. We believe we are taking important initial steps addressing the core aspect of this complex multi-dimensional problem. As a future direction, we plan to develop a prototype implementation of our algorithms.

References

1. Mitchell, D.G., Ternovska, E.: A framework for representing and solving NP search problems. In: Proceedings of AAAI, pp. 430–435 (2005)
2. Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 259–274. Springer, Heidelberg (2011)
3. Niemelä, I.: Integrating answer set programming and satisfiability modulo theories. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, p. 3. Springer, Heidelberg (2009)
4. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM **53**, 937–977 (2006)
5. Pardalos, P., Resende, M.: Handbook of applied optimization, vol. 126. Oxford University Press, New York (2002)
6. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 235–249. Springer, Heidelberg (2009)
7. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: a constraint language for specifying combinatorial problems. Constraints **13**, 268–306 (2008)

8. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 155–168. Springer, Heidelberg (2009)
9. Vaezipoor, P., Mitchell, D., Mariën, M.: Lifted unit propagation for effective grounding. In: 19th International Conference on Applications of Declarative Programming and Knowledge Management (2011). CoRR abs/1109.1317
10. Sebastiani, R.: Lazy satisfiability modulo theories. JSAT **3**, 141–224 (2007)
11. De Cat, B., Denecker, M.: DPLL(Agg): An efficient SMT module for aggregates. In: LaSh 2010 Workshop (2010)
12. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
13. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Ann. Math. Artif. Intell. **53**, 251–287 (2008)
14. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Dix, J., Furbach, U., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 290–309. Springer, Heidelberg (1997)
15. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: The Proceedings of NMR 2006, pp. 10–18 (2006)
16. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 175–187. Springer, Heidelberg (2007)
17. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular nonmonotonic logic programming revisited. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 145–159. Springer, Heidelberg (2009)
18. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)
19. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: SEA, pp. 41–55 (2007)
20. Denecker, M., Ternovska, E.: A logic of non-monotone inductive definitions. Trans. Comput. Logic **9**(2), 1–51 (2008)
21. Swift, T., Warren, D.S.: The XSB System (2009)
22. Elkhatib, O., Pontelli, E., Son, T.C.: $\mathbb{ASP} - \mathbb{PROLOG}$: a system for reasoning about answer set programs in prolog. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 148–162. Springer, Heidelberg (2004)
23. Garcia de la Banda, M., Marriott, M., Rafeh, R., Wallace, M.: The modelling language zinc. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 700–705. Springer, Heidelberg (2006)
24. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. Proceedings of the 22nd AAAI Conference on Artificial Intelligence, vol. 1, pp. 385–390. AAAI Press, Vancouver (2007)
25. Bairakdar, S.-D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The DMCS solver for distributed nonmonotonic multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 352–355. Springer, Heidelberg (2010)

26. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in multi-context systems. In: Lin, F., Sattler, U., Truszczynski, M. (eds.), Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference (KR 2010), Toronto, Ontario, Canada, 9–13 May 2010. AAAI Press (2010)
27. Eiter, T., Fink, M., Schüller, P.: Approximations for explanations of inconsistency in partially known multi-context systems. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 107–119. Springer, Heidelberg (2011)

INAP Application Papers

FdConfig: A Constraint-Based Interactive Product Configurator

Denny Schneeweiss[1(✉)] and Petra Hofstedt[2]

[1] Berlin Institute of Technology, Berlin, Germany
`dschneeweiss@mailbox.tu-berlin.de`
[2] Brandenburg University of Technology, Cottbus, Germany
`hofstedt@informatik.tu-cottbus.de`

Abstract. We present a constraint-based approach to *interactive* product configuration *with arithmetic constraints and support for optimization*. Our configurator tool *FdConfig* is based on feature models (from software product line engineering) for the representation of the valid product variants. Such models can be directly mapped into constraint satisfaction problems and dealt with by appropriate constraint solvers. During the interactive configuration process the user generates new constraints as a result of his configuration decisions and even may retract constraints posted earlier. We discuss the configuration process, explain the underlying techniques and show optimizations.

1 Introduction

Product lines for mass customization [26] allow to fulfill the needs and requirements of the individual consumer while keeping the production cost low. They enhance extensibility and maintenance by re-using the common core of the set of all products.

Product configuration describes the process of specifying a product according to user-specific needs based on the description of all possible (valid) products (the search space). When done *interactively*, the user specifies the features of the product step-by-step according to his requirements, thus, gradually shrinking the search space of the configuration problem. This interactive configuration process is supported by a software tool, the *configurator*.

In this paper we present an approach to interactive product configuration based on constraint programming techniques. Building on constraints enables us to equip our interactive product configurator *FdConfig* with functionality and expressiveness exceeding traditional approaches [1,7,8,12,13,21,23,24]. These either support only Boolean constraints or a very restricted form of arithmetic constraints or do not take an interactive configuration process into consideration or both (for a discussion see Sect. 2). Providing an extended functionality *and* supporting interactive configuration, however, comes with the cost of performance penalty which must be dealt with in turn.

The paper is structured as follows: In Sect. 2 we briefly review the area of interactive configuration methods and discuss related work. Section 3 introduces important notions from the constraint paradigm as needed for the discussion of our approach. We present the constraint-based interactive product configurator *FdConfig* in Sect. 4. There, we introduce *FdFeatures*, a language for the definition of feature models, it's transformation into constraint problems, and the configuration process using *FdConfig*. Furthermore, we discuss optimizations and improvements by analyses and multithreading. Section 5 draws a conclusion and points out directions of future research.

2 Interactive Configuration Methods

An interactive product configurator is a tool which allows the user to specify a product according to his specific needs based on the common core of the set of all products of a product line. This process can be done interactively, i.e. in a step-wise fashion, thus gradually shrinking the search space of the configuration problem.

For the sake of applicability and user-friendliness, a configurator requires a number of properties like backtrack-freeness, completeness, order-independent retraction of decisions, short response times and others. These strongly depend on the method[1] underlying the configurator system. Cost optimization and arithmetic constraints are a desired functionality too, but these are seldom supported or only provided in a very restricted form.

While *completeness* ensures that no solutions are lost, *backtrack-freeness* [7, 23] guarantees that the configurator only offers decision alternatives for which solutions remain. Thus, the user can always generate a valid solution from the current configuration state and does not need to unwind a decision (i.e. he, the user, does not need to backtrack his decisions during the configuration process). The *Calculate Valid Domains (CVD) function* [7] of a configurator realizes this latter property.

Feature models are particularly used in the context of software product line engineering to support the reuse when building software-intensive products. However, they are of course applicable to many other product line domains. They stem from the *feature oriented domain analysis methodology* (FODA) [14].

A feature model describes a product domain by a combination of features, i.e. specific aspects of the product which the user can configure by instantiation and further constraints. A product line is given by the set of possible combinations of feature alternatives.

The semantics of feature models is typically mapped to propositional logics [11] and can accordingly be mapped onto a restricted class of constraint satisfaction problems (cf. Sect. 3), namely constraints of the Boolean domain. While many approaches in the literature (e.g. [7,8,12]) only consider constraints of the Boolean domain (including equality constraints), Benavides et al. [1] discuss the realization of arithmetic computations and cost optimization in a feature model (by so-called "extra-functional features") which can be represented by general constraint problems.

[1] For a discussion of the solutions methods, see below.

Solution techniques applied to the interactive configuration problem have been compared by Hadzic et al. [7,8] and Benavides et al. [2]. They mainly distinguish approaches based on propositional logic on the one hand and on constraint programming on the other hand.

When using *propositional logic* based approaches, configuration problems are restricted to logic connectives and equality constraints (see e.g. [7,24]). Note that arithmetic expressions are excluded because of the underlying solution methods. These approaches perform in two steps. First, the feature model is translated into a propositional formula. In the second step the formula is solved (satisfiability checking, computation of solutions) by appropriate solvers, in particular *SAT solvers* (as in [12]) and *BDD-based solvers* (see e.g. [8,23]). BDD-based solvers translate the propositional formula into a compact representation, the binary decision diagram (BDD). While many operations on BDDs can be implemented efficiently, the structure of the BDD is crucial as a bad variable ordering may result in exponential size and, thus, in memory blow up. Therefore the compilation of the BDD is done in an offline phase, so a suitable variable ordering can be found and the BDD's size becomes reasonably small.

Feature models can be naturally mapped into *constraint systems*, in particular into CSPs. There are some approaches [1,24] using this correspondence to deal with interactive configuration problems. These typically work as follows: The feature model is translated into a constraint satisfaction problem (CSP, see Definition 1 below) and afterwards analysed by a CSP solver. Using this approach, no pre-compilation is necessary. In general it is possible to use predicate logic expressions and arithmetics in the feature definitions, even if this is not realized in the above mentioned approaches.

The system of [13] supports a restricted class of arithmetic constraints but is not intended to be used interactively. Feature model analysis is the focus of the FAMA-framework [25] which incooperates SAT- and BSD-based techniques as well as CSP solvers. The web-based SPLOT-tool described in [18] supports interactive, backtracking-free configuration of Boolean feature models. SPLOT uses a SAT solver for the configuration but also employs a BSD-based solver for feature models analysis.

Soininen and Niemelä [21,22] present an approach to configuration based on answer set programming and discuss its relationship to (C)LP as well as complexity issues. In general their approach is suitable for interactive scenarios, but supports only Boolean constraints.

Transformations of feature models into programs of CLP languages (i.e. Prolog systems with constraints) have been shown recently in [15,17]. However, beside the transformation target being different from ours, these approaches do not focus on using these methods for configuration done *interactively*.

Since our *FdConfig* tool aims primarily at the software engineering community as the main users of feature models, we decided in favour of a Java-implementation (facilitating the Java-based Choco-library), which would make later integration with common software development infrastructure like *Eclipse* easier.

Benavides et al. [2] elaborately compare the approaches sketched above, particularly with respect to *performance* and *expressiveness or supported operations, resp.* They point out that CSP-based approaches, in contrast to others, can allow extra functional features [1,14] and, in addition, arithmetic and optimization. Furthermore, they state that "the available results suggest" that constraint-based and propositional logic-based approaches "provide similar performance", except for the BDD-approach which "seems to be an exception as it provides much faster execution times", but with the major drawback of BDDs having worst-case exponential size.

Extended feature models *with* numerical attributes, arithmetic, and optimization are denominated as an important challenge in *interactive* configuration by Benavides et al. [2]. Our approach aims at this challenge. We realize an *interactive* configuration approach *with extended features like arithmetics on product attributes and support for optimizations.* The main idea is to follow the constraint-based approach while using the combination of different constraint methods and concurrency to deal with the computational cost. At this, a major challenge is to support the user when making and withdrawing decisions in an interactive process.

3 Constraint Programming

Feature models can directly be mapped into corresponding constraint problems. We will discuss this approach more detailed in Sect. 4.1 but introduce the necessary notions from the constraint paradigm here.

Constraints are predicate logic formulae which express relations between the elements or objects of the problem. They are classified into *constraint domains* (see [9,19]), e.g. linear arithmetic constraints on reals, Boolean constraints and finite domain constraints. This partitioning is due to the different applicable constraint solution algorithms implemented in so-called constraint solvers (see below).

Considering feature models as constraint problems, the domains of the involved variables are a priori finite.[2] Thus, we consider a particular class of constraints: *finite domain constraints.* Finite domain constraint problems are given by means of constraint satisfaction problems.

Definition 1 (CSP). *A* Constraint Satisfaction Problem (CSP) *is a triple* $P = (X, D, C)$*, where* $X = \{x_1, \dots, x_n\}$ *is a finite set of variables,* $D = (D_1, \dots, D_n)$ *is an n-tuple of their respective finite domains, and* C *is a conjunction of constraints over* X*.*

Definition 2 (solution). *A* solution *of a CSP* P *is a valuation* $\varsigma : X \rightarrow \bigcup_{i \in \{1,\dots,n\}} D_i$ *with* $\varsigma(x_i) \in D_i$ *which satisfies the constraint conjunction* C*.*

A CSP can have one solution or a number of solutions, or it can be unsatisfiable. Optimization functions may also be given which specify optimal solutions.

[2] An extension to infinite domains would be possible, in general.

Example 1. Consider a CSP $P = (X, D, C)$ with the set X of variables with $X = \{Cost, Color, Band\}$ and their respective domains $D = (D_{Cost}, D_{Color}, D_{Band})$ with $D_{Color} = \{Red, Gold, Black, Blue\}$, $D_{Cost} = \{0, ..., 1500\}$, and $D_{Band} = \{700, 900, 1000\}$.

$C = (Band = 700 \rightarrow Color = Blue) \wedge (Cost = Band + 500)$ is a conjunction of constraints over the variables from X.

Solutions of the CSP P are, e.g. ς_1 with $\varsigma_1(Cost) = 1200$, $\varsigma_1(Color) = Blue$, and $\varsigma_1(Band) = 700$, also denoted by $\varsigma_1 = \{Cost/1200, Color/Blue, Band/700\}$, and $\varsigma_2 = \{Cost/1400, Color/Red, Band/900\}$.

Constraint solvers are algorithms, which are able to check the satisfiability of constraints and to compute solutions and implications of constraints.

CSPs are typically solved by narrowing the variable's domains using search nested with consistency techniques (e.g. node, arc, and path consistency). Given a CSP, in the first step consistency techniques are applied. Such consistency checking algorithms work on n-ary constraints and try to remove values from the variables domains which cannot be elements of solutions. Afterwards, search is initiated, e.g. using backtracking, where we assign domain values to variables and perform consistency techniques to narrow the other variable's domains again. This search process is controlled by heuristics on variable and value ordering (for the complete process, see [19]).

There are some finite domain constraint solver libraries available, for example the JAVA-libraries CHOCO [3] and JACOP [10] as well as the C++-library GECODE [6]. We decided in favour of the CHOCO library, as it is under continuous development and available under BSD-license. Moreover, it's pure JAVA nature fitted our purposes best (see above).

Additionally, we need the notions of global consistency and of valid domains.

Definition 3 (global consistency, see [19]). *A CSP is i-consistent iff given any consistent instantiation of $i - 1$ variables, there exists an instantiation of any ith variable such that the i values taken together satisfy all of the constraints among the i variables. A CSP $P = (X, D, C)$ is globally consistent, if it is i-consistent for every i, $1 \leq i \leq n$, where n is the number of variables of C.*

Definition 4 (valid domains). *Given a CSP P, the* valid domains *of P is an n-tuple $D_{vd} = (D_{vd,1}, \ldots, D_{vd,n})$ such that each $D_{vd,i} \subseteq D_i$ contains exactly the values which are elements of solutions of P.*

So, if a CSP is globally consistent, then its domains are valid domains.

Example 2. (continuation of Example 1) The valid domains of the CSP P is $D_{vd} = (\{1200, 1400\}, D_{Color}, \{700, 900\})$.

4 The Interactive Configurator *FdConfig*

Our approach on interactive configuration consists of two phases: In the first phase a feature model is analysed and then transformed into a CSP and passed to the CHOCO solver. Afterwards the interactive configuration phase follows.

Figure 1 illustrates the analysis and transformation phase. *FdConfig* uses *FdFeatures* files as input. *FdFeatures* is a textual domain specific language for extended feature models which supports integer feature attributes and arithmetic constraints. An *FdFeatures* parser reads the input-file and creates the feature model which is transformed into a Choco CSP. Section 4.1 describes the language *FdFeatures* and the transformations in greater detail. Additionally, a quick pre-calculation of the variable's domains is performed. It generates redundant constraints which, nevertheless, help to improve the solver's performance. In constraint programming, the generation of redundant constraints from a given constraint problem is a frequently used method which helps to speed up the solver (see [19], Sect. 12.4.5).[3] This *domain analysis* is covered in Sect. 4.2.

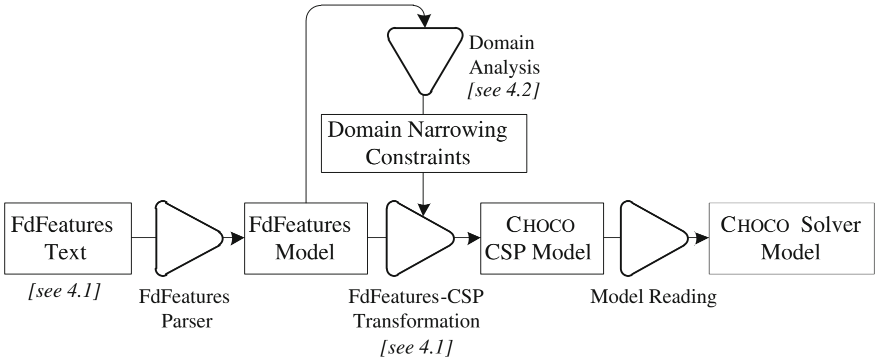


Fig. 1. Transformations performed before the user can start configuring

In the second phase, the generated CSP is passed to the Choco solver which reads the model and creates an internal representation from it: the solver model. Then the solver is started to perform an initial calculation of consequence decisions that yield from the constraints in the *FdFeatures* model. Afterwards, the user can start with the interactive configuration. The implementation of this process is explained in Sect. 4.3. Section 4.4 describes the reduction of response times by using multithreading.

4.1 *FdFeatures* Models and CSPs

FdConfig provides *FdFeatures* as a language for the definition of feature models based on the approach of [5]. *FdFeatures* borrows from the Textual Variability Language (TVL, [4]) but was adapted for our needs (e.g. including support for the realization of the user interface, certain detailed language elements and syntactic sugar). *FdFeatures* has been implemented using Xtext [27].

[3] Note that the elimination of verification-irrelevant features and constraints (i.e. "redundant relationships", [28]) from feature models with the aim of reducing the problem size is a completely different concept.

An *FdFeatures feature model* in general has a tree structure, i.e. there is a distinguished root feature which stands for the item to be configured, but apart from this behaves like any other feature. The model may have additional constraints between (sub-)features and their attributes which, in fact, makes the tree a general graph. Nevertheless, the tree structure is dominant.

A feature may consist of sub-features and attributes (both in general optional), where, following the approach of [5], the sub-features can be organized in *feature groups*. A feature group allows to describe whether one, some, or all of the sub-features must be included in the configured product.

With similar effects, *features* can be specified to be **mandatory** or **optional**. Furthermore, features may exclude or require other features.

Example 3. Consider the cut-out of a feature model of events organized by an event agency in Listing 4.1.[4] For an event (the root feature) we can optionally order a band and a stage, but we must order a carpet (e.g. for a film premiere or a wedding) and colored balloons. These are all modeled as sub-features (which are not organized in a feature group here). Ordering a band makes a stage necessary, expressed by the **requires**-statement in Line 11.

FdFeatures supports three kinds of feature *attributes:* integers, enumerations, and Boolean values.

Example 4. (continuation of Example 3) The feature *Carpet* is determined by several attributes, including an enumeration attribute *Color*, whose domain elements must be given explicitly and a Boolean attribute *SlipResistance*.[5]

Length and *Breadth* are integer attributes. While *Breadth* is specified by an interval, *Length* is unbounded. As we can see by the attributes of *ColoredBalloons*, the domain of an integer attribute can also be specified by a finite set (*Amount*, Line 20) or even by an arithmetic formula (*Cost*, Line 25). The definition of Boolean attributes is also possible using Boolean expressions (but is left out in our example).

The domain definition of *PriceReduction.Coupon* (Lines 22, 23) uses *Guards* to define the attribute domain depending on the configuration state (**ifIn** and **ifOut** correspond to selected and deleted, resp.) of the feature (here *PriceReduction*). Furthermore, it is possible to define *constraints* on attributes and features, also accessing the configuration state of a feature as shown in Line 15. This constraint makes sure that the Blues-band plays in an adequate ambiance.

The *transformation* into a Choco CSP is straightforward, for details see [20]. In general, our transformation is similar to these of [15,17]. Differences come from the fact that the transformation target of these approaches are CLP

[4] The description of certain features and attributes, which are not necessary for the understanding of this example and the concepts behind, is left out and represented by "..." in the program.

[5] Note that domain elements of an enumeration can optionally be assigned explicit integer values as e.g. done for the attribute *Band.Type* in Line 12.

Listing 4.1 Feature model of an event organized by an event agency (cut-out)

```
root feature Event {
 enum Discount in {Gold = 8, Staff = 3, None = 0};
  feature Carpet {
     int Length;
     int Breadth in [50..300];
     enum Color in {Red, Gold, Black, Blue};
     bool SlipResistance;
     int Cost is ...
   }
   feature Band : optional {
     requires Stage;
     enum Type in {Classic = 1000, Blues=700, Rock=900};
   }
   feature Stage : optional { ... }
   constraint BluesOnBlueCarpet
      Band is selected and Band.Type = Blues →
         Carpet.Color = Blue
   }
   feature ColoredBalloons {
     int Amount in {500, 1000, 2500, 5000, 10000};
     feature PriceReduction {
       int Coupon ifIn:  is 1000
                  ifOut: is 0;
     }
     int Cost is Amount * 3 − PriceReduction.Coupon;
   }
   int OverallCost is Carpet.Cost + Band.Type + ... +
     (ColoredBalloons.Cost/100 + ...) * (100−Discount)/100;
}
```

languages and they aim at feature model analysis in contrast to interactive configuration, as does *FdConfig*. We show an example of the generated CSP in a mathematical notation and leave out the CHOCO constraint syntax for reasons of space limitations.

Example 5. The following CSP is generated from Listing 4.1 (where *CBal* stands for *ColoredBalloons*, *PRed* for *PriceReduction*, and *SRes* for *SlipResistance* resp.). Note that we do not enumerate the set of variables X explicitly and give the domains D by means of element constraints $C_{Domains}$.

$$\begin{aligned}
CSP \quad &= C_{Domains} \wedge C \text{ with} \\
C_{Domains} &= Event, Carpet, Band, Stage, CBal \in \{False, True\} \wedge \\
&\quad CBal.PRed, Carpet.SRes \in \{False, True\} \wedge \\
&\quad Discount \in \{0, 3, 8\} \wedge Band.Type \in \{700, 900, 1000\} \wedge \\
&\quad Carpet.Length \in [-2^{31}, 2^{31} - 1] \wedge Carpet.Breadth \in [50, 300] \wedge \\
&\quad Carpet.Color \in [0, 3] \wedge Carpet.Cost = ... \wedge \\
&\quad CBal.Amount \in \{500, 1000, 2500, 5000, 10000\} \wedge \\
&\quad CBal.Cost \in [-2^{31}, 2^{31} - 1] \wedge \\
&\quad CBal.PRed.Coupon \in [-2^{31}, 2^{31} - 1] \wedge \\
&\quad OverallCost \in [-2^{31}, 2^{31} - 1] \text{ and}
\end{aligned}$$

$$\begin{aligned}
C = \; &(Carpet \vee Band \vee Stage \vee CBal \rightarrow Event) \wedge \\
&(CBal.PRed \rightarrow CBal) \wedge (Band \rightarrow Stage) \wedge \\
&((Band \wedge Band.Type = 700) \rightarrow Carpet.Color = 3) \wedge \\
&(CBal.PRed \rightarrow CBal.PRed.Coupon = 1000) \wedge \\
&(\neg\, CBal.PRed \rightarrow CBal.PRed.Coupon = 0) \wedge \\
&(CBal.Cost = CBal.Amount * 3 - CBal.PRed.Coupon) \wedge \ldots
\end{aligned}$$

4.2 Domain Analysis

In *FdFeatures* the specification of an attribute's base domain is optional. If no domain is given by the user, as e.g. for *Carpet.Length* or *ColoredBalloons.Cost* in Listing 4.1, it is set by default to the maximal possible domain of the corresponding attribute type. For example, for integer attributes the maximal domain is $[-2^{31}, 2^{31} - 1]$ which we denote by $MAXDOM$ in the following.

When the Choco solver computes the valid domains of the CSP in the second phase of our approach (cf. Sect. 4.3), this may become time consuming. The solver must establish global consistency. Thus, up to $4.3 * 10^9$ values must be checked for every attribute (or its corresponding variable, resp.) with $MAXDOM$. Of course, we cannot require the user to specify attribute domains just big enough to contain all solutions, in particular, because a manual estimation of the base domain can be very difficult for complex feature models. Thus, we apply an automatic pre-analysis to the feature model which is merged with the CSP generated from the model.

Our *domain analysis* aims at an approximate yet quick pre-calculation of the base domains of variables using knowledge about the feature model's structure. We only consider integer attributes, as enumerated attributes will in general have small domains. The analysis is based on interval arithmetics [16] which allow a fast approximation of the variable's minimum and maximum values by calculating with intervals instead of single domain values.

The domain DOM_{FM} of an attribute in an *FdFeatures* feature model can be specified directly by giving a single value or a set or interval, resp. of values. Additionally, it is possible to specify particular sub-domains depending on the configuration state, i.e. IN_{FM} and OUT_{FM} in case the attribute is selected or deleted, resp. Furthermore, arithmetic expressions can be used to specify the domain or sub-domains. We determine DOM_{FM}, IN_{FM}, and OUT_{FM} in form of intervals from the attribute expressions, where enumerations are handled as intervals, too.

Starting from these domains, we calculate the narrowed base domain $BASEDOM$, and new sub-domains IN and OUT as follows (where we take arithmetic expressions into consideration):

$$BASEDOM = (IN_{FM} \cup OUT_{FM}) \cap DOM_{FM} \tag{1}$$

$$IN = BASEDOM \cap IN_{FM} \tag{2}$$

$$OUT = BASEDOM \cap OUT_{FM} \tag{3}$$

The intervals for the incorporated arithmetic expressions are determined by traversing their formula tree. The leafs are either elementary expressions or references to other attributes, in case of which the domain of the referenced attribute must be calculated first. The analysis of cyclic formulae is interrupted and $MAXDOM$ is used instead, leaving domain narrowing to the Choco solver, which uses accurate but time consuming consistency techniques.

Example 6. Consider the pre-calculation of the base domain $BASEDOM$ of the attribute *ColoredBalloons.Cost* (Line 25 of Listing 4.1). Figure 2 illustrates the calculation.

For the attribute under consideration, only the set DOM_{FM} is specified by means of an arithmetic expression, while IN_{FM} and OUT_{FM} both default to $MAXDOM$. During the analysis, the formula tree of the arithmetic expression is traversed. Dashed arrows depict the domain analysis of a referenced attribute which is shown in its own box.

In the beginning the analysis moves to the first leaf: a reference to the attribute *Amount*. The determination of the base sets is trivial as only DOM_{FM}

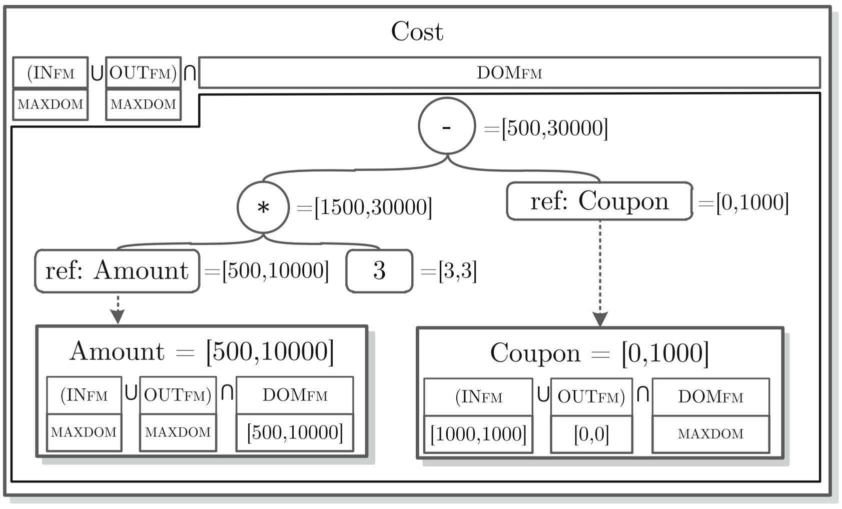


Fig. 2. Domain analysis of $BASEDOM$ of the attribute *ColoredBalloons.Cost*

is defined as an enumeration of integers which yields an interval $[500, 10000]$ using Eq. 1. The right operand of the multiplication is a constant value, which is turned into the point interval $[3, 3]$ resulting in the intermediate result $[500, 10000] * [3, 3] = [1500, 30000]$. The analysis of the attribute *Coupon* yields $BASEDOM = [0, 1000]$ from $IN_{FM} = [1000, 1000]$, $OUT_{FM} = [0, 0]$ and $DOM_{FM} = MAXDOM$ (again using Eq. 1). Finally, the analysis returns to the root attribute *ColoredBalloons.Cost* and performs the subtraction with result $BASEDOM = [1500, 30000] - [0, 1000] = [500, 30000]$.

From the $BASEDOM$ intervals of the attributes the respective sub-domains IN and OUT can be inferred by means of the Eqs. 2 and 3 (not shown in the figure).

Example 7. (continuation of Examples 5 and 6) The domain analysis yields the following domain constraints as an update on the generated CSP of our event feature model.

$$\begin{aligned} C'_{Domains} = \ldots \wedge \\ &CBal.Cost \in [500, 30000] \wedge \\ &CBal.PRed.Coupon \in \{0, 1000\} \wedge \ldots \end{aligned}$$

Note, that for computed intervals we finally build intersections in case the domain was initially given by enumerations or single values. This yields the two-element set for $CBal.PRed.Coupon$.

4.3 The Configuration Phase

The second phase of our approach, i.e. the configuration phase, starts with the initialization of *FdConfig* before the user can start with the interactive configuration process.

Model pre-processing. The solver reads the CSP-model and performs a feasibility check (e.g. by finding the first solution). If successful, the configurator computes the valid domains as initial *model consequences* that derive from the CSP. The calculation of these model consequences is performed in the same way as the user consequences are calculated later on in the interactive user configuration phase (see below). However, once the model consequences have been computed, they are immutable during the interactive configuration as they don't depend on the user decisions.

The current, global consistent state of the solver is recorded. To this *ground level state* the solver can be reset when, after a retraction of user constraints, a re-computation of the valid domains becomes necessary.

User configuration. The user starts a configuration step by executing a *configuration action.* This is either a configuration decision, i.e. limiting the domain of a feature- or attribute variable which manifests as a user constraint or the retraction of a decision made earlier. In this case the corresponding user constraint is removed from the constraint system. User decisions are posted by *FdConfig* to the solver as user constraints.

Now, the solver is activated to establish global consistency and to find all solutions of the constraint system. These are evaluated to derive the valid domains. Since the valid domains define the configuration options available to the user in the next configuration step, the constraint system always remains feasible after a user decision (cf. Sect. 2, backtrack-freeness).

After the user consequences have been computed, the user interface is updated accordingly and the user can perform the next configuration action.

In the usual modus operandi for FD solvers, a CSP is once declared and then read by the solver which computes and returns solutions. In contrast, for interactive configuration we need to re-calculate sets of solutions again and again because a sub-set of the constraints (the user constraints) keeps changing over time as a result of the user making configuration decisions.

As the solver maintains a heavyweight internal representation of the constraint system and reading the CSP-model as well as establishing consistency are time consuming, the option of re-creating the model for every user decision is inapplicable. Therefore we control the solver from outside by utilizing its backtracking infrastructure and reset the solver into the aforementioned ground level state in case a user decision has been retracted. This works for the general case, but is not optimal if the user wants to undo his very last decisions, because the solver starts over from the ground level state. Instead, we only let it backtrack to the state before the user constraints to be undone have been posted.

Optimization. In case the user wants to find an optimal configuration, i.e. configure an event with the lowest overall cost as in Example 3, he can simply select the respective minimum or maximum value available under the current partial configuration. If a more complex optimization is needed, he can choose a set of attributes that become part of a weighted sum, which is then used as an objective function. After the sum's coefficients have been entered, the function can be maximized or minimized under the current partial configuration. If more than one configuration is optimal, the user can select which one should be applied.

4.4 Improving the User Experience by Multithreading

When computing the valid domains of the variables, the constraint solver must establish global consistency, and thus, potentially find all solutions of the CSP. This calculation may be time consuming depending on the size and complexity of the feature model (and the CSP it was transformed into, resp.). Furthermore, the GUI would not be updated or process user input during this calculation. The program would appear to be frozen.

Therefore we introduced multithreading with the solver running in a background thread, allowing the GUI to be updated and accept user input during a long running computation. However, as the user would still have to wait for the calculation to complete before he can enter another configuration decision, the multithreading structure has been extended as follows:

The elements of the valid domains are collected gradually with the computation of the set of solutions still in progress. Whenever new elements have been found, they are immediately displayed in the GUI and made available for configuration decisions. Elements, that did not yet occur in a solution, are greyed out and disabled for user decisions. If the user makes a decision, the background calculation is interrupted and restarted with the changed set of user decisions.

In the sequential model the valid domains were calculated in one go and then evaluated to generate consequence decisions if necessary. If, for example, the valid domain of a feature A was found to be $D_{vd,A} = \{true\}$ this resulted in a consequence decision forcing the feature to be *selected*. [6]

With multithreading we have to re-evaluate the valid domains whenever new elements are found during the calculation process. This results in changing consequence decisions while the computation has not finished. For example, the valid domain of feature A can become $D_{vd,A} = \{true\}$ during the computation process at first, creating the consequence decision that A must be selected. However, as the result of new solutions the valid domain might later become $D_{vd,A} = \{true, false\}$, thus making the consequence decision disappear again. Attributes are handled similarly, as single value domains (interpreted as consequence decisions to select this particular value) may become multi-value domains later on. The GUI flags these consequence decisions as *incomplete*, so the user can see that further configuration options might become available. On the completion of the computation process, this flag is removed.

Figure 3 illustrates the different states for feature and attribute domains, resulting from the multithreading approach. Consequence decisions are drawn in bold typeface. Furthermore Fig. 4 shows a screenshot of the *FdConfig* tool during a long running calculation of the valid domains. Incomplete consequence decisions are visible, i.e. for the attribute *ColoredBalloons.Cost*, whose valid domain has exactly one element (14990) at the moment. The other elements were either eliminated by the user or have not yet occurred in a solution (displayed in grey).

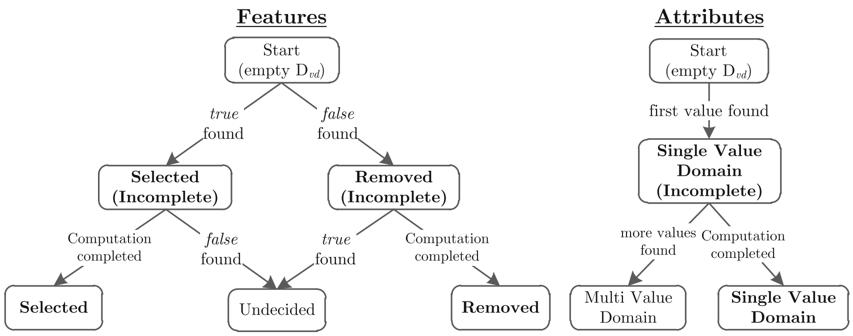
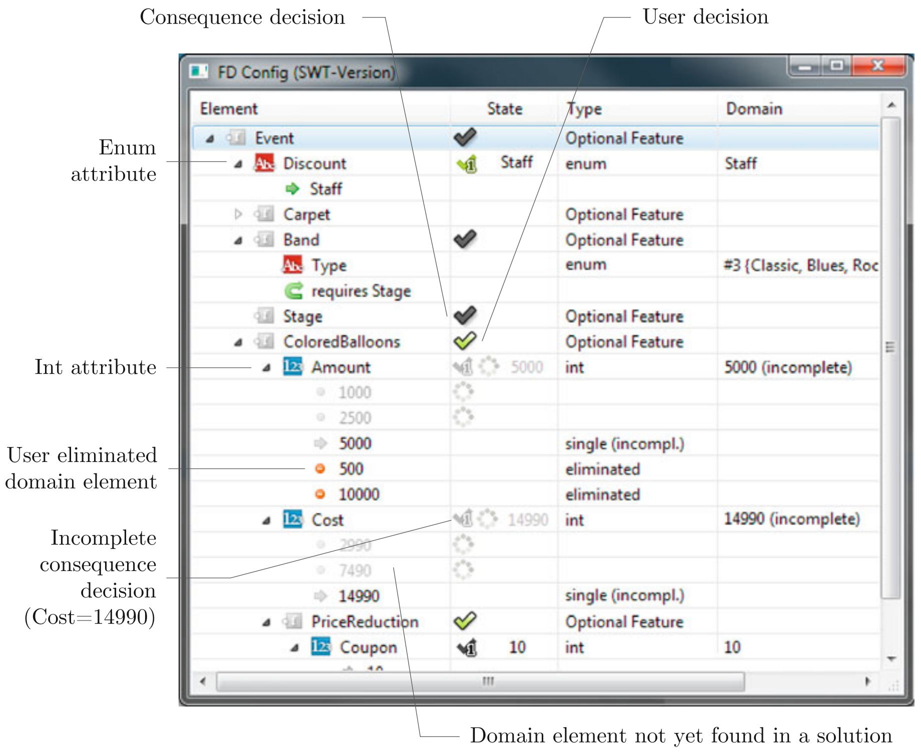


Fig. 3. Intermediate states of valid domains for features and attributes with multithreading

[6] Likewise $D_{vd,A} = \{false\}$ results in a *removed* feature and $D_{vd} = \{true, false\}$ in the *undecided* state, where the user can decide.


Consequence decision
User decision
Enum attribute
Int attribute
User eliminated domain element
Incomplete consequence decision (Cost=14990)
Domain element not yet found in a solution
FD Config (SWT-Version)

Element	State	Type	Domain
Event	✓	Optional Feature	
Discount	Staff	enum	Staff
Staff			
Carpet		Optional Feature	
Band	✓	Optional Feature	
Type		enum	#3 {Classic, Blues, Roc
requires Stage			
Stage	✓	Optional Feature	
ColoredBalloons	✓	Optional Feature	
Amount	5000	int	5000 (incomplete)
1000			
2500			
5000		single (incompl.)	
500		eliminated	
10000		eliminated	
Cost	14990	int	14990 (incomplete)
2990			
7490			
14990		single (incompl.)	
PriceReduction	✓	Optional Feature	
Coupon	10	int	10

Fig. 4. Screenshot of *FdConfig* during a CVD calculation

First experiments show that this multithreading approach leads to a smoother, more fluent user experience when performing product configuration. Since reaching the goal of calculating the valid domains in under 250 ms[7] is currently not realistic with the underlying solvers, this enhancement is a good compromise as configuration options will become available very quickly.

5 Conclusion and Future Work

In this paper we discuss an approach on *interactive* product configuration *with extended features like arithmetics on product attributes and support for optimizations* based on constraint techniques, which was implemented in our configurator tool *FdConfig*. We gave an overview of the product configuration domain, feature models, and constraint programming in this context and introduced our approach.

In *FdConfig* we employ a finite domain constraint solver that enables us to deal with integer attributes and arithmetic constraints in extended feature

[7] A response time of about this duration is considered desirable, as this still gives the user the impression to work in real time [7].

models. These constraints are usually not supported in traditional approaches (e.g. SAT, BDDs) or only in very restricted forms. However, this enhanced expressiveness comes at the cost of performance penalties. We deal with this in two ways: We apply a preliminary domain analysis in order to relief the solver from unnecessary computation time for establishing consistency. And, furthermore, we use a multithreading approach to enhance the user experience. This allows the user to continue configuring in a limited way, even if the overall computation has not yet finished.

Future work will include the further development of the multithreading approach. We plan to incorporate multiple solvers that might concentrate on particular parts of the computation. For example, the feature model element with the current GUI focus could be taken into account. This focus-based computation strategy could additionally improve user friendliness: Domain elements, that the user might want to configure most likely would become available more quickly for configuration decisions.

Also a more subtle handling of the non-chronological retraction of constraints promises improvement but needs further investigation.

In order to improve the overall performance we consider adding support for compilation-based approaches (i.e. BDDs). These could be integrated with the solver in the form of custom constraints to speed up the search. If a pre-compiled version of a feature model is available, the implementation of these constraints could access the BDD. Otherwise the regular solution methods would be applied.

Transformation-based optimizations should be investigated as well. E.g. [13] use a clustering optimization to reduce the number of constraint-variables and constraints. Using feature models may support or even inherently realize a form of clustering ([13], in contrast to ours, directly use constraints).

Another optimization is presented in [17]. The authors discuss the improvement of efficiency when solving CSPs as transformation results due to a reformulation of particular Boolean constraints into arithmetic constraints. While this representation is available in our approach too, the examination of similar optimizations may be worth considering in the future.

In the approach of [17] the structure of feature models is not preserved. This holds optimization potential as well, but must be done sensitively to retain a mapping to the feature model to allow an interactive configuration process as needed in our approach.

References

1. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: Pastor, O., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
2. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inf. Syst. **35**(6), 615–636 (2010)
3. ChocoSolver. http://www.emn.fr/z-info/choco-solver/. Accessed 3 Feb 2012
4. Classen, A., Boucher, Q., Faber, P., Heymans, P.: The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium (2010)

5. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, New York (2000)
6. Gecode - Generic Constraint Development Environment. http://www.gecode.org. Accessed 3 Feb 2012
7. Hadzic, T., Andersen, H.R.: An introduction to solving interactive configuration problems. Technical Report TR-2004-49, The IT University of Copenhagen (2004)
8. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: International Conference on Economic, Technical and Organizational Aspects of Product Configuration Systems (PETO), pp. 131–138 (2004)
9. Hofstedt, P.: Multiparadigm Constraint Programming Languages. Springer, Heidelberg (2011)
10. JaCoP - Java Constraint Programming solver. http://jacop.osolpro.com/. Accessed 3 Feb 2012
11. Janota, M., Botterweck, G., Grigore, R., Marques-Silva, J.: How to complete an interactive configuration process? In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM 2010. LNCS, vol. 5901, pp. 528–539. Springer, Heidelberg (2010)
12. Janota, M.: Do SAT solvers make good configurators?. In: First Workshop on Analyses of Software Product Lines (ASPL), September 2008
13. John, U., Geske, U.: Constraint-based configuration of large systems. In: Bartenstein, O., Geske, U., Hannebauer, M., Yoshie, O. (eds.) INAP 2001. LNCS (LNAI), vol. 2543, pp. 217–234. Springer, Heidelberg (2003)
14. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Spencer Peterson, A.: Feature-oriented domain analysis (foda). Feasibility study. Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222, SW Engineering Institute, Carnegie Mellon University (1990)
15. Karataş, A.S., Oğuztüzün, H., Doğru, A.: Mapping extended feature models to constraint logic programming over finite domains. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 286–299. Springer, Heidelberg (2010)
16. Kearfott, R.B.: Interval computations: introduction, uses, and resources. Euromath Bull. **2**, 95–112 (1996)
17. Mazo, R., Salinesi, C., Diaz, D., Lora-Michiels, A.: Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In: Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). Springer, Heidelberg (2011)
18. Mendonca, M., Branco, M., Cowan, D.: Splot: software product lines online tools. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, pp. 761–762. ACM, New York (2009)
19. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier, Amsterdam (2007)
20. Schneeweiss, D.: Grafische, interaktive Produktkonfiguration mit Finite-Domain-Constraints. Diploma thesis, Brandenburg University of Technology, Cottbus, September 2011. http://www.denny-schneeweiss.de/academic/publications/Schneeweiss2011--DiplomaThesis.pdf
21. Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, p. 305. Springer, Heidelberg (1999)

22. Soininen, T., Niemelä, I., Tiihonen, J., Sulonen, R.: Representing configuration knowledge with weight constraint rules. In: AAAI Symposium on Answer Set Programming, pp. 195–201. AAAI Press, March 2001
23. Subbarayan, S.: Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 351–365. Springer, Heidelberg (2005)
24. Subbarayan, S., Jensen, R.M., Hadzic, T., Andersen, H.R., Hulgaard, H., Møller, J.: Comparing two implementations of a complete and backtrack-free interactive configurator. In: Workshop on CSP Techniques with Immediate Application (CSPIA), pp. 97–111 (2004)
25. Trinidad, P., Benavides, D., Ruiz-Cortés, A., Segura, S., Jimenez, A.: Fama framework. In :12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, pp. 359–359. IEEE (2008)
26. Tseng, M.M., Jiao, J.: Mass customization. In: Salvendy, G. (ed.) Handbook of Industrial Engineering, Technology and Operation Management, 3rd edn. Wiley, New York (2001)
27. Xtext. Language development framework. http://www.eclipse.org/Xtext/. Accessed 3 Feb 2012
28. Yan, H., Zhang, W., Zhao, H., Mei, H.: An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. In: Edwards, S.H., Kulczycki, G. (eds.) ICSR 2009. LNCS, vol. 5791, pp. 65–75. Springer, Heidelberg (2009)

INAP System Descriptions

dynPARTIX - A Dynamic Programming Reasoner for Abstract Argumentation

Wolfgang Dvořák[1(✉)], Michael Morak[2], Clemens Nopp[2], and Stefan Woltran[2]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
wolfgang.dvorak@univie.ac.at
[2] Institute of Information Systems, Vienna University of Technology, Vienna, Austria

Abstract. The aim of this paper is to announce the release of a novel system for abstract argumentation which is based on decomposition and dynamic programming. We provide first experimental evaluations to show the feasibility of this approach.

1 Introduction

Argumentation has evolved as an important field in AI, with abstract argumentation frameworks (AFs, for short) as introduced by Dung [5] being its most popular formalization. Several semantics for AFs have been proposed (see e.g. [2] for an overview), but here we shall focus on the so-called preferred semantics. Reasoning under this semantics is known to be intractable [6]. An interesting approach to dealing with intractable problems comes from parameterized complexity theory which suggests to focus on parameters that allow for fast evaluations as long as these parameters are kept small. One important parameter for graphs (and thus for argumentation frameworks) is tree-width, which measures the "tree-likeness" of a graph. To be more specific, tree-width is defined via a certain decomposition of graphs, the so-called tree decomposition. Recent work [7] describes novel algorithms for reasoning in the preferred semantics, such that the performance mainly depends on the tree-width of the given AF, rather than on the size of the AF. To put this approach to practice, we shall use the *SHARP* framework,[1] a C++ environment which includes heuristic methods to obtain tree decompositions [4], provides an interface to run algorithms on these decompositions, and offers further useful features, for instance for parsing the input. For a description of the *SHARP* framework, see [9].

The main purpose of our work here is to support the theoretical results from [7] with experimental ones. Therefore we use different classes of AFs and analyze the performance of our approach compared to an implementation based on answer-set programming (see [8]). Our prototype system together with the

This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT08-028, by the Austrian Science Fund (FWF): P20704-N18, and by the Vienna University of Technology program "Innovative Ideas".

[1] http://www.dbai.tuwien.ac.at/research/project/sharp

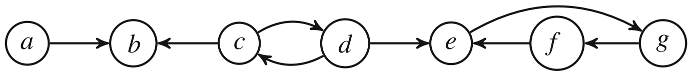
used benchmark instances is available as a ready-to-use tool from http://www.dbai.tuwien.ac.at/research/project/argumentation/dynpartix/.

2 Background

Argumentation Frameworks

An *argumentation framework* (*AF*) is a pair $F = (A, R)$ where A is a set of arguments and $R \subseteq A \times A$ is the attack relation. If $(a, b) \in R$ we say a attacks b. For $S \subseteq A$ and $a \in A$, we write $S \rightarrowtail a$ (resp. $a \rightarrowtail S$) iff there exists $b \in S$, such that $b \rightarrowtail a$ (resp. $a \rightarrowtail b$). An $a \in A$ is *defended* by a set $S \subseteq A$ iff for each $(b, a) \in R$, there exists a $c \in S$ such that $(c, b) \in R$. An AF can naturally be represented as a digraph.

Example 1. Consider the AF $F = (A, R)$, with $A = \{a, b, c, d, e, f, g\}$ and $R = \{(a, b), (c, b), (c, d), (d, c), (d, e), (e, g), (f, e), (g, f)\}$. The graph representation of F is given as follows:



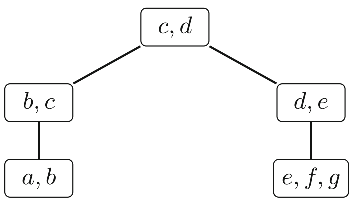
We require the following semantical concepts: Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is (i) *conflict-free* in F, if there are no $a, b \in S$, such that $(a, b) \in R$; (ii) *admissible* in F, if S is conflict-free in F and each $a \in S$ is defended by S; (iii) a *preferred extension* of F, if S is a $\subseteq$-maximal admissible set in F. in Example 1, we get the admissible sets $\{\}, \{a\}, \{c\}, \{d\}, \{d, g\}, \{a, c\}, \{a, d\}$, and $\{a, d, g\}$. Consequently, the preferred extensions of this framework are $\{a, c\}, \{a, d, g\}$.

The typical reasoning problems associated with AFs are the following: (1) Credulous acceptance asks whether a given argument is contained in at least one preferred extension of a given AF; (2) skeptical acceptance asks whether a given argument is contained in all preferred extensions of a given AF. Credulous acceptance is NP-complete, while skeptical acceptance is even harder, namely Π_2^P-complete [6].

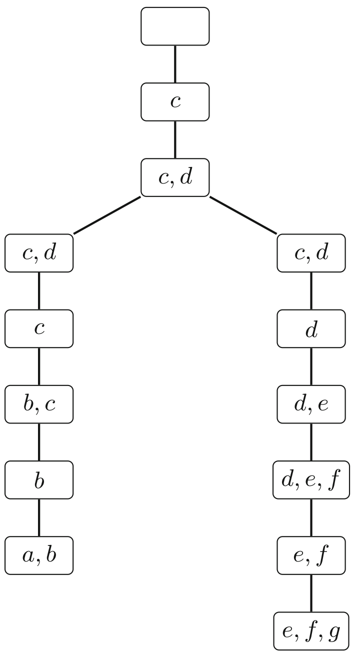
Tree Decompositions and Tree-Width

As already outlined, tree decompositions will underlie our implemented algorithms. We briefly recall this concept (which is easily adapted to AFs). A *tree decomposition* of an undirected graph $G = (V, E)$ is a pair $(\mathcal{T}, \mathcal{X})$ where $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is a tree and $\mathcal{X} = (X_t)_{t \in V_\mathcal{T}}$ is a set of so-called bags, which has to satisfy the following conditions:

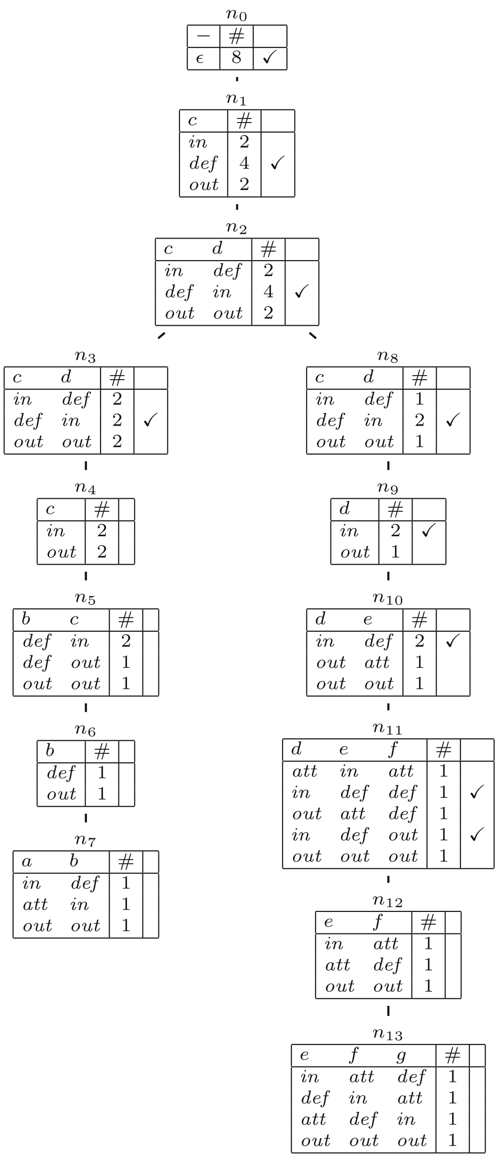
(a) $\bigcup_{t \in V_\mathcal{T}} X_t = V$, i.e. $\mathcal{X}$ is a cover of V;
(b) for each $v \in V$, the subgraph of $\mathcal{T}$ induced by $\{t \mid v \in X_t\}$ is connected;
(c) for each $\{v_i, v_j\} \in E$, $\{v_i, v_j\} \subseteq X_t$ for some $t \in V_\mathcal{T}$.



(a) Tree-decomposition of our running example.



(b) Normalized tree-decomposition of our running example.



n_0

−	#	
ϵ	8	✓

n_1

c	#	
in	2	
def	4	✓
out	2	

n_2

c	*d*	#	
in	*def*	2	
def	*in*	4	✓
out	*out*	2	

n_3

c	*d*	#	
in	*def*	2	
def	*in*	2	✓
out	*out*	2	

n_4

c	#	
in	2	
out	2	

n_5

b	*c*	#	
def	*in*	2	
def	*out*	1	
out	*out*	1	

n_6

b	#	
def	1	
out	1	

n_7

a	*b*	#	
in	*def*	1	
att	*in*	1	
out	*out*	1	

n_8

c	*d*	#	
in	*def*	1	
def	*in*	2	✓
out	*out*	1	

n_9

d	#	
in	2	✓
out	1	

n_{10}

d	*e*	#	
in	*def*	2	✓
out	*att*	1	
out	*out*	1	

n_{11}

d	*e*	*f*	#	
att	*in*	*att*	1	
in	*def*	*def*	1	✓
out	*att*	*def*	1	
in	*def*	*out*	1	✓
out	*out*	*out*	1	

n_{12}

e	*f*	#	
in	*att*	1	
att	*def*	1	
out	*out*	1	

n_{13}

e	*f*	*g*	#	
in	*att*	*def*	1	
def	*in*	*att*	1	
att	*def*	*in*	1	
out	*out*	*out*	1	

(c) Computation of vcolorings for our running example on the given nice tree-decomposition

Fig. 1. Illustration of the algorithm for admissible sets.

The width of a tree decomposition is given by $\max\{|X_t| \mid t \in V_\mathcal{T}\} - 1$. The *tree-width* of G is the minimum width over all tree decompositions of G.

It can be shown that our example AF has tree-width 2 and we illustrate a tree decomposition of width 2 in Fig. 1(a).

However for our purposes we shall use so-called normalized decompositions, that is the tree-decomposition has a root node r with $X_r = \emptyset$ and consists only of nodes of one of the following types. A node $t \in V_{\mathcal{T}}$ is a:

- Leaf-node if t has no children nodes in $\mathcal{T}$;
- Branch-node if t has two successors t', t'' in $\mathcal{T}$ and $X_t = X_{t'} = X_{t''}$
- Insert-node if t has only one successor t' and $X_t = X_{t'} \cup v$ for some $v \in V$;
- Removal-node if t has only one successor t' and $X_t = X_{t'} \setminus v$ for some $v \in V$.

A normalized version of the first tree-decomposition is presented in Fig. 1(b).

Dynamic programming algorithms traverse such tree decompositions and compute local solutions for each node in the decomposition. Thus the combinatorial explosion is now limited to the size of the bags, that is, to the width of the given tree decomposition.

3 Dynamic Programming Algorithm

In this section we sketch the dynamic programming algorithm for credulous acceptance. For more detailed explanations as well as for the dynamic programming algorithm for skeptical acceptance, the interested reader is referred to [7]. In the following we tacitly assume an AF $F = (A, R)$ and a corresponding normalized tree-decomposition $(\mathcal{T}, \mathcal{X})$.

Towards an algorithm for admissible sets we need the following concept: A set of arguments E is a *B-restricted admissible set for F*, if E is conflict-free in F and E defends itself against all $a \in A \cap B$. Clearly we have that the A-restricted admissible sets coincide with the admissible sets. Now the main idea behind the dynamic programming algorithm is to consider for each node $t \in \mathcal{T}$, the AF $F_{\geq t}$ induced by the union of the bags $X_{\geq t}$ of the sub-tree of $\mathcal{T}$ rooted at t and computing the $X_{\geq t} \backslash X_t$-restricted admissible sets. As for the root note r, $X_r = \emptyset$ and $X_{\geq t} = A$ we then have a handle on the admissible sets.

Next we consider how we represent $(X_{\geq t} \backslash X_t)$-restricted admissible sets in a node t. First let us mention that by the definition of tree-decompositions we have that the AF $F_{\geq t}$ already contains all attacks incident with arguments in $X_{\geq t} \backslash X_t$ and thus we do not need the status of these arguments for the computation in the ancestor nodes of t. Hence for each node t it suffices to store the status of the arguments X_t for each $X_{\geq t} \backslash X_t$-restricted admissible set E. This is implemented by so called *vcolorings* $C_t : X_t \mapsto \{in, def, att, out\}$ for t with the following intuition: $C_t(a) = in$ iff $a \in E$; $C_t(a) = def$ iff $E \rightarrowtail a$; $C_t(a) = att$ iff $E \not\rightarrowtail a$ and $a \rightarrowtail E$; $C_t(a) = out$ iff $E \not\rightarrowtail a$ and $a \not\rightarrowtail E$. We have that each $(X_{\geq t} \backslash X_t)$-restricted admissible set corresponds to exactly one vcoloring, but one vcoloring in general corresponds to several $X_{\geq t} \backslash X_t$-restricted admissible sets.

In the following we discuss how to compute the vcolorings for each node-type. Starting with leaf-nodes t we are interested in $\emptyset$-restricted admissible sets of $F_{\geq t}$, which coincide with the conflict-free sets. So we simply compute the conflict-free sets of $F_{\geq t}$ and map them to the corresponding vcolorings.

In a removal-node t with successor t' and $X_t = X_{t'} \setminus \{a\}$ we consider the successor's vcolorings $C_{t'}$ with $C_{t'}(a) \neq att$ and project them to X_t. We have that $C_{t'}(a) = att$ corresponds to a violation of admissibility, i.e. a attacks an argument in E and is not attacked by E, and as $a \in X_{\geq t} \setminus X_t$ the sets E corresponding to $C_{t'}$ are not $X_{\geq t} \setminus X_t$-restricted admissible.

Now let us focus on Insert node t with successor t' and $X_t = X_{t'} \cup \{a\}$. Again we consider vcolorings $C_{t'}$ of t'. Given a $X_{\geq t} \setminus X_t$ restricted admissible set E and adding a new argument a to the AF there are two ways to update E, either adding the new argument to E or not. This observation is mirrored by the following two operations. First we construct the vcoloring C_t^1 extending $C_{t'}$ to a such that a is labeled by one of the labels *def*, *att*, *out*, depending on the attacks between a and the arguments $\{a \in X_{t'} \mid C_{t'}(a) = in\} =: [C_{t'}]$. Moreover if $[C_{t'}] \cup \{a\}$ is conflict-free in F we also generate a vcoloring C_t^2 extending $C_{t'}$ such that $C_t^2(a) = in$ and faithfully update labels *att*, *out* according to attacks incident with a.

In a branch node t with successors t', t'' we union two sub-frameworks $F_{\geq t'}, F_{\geq t''}$ that intersect on X_t. Consequently to obtain an $(X_{\geq t} \setminus X_t)$-restricted admissible set of $F_{\geq t}$ we can combine each $(X_{\geq t'} \setminus X_{t'})$-restricted admissible set of $F_{\geq t'}$ with each $(X_{\geq t''} \setminus X_{t''})$-restricted admissible set of $F_{\geq t'}$ as long they coincide on X_t. Thus the vcolorings C of t are computed by combining vcolorings C' of t' and vcolorings C'' of t'' such that $[C'] = [C'']$. The coloring C computed from C',C'' is defined as follows. For $b \in X_t$ we have: $C(b) = in$ iff $C'(b) = C''(b) = in$; $C(b) = def$ iff $C'(b) = def$ or $C''(b) = def$; $C(b) = out$ iff $C'(b) = out$ and $C''(b) = out$; and $C(b) = att$ in the remaining cases.

Proposition 1. *For node t and $a \in X_t$. There is a vcoloring C_t for t with $C_t(a) = in$ iff a is contained in an $X_{\geq t} \setminus X_t$-restricted admissible sets of $F_{\geq t}$.*

Finally we discuss how credulous acceptance can be decided via vcolorings. We just mark each vcoloring which assigns the value *in* to the argument we are interested in and accordingly pass this mark up to the root. That is we mark a coloring if it is constructed by using at least one marked coloring. Finally at the (empty) root node we have that the argument is credulously accepted iff the vcoloring of the root is marked.

Example 2. Recall our running example, the computation of vcolorings is illustrated in Fig. 1(c). For deciding the credulous acceptance of argument d we mark vcolorings corresponding to at least one set containing d with a ✓, according to the above rules. The argument d is introduced two times, in the node n_3 and in the node n_{11}. Thus, we mark their vcolorings C satisfying $C(d) = in$. Now consider n_8 with the colorings $C_1(c) = in, C_1(d) = def$, $C_2(c) = def, C_2(d) = in$ and $C_3(c) = out, C_3(d) = out$. The child node n_9 has colorings $C_1'(d) = in$ and $C_2'(d) = out$, the first marked. As C_2 is constructed via C_1' it is also marked and as C_1 and C_3 are both constructed via C_2' they are not marked. ◊

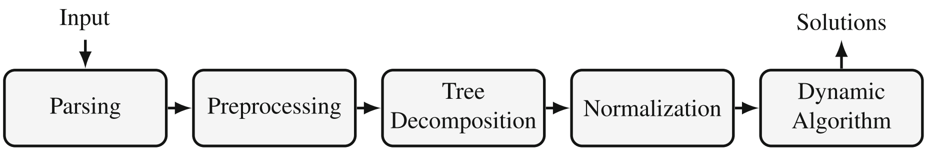


Fig. 2. Architecture of the *SHARP* framework.

4 Implementation and the *SHARP* Framework

dynPARTIX implements these dynamic programming algorithms based on tree decompositions using the *SHARP* framework [9], which is a purpose-built framework for implementing algorithms that are based on tree decompositions. Figure 2 shows the typical architecture, that systems working with the *SHARP* framework follow. In fact, *SHARP* provides interfaces and helper methods for the preprocessing and dynamic algorithm steps as well as ready-to-use implementations of various tree decomposition heuristics, i.e. Minimum-Fill, Maximum-Cardinality-Search and Minimum-Degree heuristics (cf. [4]), as well as different normalization algorithms.

As mentioned before, *dynPARTIX* builds on normalized tree decompositions provided by *SHARP*, which contain the four mentioned types of nodes. To implement our algorithms we just have to provide data structures storing the vcolorings of a node and the methods for each of these node types.

SHARP handles data-flow management and provides data structures where the calculated (partial) solutions to the problem under consideration can be stored. The amount of dedicated code for *dynPARTIX* comes to around 2700 lines in C++. Together with the *SHARP* framework (and the used libraries for the tree-decomposition heuristics), our system roughly comprises of 13000 lines of C++ code.

5 System Specifics

Currently the implementation is able to calculate the admissible and preferred extensions of a given argumentation framework and to check if credulous or skeptical acceptance holds for a specified argument. The basic usage of *dynPARTIX* is as follows:

```
> ./dynpartix [-f <file>] [-s <semantics>]
        [--enum | --count | --cred <arg> | --skept <arg>]
```

The argument `-f <file>` specifies the input file, the argument `-s <semantics>` selects the semantics to reason with, i.e. either admissible or preferred, and the remaining arguments choose one of the reasoning modes.

Input file conventions: We borrow the input format from the *ASPARTIX* system [8]. *dynPARTIX* thus handles text files where an argument a is encoded as `arg(a)` and an attack (a, b) is encoded as `att(a,b)`. For instance, consider the following encoding of our running example and let us assume that it is stored in a file `inputAF`.

```
arg(a). arg(b). arg(c). arg(d). arg(e). arg(f). arg(g).
att(a,b). att(c,b). att(c,d). att(d,c).
att(d,e). att(e,g). att(f,e). att(g,f).
```

Enumerating extensions: First of all, *dynPARTIX* can be used to compute extensions, i.e. admissible sets and preferred extensions. For instance to compute the admissible sets of our running example one can use the following command:

```
> ./dynpartix -f inputAF -s admissible
```

Credulous Reasoning: *dynPARTIX* decides credulous acceptance using proof procedures for admissible sets (even if one reasons with preferred semantics) to avoid unnecessary computational costs. The following statement decides if the argument d is credulously accepted in our running example.

```
 > ./dynpartix -f inputAF -s preferred --cred d
```

Indeed the answer would be *YES* as $\{a, d, g\}$ is a preferred extension.

Skeptical Reasoning: To decide skeptical acceptance, *dynPARTIX* uses proof procedures for preferred extensions which usually results in higher computational costs (but is unavoidable due to complexity results). To decide if the argument d is skeptically accepted, the following command is used:

```
 > ./dynpartix -f inputAF -s preferred --skept d
```

Here the answer would be *NO* as $\{a, c\}$ is a preferred extension not containing d.

Counting Extensions: Recently the problem of counting extensions has gained some interest [1]. We note that our algorithms allow counting without an explicit enumeration of all extensions (thanks to the particular nature of dynamic programming; see also [10]). Counting preferred extensions with *dynPARTIX* is done by

```
 > ./dynpartix -f inputAF -s preferred --count
```

6 Benchmark Tests

In this section we compare *dynPARTIX* with *ASPARTIX* [8], one of the most efficient reasoning tools for abstract argumentation (for an overview of existing argumentation systems see [8]). For our benchmarks we used randomly generated AFs of low tree-width. To ensure that AFs are of a certain tree-width we considered random grid-structured AFs. In such a grid-structured AF each argument is arranged in an $n \times m$ grid and attacks are only allowed between neighbours in the grid (we used an 8-neighborhood here to allow odd-length cycles, which are crucial for the full complexity of preferred semantics). When generating the benchmark instances we varied the following parameters: the number of arguments from 25 to 500; the tree-width; and the probability that a possible attack is actually in the AF.

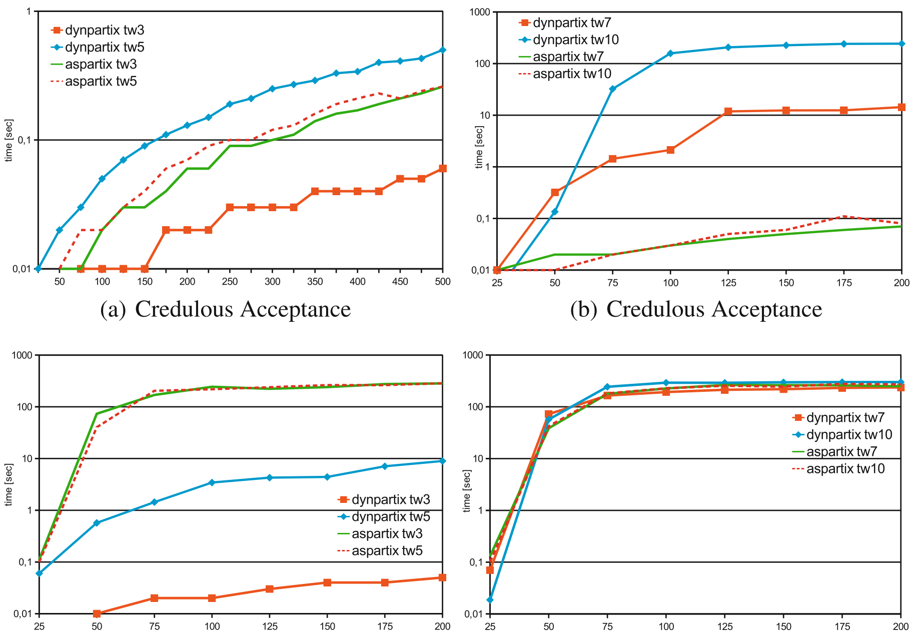
The benchmark tests were executed on an Intel®Core$^{\mathrm{TM}}$2 CPU 6300@1.86GHz machine running SUSE Linux version 2.6.27.48. We generated a total of 4800 argumentation frameworks with varying parameters as mentioned above. The two graphs on the left-hand side compare the running times of *dynPARTIX* and *ASPARTIX* (using dlv) on instances of small treewidth (viz. 3 and 5). For the graphs on the right-hand side, we have used instances of higher width. Results for credulous acceptance are given in the upper graphs and those for skeptical acceptance in the lower graphs. The y-axis gives the runtimes in logarithmic scale; the x-axis shows the number of arguments. Note that the upper-left picture has different ranges on the axes compared to the three other graphs. We remark that the test script stopped a calculation if it was not finished after 300 s. For these cases we stored the value of 300 s in the database.

Interpretation of the Benchmark Results: We observe that, independent of the reasoning mode, the runtime of *ASPARTIX* is only minorly affected by the tree-width while *dynPARTIX* strongly benefits from a low tree-width, as expected by theoretical results [7].

For the *credulous acceptance* problem we have that our current implementation is competitive only up to tree-width 5. Considering Fig. 3(a) and (b), there is to note that for credulous acceptance *ASPARTIX* decided every instance in less than 300 s, while *dynPARTIX* exceeded this value in 4 % of the cases.

Now let us consider the *skeptical acceptance* problem. As mentioned before, skeptical acceptance is computationally much harder than credulous acceptance, which is reflected by the bad runtime behaviour of *ASPARTIX*. Indeed we have that for tree-width ≤ 5, *dynPARTIX* has a significantly better runtime behaviour, and that it is competitive on the whole set of test instances. As an additional comment to Fig. 3(c) and (d), we note that for skeptical acceptance, *dynPARTIX* was able to decide about 71 % of the test cases within the time limit, while *ASPARTIX* only finished 41 %.

Finally let us briefly mention the problem of *Counting preferred extensions*. On the one side we have that *ASPARTIX* has no option for explicitly counting extensions, so the best thing one can do is enumerating extensions and then counting them. It can easily be seen that this can be quite inefficient, which is reflected by the fact that *ASPARTIX* only finished 21 % of the test instances in time. On the other hand we have that the dynamic algorithms for counting preferred extensions and deciding skeptical acceptance are essentially the same and thus have the same runtime behaviour.



(a) Credulous Acceptance (b) Credulous Acceptance

(c) Skeptical Acceptance (d) Skeptical Acceptance

Fig. 3. Runtime of *dynPARTIX* for graphs of different tree-width compared to *ASPARTIX*.

7 Discussion

In this paper, we have presented a novel system for abstract argumentation which is based on decomposition and dynamic programming. Experimental evaluations show that such an approach is able to outperform systems relying on answer-set programming, at least on certain instances. This indicates that despite the high sophistication answer-set programming systems have reached nowadays, structural features of the problem instance are not sufficiently recognized yet by these systems. As ongoing work we thus focus on a combination of the both paradigms, i.e. decomposition and making use of declarative programming languages, see http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/ for further details.

For future work, we need a more comprehensive empirical evaluation, in particular on real world instances. To this end, we need more knowledge about the tree-width typical argumentation instances comprise, i.e. whether it is the case that such instances have low tree-width. Due to the unavailability of benchmark libraries for argumentation, so far we had to omit such considerations. we plan to extend *dynPARTIX* by additional argumentation semantics mentioned in [2] and by further reasoning modes, which can be efficiently computed on tree decompositions. Finally, we plan to further develop *dynPARTIX* for non-normalized tree decompositions [3].

References

1. Baroni, P., Dunne, P.E., Giacomin, M.: On extension counting problems in argumentation frameworks. In: Proceedings of the COMMA 2010, pp. 63–74 (2010)
2. Baroni, P., Giacomin, M.: Semantics of abstract argument systems. In: Rahwan, I., Simari, G.R. (eds.) Argumentation in Artificial Intelligence, pp. 25–44. Springer, Heidelberg (2009)
3. Charwat, G.: Tree-decomposition based algorithms for abstract argumentation frameworks. Master's thesis, TU Wien (2012)
4. Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N., Samer, M.: Heuristic methods for hypertree decomposition. In: Gelbukh, A., Morales, E.F. (eds.) MICAI 2008. LNCS (LNAI), vol. 5317, pp. 1–11. Springer, Heidelberg (2008)
5. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. **77**(2), 321–358 (1995)
6. Dunne, P.E., Bench-Capon, T.J.M.: Coherence in finite argument systems. Artif. Intell. **141**(1/2), 187–203 (2002)
7. Dvořák, W., Pichler, R., Woltran, S.: Towards fixed-parameter tractable algorithms for abstract argumentation. Artif. Intell. **186**, 1–37 (2012)
8. Egly, U., Gaggl, S., Woltran, S.: Answer-set programming encodings for argumentation frameworks. Argument Comput. **1**(2), 147–177 (2010)
9. Morak, M.: SHARP - a smart hypertree-decomposition-based algorithm framework for parameterized problems. TU Wien. http://www.dbai.tuwien.ac.at/research/project/sharp/sharp.pdf (2010)
10. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms **8**(1), 50–64 (2010)

HEX-Programs with Nested Program Calls

Thomas Eiter, Thomas Krennwallner, and Christoph Redl(✉)

Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter, tkren, redl}@kr.tuwien.ac.at

Abstract. Answer-Set Programming (ASP) is an established declarative programming paradigm. However, classical ASP lacks subprogram calls as in procedural programming, and access to external computations (akin to remote procedure calls) in general. This feature is desired for increasing modularity and—assuming proper access in place—(meta-)reasoning over subprogram results. While HEX-programs extend classical ASP with external source access, they do not support calls of (sub-)programs upfront. We present *nested* HEX-*programs*, which extend HEX-programs to serve the desired feature in a user-friendly manner. Notably, the answer sets of called sub-programs can be individually accessed. This is particularly useful for applications that need to reason over answer sets like belief set merging, user-defined aggregate functions, or preferences of answer sets. We will further present a novel method for rapid prototyping of external sources by the use of nested programs.

1 Introduction

Answer-Set Programming, based on [8], has been established as an important declarative programming formalism [3]. However, a shortcoming of classical ASP is the lack of means for modularity, i.e., dividing programs into several interacting components. Even though reasoners such as DLV, CLASP, and DLVHEX allow to partition programs into several files, they are still viewed as a single monolithic set of rules.On top of that, passing input to selected (sub-)programs is not possible upfront.

In procedural programming, the idea of calling subprograms and processing their output is in permanent use. Also in functional programming such modularity is popular. This helps reducing development time (e.g., by using third-party libraries), the length of source code, and, last but not least, makes code human-readable. Reading, understanding, and debugging a typical size application written in a monolithic program is cumbersome. Modular extensions of ASP have been considered [5,9] with the aim of building an overall answer set from program modules; however, multiple results of subprograms (as typical for ASP) are respected, and no reasoning about such results is supported. XASP [11] is an

This research has been supported by the Austrian Science Fund (FWF) project P20840, P20841, and P24090, and by the Vienna Science and Technology Fund (WWTF) project ICT 08-020.

SMODELS interface for XSB-Prolog. This system is related to our work but less expressive, as it is designed for host programs under well-founded semantics. Moreover, our system allows the seamless integration of queries over subprograms with other external sources. Both is important for some applications, e.g., for the MELD belief set merging system[10], which require on the one hand choices, which is described in Sect. 4, and on the other hand access to arbitrary external sources in order to query the data sources to be merged. Adding such nesting to available approaches is not easy and requires to adapt systems similar to our approach.

HEX-programs [6] extend ASP with higher-order atoms, which allow the use of predicate variables, and external atoms, through which a bidirectional communication with external sources is enabled. But HEX-programs do not support modularity and meta-reasoning directly. In this context, modularity means the encapsulation of subprograms which interact through well-defined interfaces only, and meta-reasoning requires reasoning over *sets of* answer sets. Moreover, in HEX-programs external sources are realized as procedural C++ functions. Therefore, as soon as external sources are queried, we leave the declarative formalism. However, the generic notion of external atom, which facilitates a bidirectional data flow between the logic program and an external source (viewed as abstract Boolean function), can be utilized to provide these features.

To this end, we present *nested* HEX*-programs*, which support (possibly parameterized) *subprogram calls*. It is the nature of nested HEX-programs to have multiple programs which reason over the answer sets of each individual subprogram. This can be done in a user-friendly way and enables the user to write purely *declarative* applications consisting of multiple interacting modules. Notably, call results and answer sets are *objects* that can be accessed by identifiers and processed in the calling program. Thus, different from [5,9] and related formalisms, this enables *(meta)-reasoning about the set of answer sets* of a program. In contrast to [11], both the calling and the called program are in the same formalism. In particular, the calling program has also a declarative semantics. As an important difference to [1], nested HEX-programs do not require extending the syntax and semantics of the underlying formalism, which is the HEX-semantics. The integration is, instead, by defining some external atoms (which is already possible in ordinary HEX-programs), making the approach simple and user-friendly for many applications. Furthermore, as nested HEX-programs are based on HEX-programs, they additionally provide access to external sources other than logic programs. This makes nested HEX-programs a powerful formalism, which has been implemented using the DLVHEX reasoner for HEX-programs; applications like belief set merging [10] show its potential and usefulness. Moreover, we will show how nested programs can be used for external source simulation. This allows for rapid prototyping without actually implementing plugins for the reasoner, which is time-consuming.

2 HEX-Programs

We briefly recall HEX-programs, which have been introduced in [6] as a generalization of (disjunctive) extended logic programs under the answer set semantics [8]; for more details and background, we refer to [6]. A HEX-program consists of rules of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ , \ (m, n \geq 0)$$

where each a_i is a literal, i.e., an atom $p(t_1, \ldots, t_\ell)$ or a negated atom $\neg p(t_1, \ldots, t_\ell)$, and each b_j is either a classical literal or an external atom, and not is negation by failure (under stable semantics). An *external atom* is of the form

$$\&g[q_1, \ldots, q_k](t_1, \ldots, t_\ell) \ ,$$

where g is an external predicate name, the q_i are predicate names or constants, and the t_j are terms. Informally, the semantics of an external g is given by a $k+\ell+1$-ary Boolean *oracle function* $f_{\&g}$. The external atom is true relative to an interpretation I and a grounding substitution θ iff $f_{\&g}(I, q_1, \ldots, q_k, t_1\theta, \ldots, t_\ell\theta) = 1$. External atoms allow for including arbitrary (computable) functions. E.g., built-in functions can be realized via external atoms, or library functions such as string manipulations, sorting routines, etc. As external sources need not be on the same machine, knowledge access across the Web is possible, e.g., belief set import. Strictly, [6] omits classical negation $\neg$ but the extension is routine; furthermore, [6] also allows terms for the q_i and variables for predicate names, which we do not consider.

Example 1. Suppose an external knowledge base consists of an RDF file located on the web at http://.../data.rdf. Using an external atom $\&rdf[\texttt{url}](\texttt{X}, \texttt{Y}, \texttt{Z})$, we may access all RDF triples (s, p, o) at the URL specified with `url`. To form belief sets of pairs that drop the third argument from RDF triples, we may use the rule

$$bel(X, Y) \leftarrow \&rdf[\text{http://.../data.rdf}](X, Y, Z) \ .$$

The above program has a single answer set which consists of all literal $bel(c_1, c_2)$ such some RDF triple (c_1, c_2, c_3) occurs at the respective URL.

We use the DLVHEX system from http://www.kr.tuwien.ac.at/research/systems/dlvhex/ as a backend. DLVHEX implements (a fragment of) HEX-programs. It provides a plugin mechanism for external atoms. Besides library atoms, the user can define her own atoms using C++ methods.

3 Nested HEX-Programs

Limitations of ASP. As a simple example demonstrating the limits of ordinary ASP, assume a program computing the shortest paths between two (fixed) nodes in a connected graph. The answer sets of this program then correspond to the shortest paths. Suppose we are just interested in the *number* of such paths. In

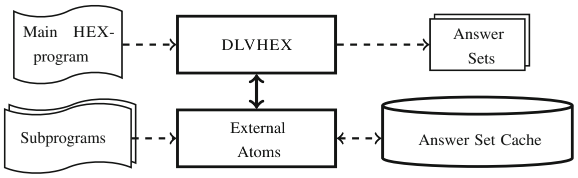


Fig. 1. System Architecture of Nested HEX (data flow $\dashrightarrow$, control flow $\rightarrow$)

a procedural setting, this is easily computed if the function returns all paths in an suitable data structure (e.g., an array or a linked list).

In ASP, the solution is non-trivial if the given program must not be modified (e.g., if it is provided by a third party); above, we must count the answer sets. Thus, we need to reason on *sets of* answer sets, which is infeasible inside the program. Means to call the program at hand and reason about the results of this *"callee"* (*subprogram*) in the *"calling program"* (*host program*) would be useful. Aiming at a logical counterpart to procedural function calls, we define a framework which allows to input facts to the subprogram prior to its execution. Host and subprograms are decoupled and interact merely by relational input and output values. To realize this mechanism, we exploit external atoms, leading to nested HEX-programs.

Architecture. Nested HEX-programs are realized as a plugin for the reasoner DLVHEX,[1] which consists of a *set of external atoms* and an *answer cache* for the results of subprograms (see Fig. 1). Technically, the implementation is part of the belief set merging system MELD, which is an application on top of a nested HEX-programs core. This core can be used independently from the rest of the system.

When a subprogram call (corresponding to the evaluation of a special external atom) is encountered of the host program, the plugin creates another instance of the reasoner to evaluate the subprogram. Its result is then stored in the answer cache and identified with a unique *handle*, which can later be used to reference the result and access its components (e.g., predicate names, literals, arguments) via other special external atoms. For economic memory management, the implementation may remove answer cache entries dynamically in the style of a *least frequently used* heuristics, and reevaluate the corresponding program again if it is later accessed again.

There are two possible sources for the called subprogram: (1) either it is *directly embedded* in the host program, or (2) it is *stored in a separate file*. In (1), the rules of the subprogram must be represented within the host program. To this end, they are encoded as string constants. An embedded program must not be confused with a subset of the rules of the host program. Even though it is syntactically part of it, it is logically separated to allow independent evaluation. In (2) merely the *path* to the location of the external program in the file system

[1] http://www.kr.tuwien.ac.at/research/systems/dlvhex/meld.html

is given. Compared to embedded subprograms, code can be reused without the need to copy, which is clearly advantageous when the subprogram changes. We now present concrete external atoms $\&callhex_n, \&callhexfile_n, \&answersets$, $\&predicates$, and $\&arguments$ which are used to realize nested HEX-programs.

External Atoms for Subprogram Handling. We start with two families of external atoms

$$\&callhex_n[\mathtt{P}, p_1, \ldots, p_n](H) \quad \text{and} \quad \&callhexfile_n[\mathtt{FN}, p_1, \ldots, p_n](H)$$

that allow to execute a subprogram given by a string P respectively in a file FN; here n is an integer specifying the number of predicate names $p_i, 1 \leq i \leq n$, used to define the input facts. When evaluating such an external atom relative to an interpretation I, the system adds all facts $\{p_i(a_1, \ldots, a_{m_i}) \leftarrow \mid p_i(a_1, \ldots, a_{m_i}) \in I\}$ to the specified program, creates another instance of the reasoner to evaluate it, and returns a symbolic handle H as result. For convenience, we do not write n in $\&callhex_n$ and $\&callhexfile_n$ as it is understood from the usage.

Example 2. In the following program, we use two predicates p_1 and p_2 to define the input to the subprogram sub.hex ($n = 2$), i.e., all atoms over these predicates are added to the subprogram prior to evaluation. The call derives a handle H as result.

$$p_1(x, y) \leftarrow \qquad p_2(a) \leftarrow \qquad p_2(b) \leftarrow$$
$$handle(H) \leftarrow \&callhexfile[\mathtt{sub.hex}, p_1, p_2](H)$$

A *handle* is a unique integer representing a certain program answer cache entry. In the implementation, handles are consecutive numbers starting with 0. Hence in the example the unique answer set of the program is $\{handle(0)\}$ (neglecting facts).

Formally, given an interpretation I, $f_{\&callhexfile_n}(I, file, p_1, \ldots, p_n, h) = 1$ iff h is the handle to the result of the program in file *file*, extended by the facts over predicates $p_1, \ldots, p_n$ that are true in I. The formal notion and use of $\&callhex_n$ to call embedded subprograms is analogous to $\&callhexfile_n$.

Example 3. Consider the following program:

$$h_1(H) \leftarrow \&callhexfile[\mathtt{sub.hex}](H)$$
$$h_2(H) \leftarrow \&callhexfile[\mathtt{sub.hex}](H)$$
$$h_3(H) \leftarrow \&callhex[\mathtt{a} \leftarrow .\ \mathtt{b} \leftarrow .](H)$$

The rules execute the program sub.hex and the embedded program $P_e = \{a \leftarrow, b \leftarrow\}$. No facts will be added in this example. The single answer set is $\{h_1(0), h_2(0), h_3(1)\}$ resp. $\{h_1(1), h_2(1), h_3(0)\}$ depending on the order in which the subprograms are executed (which is irrelevant). While $h_1(X)$ and $h_2(X)$ will have the same value for $X, h_3(Y)$ will be such that $Y \neq X$. Our implementation realizes that the result of the program in sub.hex is referred to twice but executes it only once; P_e is (possibly) different from sub.hex and thus evaluated separately.

Now we want to determine how many (and subsequently which) answer sets it has. For this purpose, we define external atom $\&answersets[PH](AH)$ which maps handles PH to call results to sets of respective answer set handles. Formally, for an interpretation I, $f_{\&answersets}(I, h_P, h_A) = 1$ iff h_A is a handle to an answer set of the program with program handle h_P.

Example 4. The single rule

$$ash(PH, AH) \leftarrow \&callhex[\mathtt{a} \vee \mathtt{b} \leftarrow .](PH), \&answersets[PH](AH)$$

calls the embedded subprogram $P_e = \{a \vee b \leftarrow .\}$ and retrieves pairs (PH, PA) of handles to its answer sets. $\&callhex$ returns a handle $PH = 0$ to the result of P_e, which is passed to $\&answersets$. This atom returns a *set* of answer set handles (0 and 1, as P_e has two answer sets, viz. $\{a\}$ and $\{b\}$). The overall program has thus the single answer set $\{ash(0,0), ash(0,1)\}$. As for each program the answer set handles start with 0, only a pair of program and answer set handles uniquely identifies an answer set.

We now are ready to solve our example of counting shortest paths from above.

Example 5. Suppose `paths.hex` is the search program and encodes each shortest path in a separate answer set. Consider the following program:

$$\begin{aligned} as(AH) &\leftarrow \&callhexfile[\mathtt{paths.hex}](PH), \&answersets[PH](AH) \\ number(D) &\leftarrow as(C), D = C + 1, \mathrm{not}\, as(D) \end{aligned}$$

The second rule computes the first free handle D; the latter coincides with the number of answer sets of `paths.hex` (assuming that some path between the nodes exists).

At this point we still treat answer sets of subprograms as black boxes. We now define an external atom to investigate them.

Given an interpretation I, $f_{\&predicates}(I, h_P, h_A, p, a) = 1$ iff p occurs as an a-ary predicate in the answer set identified by h_P and h_A. Intuitively, the external atom maps pairs of program and answer set handles to the predicates names with their associated arities occurring in the accourding answer set.

Example 6. We illustrate the usage of $\&predicates$ with the following program:

$$\begin{aligned} preds(P, A) \leftarrow\; &\&callhex[\mathtt{node(a).\ node(b).\ edge(a,b).}](PH), \\ &\&answersets[PH](AH), \&predicates[PH, AH](P, A) \end{aligned}$$

It extracts all predicates (and their arities) occurring in the answer of the embedded program P_e, which specifies a graph. The answer set is $\{preds(node, 1), preds(edge, 2)\}$ as the answer set of P_e has atoms with predicate $node$ (unary) and $edge$ (binary).

The final step to gather all information from the answer of a subprogram is to extract the *literals* and their *parameters* occurring in a certain answer set. This can be done with external atom $\&arguments$, which is best demonstrated with an example.

Example 7. Consider the following program:

$$\begin{aligned}
h(PH, AH) &\leftarrow \&callhex[\texttt{node(a). node(b). node(c). edge(a,b).edge(c,a).}](PH),\\
&\quad\ \&answersets[PH](AH)\\
edge(W, V) &\leftarrow h(PH, AH), \&arguments[PH, AH, \texttt{edge}](I, 0, V),\\
&\quad\ \&arguments[PH, AH, \texttt{edge}](I, 1, W)\\
node(V) &\leftarrow h(PH, AH), \&arguments[PH, AH, \texttt{node}](I, 0, V)
\end{aligned}$$

It extracts the directed graph given by the embedded subprogram P_e and reverses all edges; the answer set is $\{h(0,0), node(a), node(b), node(c), edge(b,a), edge(a,c)\}$. Indeed, P_e has a single answer set, identified by $PH = 0, AH = 0$; via *&arguments* we can access in the second resp. third rule the facts over *edge* resp. *node* in it, which are identified by a unique literal id I; the second output term of *&arguments* is the argument position, and the third the actual value at this position. If the predicates of a subprogram were unknown, we can determine them using *&predicates*.

To check the sign of a literal, the external atom $\&arguments[PH, AH, P]$ (I, s, S) supports argument s. When $s = 0$, *&arguments* will match the sign of the I-th *positive* literal over predicate P into S, and when $s = 1$ it will match the corresponding classically negated atom.

External Atoms for External Source Prototyping. Our system provides another family of external atoms for rapid prototyping of (simple) external sources directly in ASP. This it, the input-output behavior of hypothetical external sources is encoded by ASP rules. This is useful for quick experiments before a new external source is actually implemented. It comes with less implementation overhead compared to a native implementation in C++. This gives the user the possibility to see how the planned external atom will behave in a program even before it is developed. However, it is clear the possibility of simulating external sources cannot replace the plugin mechanism of DLVHEX as it cannot access real external sources. Moreover, simulation is less efficient than a native implementation in C++.

For simulation our system supports the external atom:

$$\&simulator_{n,m}[\texttt{F}, p_1, \ldots, p_n](X_1, \ldots X_m)$$

The simulator atom takes as arguments a filename F, which refers to the ASP program defining the input-output behavior of the prototypical external source, and predicate inputs $p_1, \ldots, p_n$. The output list $X_1, \ldots, X_m$ is used to retrieve the tuples from produced by the simulated external source.

When a simulator atom is encountered in the host program, it will evaluate the ASP-program in F extended by the input parameters defined over $p_1, \ldots, p_n$. In particular, the system will add for each input atom $p_i(a_1, \ldots, a_k)$ a fact of form $in_i(a_1, \ldots, a_k)$ to F. The renaming of the predicates is necessary in order to make F independent of the input predicate names in the host program. The result of F is expected to consist of exactly one answer set, where all atoms of form $out(o_1, \ldots, o_m)$ define the output of the simulated external source.

Example 8. Consider the following program P given by the rules:

$$\begin{aligned} dom(a) &\leftarrow \qquad dom(b) \leftarrow \qquad dom(c) \leftarrow \\ sel(X) &\leftarrow dom(X), \&simulator_{2,1}[\mathtt{Q}, dom, nsel](X) \\ nsel(X) &\leftarrow dom(X), \&simulator_{2,1}[\mathtt{Q}, dom, sel](X) \end{aligned}$$

Let further Q refer to the program:

$$out(X) \leftarrow in_1(X), \mathrm{not}\, in_2(X).$$

Then Q simulates an external source which computes the set difference, where the extension of the second predicate input in_2 is subtracted from the extension of the first predicate input in_1. The program P computes then the two sets *sel* and *nsel*, corresponding to all partitionings of $\{a, b, c\}$ into two subsets.

4 Applications

MELD. The MELD system [10] deals with merging multiple *collections of belief sets*. Roughly, a belief set is a set of classical ground literals. Practical examples of belief sets include explanations in abduction problems, encodings of decision diagrams, and relational data. The merging strategy is defined by tree-shaped *merging plans*, whose leaves are the collections of belief sets to be merged, and whose inner nodes are *merging operators* (provided by the user). The structure is akin to syntax trees of terms.

The automatic evaluation of tree-shaped merging plans is based on nested HEX-programs; it proceeds bottom-up, where every step requires inspection of the subresults, i.e., accessing the answer sets of subprograms. The meta program at the root node generates then one answer set for each integrated belief set. For this purpose, guessing rules select an integrated belief set of the top-level merging operator. The meta program then inherits the conclusions of the chosen belief set in order to make it visible to the user. Note that XASP [11] is thus not appropriate for such unstratified host programs, as it can only compute the well-founded semantics.

Aggregate Functions. Nested programs can also emulate aggregate functions [7] (e.g., `#sum`, `#count`, `#max`) where the (user-defined) host program computes the function given the result of a subprogram. This can be generalized to aggregates over *multiple* answer sets of the subprogram; e.g., to answer set counting, or to find the minimum/maximum of some predicate over all answer sets (which may be exploited for global optimization).

Generalized Quantifiers. Nested HEX-programs make the implementation of brave and cautious reasoning for query answering tasks very easy, even if the backend reasoner only supports answer set enumeration. Furthermore, extended and user-defined types of query answers (cf. [5]) are definable in a very user-friendly way, e.g., majority decisions (at least half of the answer sets support a query), or minimum and/or maximum number based decisions (qualified number restrictions).

Preferences. Answer sets as accessible objects can be easily compared wrt. user-defined preference rules, and used for filtering as well as ranking results (cf. [4]): a host program selects appropriate candidates produced by a subprogram, using preference rules. The latter can be elegantly implemented as ordinary integrity constraints (for filtering), or as rules (possibly involving further external calls) to derive a rank. A popular application are online shops, where the past consumer behavior is frequently used to filter or sort search results. Doing the search via an ASP program which delivers the matches in answer sets, a host program can reason about them and act as a filter or ranking algorithm.

Nested Programs as a Development Tool for DLVHEX. The further development of our system DLVHEX uses the idea of annotated external sources. This is, known properties like monotonicity and functionality shall be exploited for speeding up the reasoning process. Developing appropriate algorithms and heuristics requires empirical experiments with a variety of external sources. As it would be cumbersome to implement all of them as real plugins to DLVHEX, simulating them via our $\&simulator_{n,m}$ atom seems to be a good alternative.

5 Conclusion

To overcome limitations of classical ASP regarding subprograms and reasoning about their possible outcomes, we briefly presented *nested* HEX*-programs*, which realize subprogram calls via special external atoms of HEX-programs; besides modularity, a plus for readability and program reusability, they allow for reasoning over *multiple* answer sets (of subprograms). Moreover, nested HEX-programs can also be used as a tool for rapid external source prototyping. An implementation on top of DLVHEX is available. Related to this is the work on macros in [2], which allow to call macros in logic programs.

The possibility to access answer sets in a host program, in combination with access to other external computations, makes nested HEX-programs a powerful tool for a number of applications. In particular, libraries and user-defined functions can be incorporated into programs easily. As an interesting aspect is that dynamic program assembly (using a suitable string library) and execution are possible, which other approaches to modular ASP programming do not offer. Exploring this remains for future work.

References

1. Analyti, A., Antoniou, G., Damásio, C.V.: Mweb: a principled framework for modular web rule bases and its semantics. ACM Trans. Comput. Log. **12**(2), 17:1–17:46 (2011)
2. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)

4. Delgrande, J.P., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Comput. Intell. **20**(2), 308–334 (2004)
5. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Dix, J., Furbach, U., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 289–308. Springer, Heidelberg (1997)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer set programming. In: IJCAI'05, pp. 90–96. Professional Book Center (2005)
7. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. **175**(1), 278–298 (2011)
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Generat. Comput. **9**, 365–385 (1991)
9. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J. Artif. Intell. Res. **35**, 813–857 (2009)
10. Redl, C., Eiter, T., Krennwallner, T.: Declarative belief set merging using merging plans. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 99–114. Springer, Heidelberg (2011)
11. Swift, T., Warren, D.S.: XSB: Extending prolog with tabled logic programming. Theor. Pract. Log. Program **12**(1–2), 157–187 (2012)

A Prototype of a Knowledge-Based Programming Environment

Stef De Pooter(✉), Johan Wittocx, and Marc Denecker

Department of Computer Science, K.U. Leuven, Leuven, Belgium
{stef.depooter, johan.wittocx, marc.denecker}@cs.kuleuven.be

Abstract. This paper presents a proposal for a knowledge-based programming environment. Within this environment, declarative background knowledge, procedures, and concrete data are represented in suitable languages and combined in a flexible manner, which leads to a highly declarative programming style. We illustrate our approach with an example application and report on our prototype implementation.

1 Context

An obvious requirement for a powerful and flexible programming paradigm seems to be that within the paradigm different types of information can be expressed in suitable languages. However, most traditional programming paradigms and languages do not really have this property. In imperative languages, for example, non-executable background knowledge cannot be described. The consequences become clear when we try to solve a scheduling problem in an imperative language: the background knowledge – the constraints that need to be satisfied by the schedule – gets mixed up with the algorithms. This makes adding new constraints and finding and modifying existing ones cumbersome.

On the other hand, most logic-based declarative programming paradigms lack the capability to express procedures. Typically, they consist of a logic together with one specific type of inference. For example, Prolog uses Horn clause logic and does querying, in Description Logic the studied task is deduction, and Answer Set Programming and Constraint Programming make use of model generation. In such paradigms, whenever we try to perform a task that does not fit the inference mechanism at hand, the declarative aspect of the paradigm is lost. For example, when we try to solve a scheduling problem (which is a typical model-generation problem) in Prolog, we need to represent the schedule as a term, say a list (rather than as a logical structure), and as a result the constraints do not really reside in the logic program, but will have to be expressed by clauses that iterate over a list [5]. Proving that a certain requirement is implied by another, is possible (in theory) for a theorem prover, but not in ASP. Etc.

To overcome these restrictions of existing paradigms, we propose a paradigm in which each component can be expressed in an appropriate language. We distinguish three components: procedures, (non-executable) background knowledge, and concrete data. For the first we need an imperative language, for the

DOI: 10.1007/978-3-642-41524-1_16, © Springer-Verlag Berlin Heidelberg 2013

second an (expressive) logic, for the third a logical structure (which corresponds to a database). The connection between these components is realized by various reasoning tasks, such as theorem proving, model generation, model checking, model revision, belief revision, constraint propagation, querying, datamining, visualization, etc.

The idea to support multiple forms of inference for the same logic or even for the same theories, was argued in [7]. There it is argued that logic has a more flexible, multifunctional and therefore also more declarative role for problem solving than provided by many declarative programming paradigms, where typically one form of inference is central and theories are written to be used for this form of inference, sometimes even for a specific algorithm implementing this form of inference (such as Prolog resolution). The framework presented here is based on this view and goes beyond it in the sense that it offers a programming environment in which complex tasks can be programmed using multiple forms of inference and processing tools.

2 Overview of the Language and System

To try out the above mentioned ideas in practice, we built a prototype interpreter that supports some basic reasoning tasks and a set of processing tools on high-level data such as vocabularies, structures and theories. In this section we will highlight various decisions in the design of our programming language and interpreter. In the next section we will illustrate the usage of the language with an example. In the remainder of this text we will call our language DECLIMP, which is an aggregation of "declarative" and "imperative".

2.1 Program Structure

A DECLIMP program typically contains several blocks of code. Each block is either a procedure, a vocabulary (which is a list of sort, predicate and function names), a logic theory over a vocabulary (which describes a piece of background knowledge using the predicate and function names of its vocabulary), or a (possibly partial) structure over a vocabulary (which represents a database over its vocabulary). To bring more structure into a program and to be able to work with multiple files, namespaces and include statements are provided.

Because vocabularies, logic theories and databases are not executable, and a program needs to be executed, control of a DECLIMP program is always in the hands of the procedures. Moreover, when a `main` procedure is available, the run of the program will start with the execution of this procedure. When there is no `main` procedure, the user can run commands in an interactive shell, after parsing the program.

In the next sections, we will describe the languages for the respective components in a DECLIMP program.

2.2 Knowledge Representation Language

For representing background knowledge we use an extended version of classical logic. A first advantage of using classical logic as the basis of our knowledge representation language lies in the fact that it is the best-known and most-studied logic. Also, classical logic has the important property that its informal semantics corresponds to its formal semantics. In other words, in classical logic the meaning of expressions[1] is intuitively clear. This is an important requirement in the design of a language that is accessible to a wider audience. Furthermore, there are already numerous declarative systems that use a language based on classical logic, or can easily be translated to it. Think of the languages of most theorem provers [8], various Description logics [2], and the language of model generators such as IDP [10,22] and ENFRAGMO [16].

On the other hand, research in the Knowledge Representation and Reasoning community has clearly shown that pure classical logic is in many ways insufficient. Aggregates and (recursive) definitions are well-known concepts that are common in the background knowledge of many applications, and which can generally not, or not in a concise and intuitively clear manner, be expressed in first-order logic. Therefore, in DECLIMP we use an order-sorted version of first-order logic, extended with inductive definitions [6], aggregates [17], (partial) functions and arithmetic. These extensions make representing knowledge much easier.

2.3 Structures

Structures in DECLIMP are written in a simple language that allows to enumerate all elements that belong to a sort and all tuples that belong to a relation or function. As an alternative to enumerating a relation, it is also possible to specify the relation in a procedural way, namely as all the tuples for which a given procedure returns 'true'. Furthermore, the interpretation of a function can be specified by a procedure, somewhat similar to "external procedures" in `DLV` [3].

As mentioned before, structures in DECLIMP do not necessarily contain complete information, they are not necessarily two-valued. Three-valued structures are useful for representing incomplete information (which might be completed during the run of the program). To enumerate a three-valued relation (or function), two out of three of the following sets must be provided: tuples that certainly belong to the relation, tuples that certainly do not belong to the relation, and tuples for which it is unknown whether they belong to the relation or not. The third set can always be computed from the two given sets.

2.4 Procedures

The imperative programming language in our prototype system is LUA [11]. The main reason for this choice is the fact that LUA is a lightweight scripting

[1] Expressions that occur in practice, not artificially constructed sentences that do not really have meaning in real life.

language and also because it has a good `C++` API [12]. This facilitates on the one hand the compilation of programs written in DECLIMP and, on the other hand, the integration with the components of our DECLIMP interpreter, which is written in `C++`. When we do not take those reasons into account, any other imperative language is candidate.

In procedures, various reasoning methods on theories and structures can be called. Currently, the most important tasks supported by the DECLIMP-interpreter are the following:

Finite model expansion: Given a three-valued structure S and a theory T, find a completion of S to a two-valued structure that satisfies T. This is essentially a generalization of the reasoning task performed by ASP solvers, constraint programming systems, Alloy analyzers, etc. It is suitable for problems such as scheduling, planning and diagnosis. In our DECLIMP interpreter, model expansion is implemented by calls to the IDP system [22], which consists of the grounder GIDL [23] and solver MINISATID [13].

Finite model checking: Check whether a given two-valued structure is a model of a theory. This is an instance of model expansion and is implemented as such.

Constraint propagation: Deduce facts that must hold in all models of a given theory which complete a given three-valued structure. This is a useful mechanism in configuration systems [20] and for query answering in incomplete databases [4]. The propagation algorithm we implemented is described in [21].

Querying: Given an formula φ and a two-valued structure S, find all substitutions for free variables of φ that make φ true in S. The implementation of this mechanism makes use of Binary Decision Diagrams as described in [23].

Theorem proving: Given two theories T_1 and T_2, check whether $T_1 \models T_2$. This is implemented by calling a theorem prover provided by the user. In principle, any theorem prover that accepts TPTP [18] can be used.

Visualization: Show a visual representation of a given structure. We implement this by calling IDPDRAW, a tool for visualizing finite structures in which visual output is specified declaratively by definitions in our knowledge representation language or in ASP.

The values returned by the reasoning methods can be used as input for other reasoning methods and LUA-statements. We will illustrate this with an example in the next section.

3 Programming in DECLIMP

Say we want to write an application that allows players to solve sudoku puzzles. Such an application should be able to perform tasks such as generating puzzles, showing puzzles on the screen, checking whether solutions (player's choices) satisfy the sudoku rules, giving hints to the player, etc. In this application the different components we described before can clearly be distinguished: (1)

the background knowledge consists of a logic theory containing the well-known sudoku constraints;

$$\begin{array}{l}
\forall r \forall n \exists ! c : \mathit{Sudoku}(r,c) = n \\
\forall c \forall n \exists ! r : \mathit{Sudoku}(r,c) = n \\
\forall b \forall n \exists ! r \exists ! c : \mathit{InBlock}(b,r,c) \wedge \mathit{Sudoku}(r,c) = n \\
\forall b \forall r \forall c : \mathit{InBlock}(b,r,c) \Leftrightarrow \\
\qquad b = ((r-1) - ((r-1) \bmod 3)) + ((c-1) - ((c-1) \bmod 3))/3 + 1
\end{array}$$

(2) the data is stored in logical structures representing puzzles, and (partial and complete) solutions; and (3) the tasks we want it to perform, can be implemented using well-known inference methods. For example, "given a partial solution, complete the solution" is a typical model expansion task.

Below we show (a part of) a DECLIMP program. The code shows the use of an include statement and a namespace, and the declaration of a vocabulary `sudokuVoc` and a theory `sudokuTheory`, where the latter is simply an ASCII version of the theory shown above. Also note the `main` procedure at the bottom, which will automatically be called when the program is passed to the interpreter.

```
#include "grid.idp"

namespace sudoku {

    vocabulary sudokuVoc {
        extern vocabulary grid::simpleGridVoc
        type Num isa nat
        type Block isa nat
        Sudoku(Row,Col) : Num
        InBlock(Block,Row,Col)
    }

    theory sudokuTheory : sudokuVoc {
        ! r n : ?1 c : Sudoku(r,c) = n.
        ! c n : ?1 r : Sudoku(r,c) = n.
        ! b n : ?1 r c : InBlock(b,r,c) & Sudoku(r,c) = n.
        ! r c b : InBlock(b,r,c) <=>
            b = ((r-1)-((r-1)%3)) + ((c-1)-((c-1)%3))/3 + 1.
    }

    procedure solve(input) {
        return modelExpand(sudokuTheory,input)
    }

    procedure printSudoku(puzzle) {
        -- code for visualizing a sudoku puzzle.
    }

    procedure createSudoku() {
        math.randomseed(os.time())
        local puzzle = grid::makeEmptyGrid(9)
            -- defined in grid.idp

        stdoptions.nrmodels = 2
        local currsols = modelExpand(sudokuTheory,puzzle)
        while #currsols > 1 do
            repeat
                col = math.random(1,9)
                row = math.random(1,9)
                num = currsols[1][sudokuVoc::Sudoku](row,col)
            until num ~= currsols[2][sudokuVoc::Sudoku]
            (row,col)
            makeTrue(puzzle[sudokuVoc::Sudoku].graph,
            {row,col,num})
            currsols = modelExpand(sudokuTheory,puzzle)
        end

        printSudoku(puzzle)
    }
}

procedure main() {
    sudoku::createSudoku()
}
```

Let us have a closer look at procedure `createSudoku` for creating sudoku puzzles. First it initializes an empty puzzle by instantiating a new logical structure. This is done by calling a procedure `makeEmptyGrid` which instantiates a structure with data about a generic grid of a certain size, and then adding domains for numbers and blocks particular for sudoku grids.

The second part of the procedure adds numbers to the grid until there is only one solution left for the puzzle. This is realized by performing model expansion (by calling `modelExpand`) to find two models of the theory that extend the given partially filled in puzzle. When two models are found, the algorithm selects a number that is unique for the first solution (that is, the number at the same position in the second solution is different) and is not yet present in the puzzle. When such an entry is found, it is added to the puzzle by making the tuple {`row`, `col`, `num`} true in the interpretation of the function `Sudoku(Row,Col):Num`. Next, the procedure asks for two new models, and the process starts over. When only one model is found, the iteration stops, and procedure `printSudoku` is called to show the result on the screen using the visualization tool mentioned in the previous section.

4 Related Work

There have been many proposals in the literature to combine procedural and declarative languages. A frequently occuring combination is that of a procedural language in which a program can post constraints expressed in an (often ad-hoc) declarative constraint language, while other primitives allow to call the constraint-solving process on the constraint store, express heuristics or call other processes, for example to edit or visualize output. Examples of systems with such languages are CPLEX [1], Mozart [19] and Comet [15]. These systems differ from declimp in the sense that they offer only one kind of inference, namely constraint solving. A similar remark can be made about CLP and Prolog systems with support for constraint propagation. There the "procedural language" is the Prolog language under its procedural semantics. In our system high-level concepts such as vocabularies, theories and structures are treated as first-class citizens that can be operated upon by arbitrary inference and processing tools, which offers more flexibility.

For another group of systems, control over execution of programs is in hands of one inference mechanism – or at least that inference is the main mechanism – and an integrated procedural language then allows users to stear some aspects of the inference mechanism, or for example format input and output, but do not allow to take over control. Examples of such systems are clingo [9] and Zinc [14]. The procedural languages in these systems have a more limited task then the one in declimp. In declimp the procedures are in control during execution, not just one of the inference mechanisms.

5 Conclusion

We have presented a knowledge-based programming environment, providing a declarative language for expressing background knowledge, an imperative programming language for writing procedures, and logical structures for expressing concrete data. The system also provides state-of-the-art inference tools for performing various reasoning tasks.

We believe that a programming environment like the one proposed here overcomes some of the limitations of "single-programming-style" paradigms, by allowing a programmer to express the different types of information in software applications in appropriate languages. Making this explicit distinction between different types of information will increase readability, maintainability and reusability of programming code.

The prototype presented here has evolved into a new version of the IDP system. It can be obtained from http://dtai.cs.kuleuven.be/krr/software.

References

1. IBM, ILOG CPLEX optimizer. http://www.ibm.com/software/integration/optimization/cplex-optimizer
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications, 2nd edn. Cambridge University Press, Cambridge (2007)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: theory and implementation. In: de la Banda, M.G., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 407–424. Springer, Heidelberg (2008)

4. Denecker, M., Calabuig, Á.C., Bruynooghe, M., Arieli, O.: Towards a logical reconstruction of a theory for locally closed databases. ACM Trans. Database Syst. **35**(3), 22:1–22:60 (2010)
5. Denecker, M., De Schreye, D., Willems, Y.: Terms in Logic programs: a problem with their semantics and its effect on the programming methodology. CCAI: J. Integr. Study Artif. Intell., Cogn. Sci. Appl. Epistemology **7**(3–4), 363–383 (1990)
6. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Trans. Comput. Logic (TOCL) **9**(2), 14:1–14:52 (2008)
7. Denecker, M., Vennekens, J.: Building a knowledge base system for an integration of logic programming and classical logic. In: Pontelli, E., Garcia de la Banda, M. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 71–76. Springer, Heidelberg (2008)
8. Fitting, M.: First-order logic and automated theorem proving, 2nd edn. Springer-Verlag New York Inc., Secaucus, NJ, USA (1996)
9. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. http://downloads.sourceforge.net/potassco/guide.pdf (2010)
10. The IDP system. http://dtai.cs.kuleuven.be/krr/software (2012)
11. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: Lua - an extensible extension language. Softw.: Pract. Experience **26**(6), 635–652 (1996)
12. Ierusalimschy, R., Henrique de Figueiredo, L., Celes, W.: Passing a language through the eye of a needle. Queue **9**, 20:20–20:29 (2011)
13. Mariën, M., Wittocx, J., Denecker, M., Bruynooghe, M.: SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In: Büning, H.K., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 211–224. Springer, Heidelberg (2008)
14. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de Banda, M.G., Wallace, M.: The design of the Zinc modelling language. Constraints **13**(3), 229–267 (2008)
15. Michel, L., Van Hentenryck, P.: The comet programming language and system. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 881–881. Springer, Heidelberg (2005)
16. Mitchell, D.G., Ternovska, E., Hach, F., Mohebali, R.: Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006–24, Simon Fraser University, Canada (2006)
17. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. Theory Pract. Logic Program. (TPLP) **7**(3), 301–353 (2007)
18. Sutcliffe, G.: The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. J. Autom. Reasoning **43**(4), 337–362 (2009)
19. Van Roy, P. (ed.): MOZ 2004. LNCS, vol. 3389. Springer, Heidelberg (2005)
20. Vlaeminck, H., Vennekens, J., Denecker, M.: A logical framework for configuration software. In: Porto, A., Javier López-Fraguas, F. (eds.) PPDP, pp. 141–148. ACM, New York (2009)
21. Wittocx, J., Denecker, M., Bruynooghe, M.: Constraint propagation for extended first-order logic. In: CoRR, abs/1008.2121 (2010)
22. Wittocx, J., Mariën, M., Denecker, M.: The IDP system: a model expansion system for an extension of classical logic. In: Denecker, M. (ed.) LaSh, pp. 153–165 (2008)
23. Wittocx, J., Mariën, M., Denecker, M.: Grounding FO and FO(ID) with bounds. J. Artif. Intell. Res. **38**, 223–269 (2010)

WLP System Descriptions

Computing with Logic as Operator Elimination: The ToyElim System

Christoph Wernhard(✉)

Technische Universität Dresden, Dresden, Germany
christoph.wernhard@tu-dresden.de

Abstract. A prototype system is described whose core functionality is, based on propositional logic, the elimination of second-order operators, such as Boolean quantifiers and operators for projection, forgetting and circumscription. This approach allows to express many representational and computational tasks in knowledge representation – for example computation of abductive explanations and models with respect to logic programming semantics – in a uniform operational system, backed by a uniform classical semantic framework.

1 Computation with Logic as Operator Elimination

We pursue an approach to computation with logic emerging from three theses:

1. Classical first-order logic extended by some second-order operators suffices to express many techniques of knowledge representation.

Like the standard logic operators, second-order operators can be defined *semantically*, by specifying the requirements on an interpretation to be a model of a formula whose principal functor is the operator, depending only on semantic properties of the argument formulas. Neither control structure imposed over formulas (e.g. Prolog), nor formula transformations depending on a particular syntactic shape (e.g. Clark's completion) are involved. Compared to classical first-order formulas, the second-order operators give additional expressive power. Circumscription is a prominent knowledge representation technique that can be expressed with second-order operators, in particular predicate quantifiers [1].

2. Many computational tasks can be expressed as elimination of second-order operators.

Elimination is a way to computationally process second-order operators, for example Boolean quantifiers with respect to propositional logic: The input is a formula which may contain the operator, for example a quantified Boolean formula such as $\exists q\ ((p \leftarrow q) \land (q \leftarrow r))$. The output is a formula that is equivalent to the input, but in which the operator does not occur, such as, with respect to the formula above, the propositional formula $p \leftarrow r$. Let us assume that the method used to eliminate the Boolean quantifiers returns formulas in which not just the quantifiers but also the quantified propositional variables do not occur. This syntactic condition is usually met by elimination procedures. Our method then subsumes a variety of tasks: Computation of uniform interpolants, QBF and SAT solving, as well as computation of certain forms of abductive explanations, of propositional circumscription, and of stable models, as will be outlined below.

> *3. Depending on the application, outputs of computation with logic are conveniently represented by formulas meeting syntactic criteria.*

If results of elimination are formulas characterized just up to semantics, they may contain redundancies and be in a shape that is difficult to comprehend. Thus, they should be subjected to simplification and canonization procedures before passed to humans or to machine clients. The output format depends on the application problem: What is a CNF of the formula? Are certain facts consequences of the formula? What are the models of the formula? What are its minimal models? What are its 3-valued models with respect to some encoding into 2-valued logics? Corresponding answers can be computed on the basis of normal form representations of the elimination outputs: CNFs, DNFs, full DNFs, and prime implicant forms. Of course, transformation into such normal forms might by itself be an expensive task. Second-order operators allow to counter this by specifying a small set of application relevant symbols that should be included in the output, e.g. by Boolean quantification upon the irrelevant atoms.

2 Features of the System

ToyElim[1] is a prototype system developed to investigate operator elimination from a pragmatic point of view with small applications. For simplicity, it is based on propositional logic, although its characteristic features should transfer to first-order logic. It supports a set of second-order operators that have been semantically defined in [11,13,15].

Formula Syntax. As the system is implemented in Prolog, formulas are represented by Prolog terms, the standard connectives corresponding to `true`/0, `false`/0, `~`/1, `,`/2, `;`/2, `->`/2, `<-`/2, `<->`/2. Propositional atoms are represented by Prolog atoms or compound ground terms. The system supports propositional expansion with respect to finite domains of formulas containing first-order quantifiers.

Forgetting. Existential Boolean quantification $\exists p\ F$ can be expressed as *forgetting* [4,11] in formula F about atom p, written $\mathsf{forget}_{\{p\}}(F)$, represented by `forg([p],` F'`)` in system syntax, where F' is the system representation of F. To get an intuition of forgetting, consider the equivalence $\mathsf{forget}_{\{p\}}(F) \equiv F[p\backslash\mathsf{true}] \vee F[p\backslash\mathsf{false}]$, where $F[p\backslash\mathsf{true}]$ ($F[p\backslash\mathsf{false}]$) denotes F with all occurrences of p replaced by true (false). Rewriting with this equivalence constitutes a naive method for eliminating the forgetting operator. The formula $\mathsf{forget}_{\{p\}}(F)$ can be said to express the same as F about all other atoms than p, but nothing about p.

Elimination and Pretty Printing of Formulas. The central operation of the ToyElim system, elimination of second-order operators, is performed by the

[1] http://cs.christophwernhard.com/toyelim/, under GNU Public License.

predicate `elim`(F,G), with input formula F and output formula G. For example, define as extension of `kb1`/1 a formula (after [3]) as follows:

```
kb1(((shoes_are_wet <- grass_is_wet),
       (grass_is_wet <- rained_last_night),
       (grass_is_wet <- sprinkler_was_on))).
```
(1)

After consulting this, we can execute the following query on the Prolog toplevel:

```
?- kb1(F), elim(forg([grass_is_wet], F), G), ppr(G).
```
(2)

This results in binding `G` to the output of eliminating the forgetting about `grass_is_wet`. The predicate `ppr`/1 is one of several provided predicates for converting formulas into application adequate shapes. It prints its argument as CNF with clauses written as reverse implications:

```
((shoes_are_wet <- rained_last_night),
 (shoes_are_wet <- sprinkler_was_on)).
```
(3)

Scopes. So far, the first argument of forgetting has been a singleton set. More generally, it can be an arbitrary set of atoms, corresponding to nested existential quantification. Even more generally, also polarity can be considered: Forgetting can, for example, be applied only to those occurrences of an atom which have negative polarity in a NNF formula. This can be expressed by *literals* with explicitly written sign in the first argument of the forgetting operator. Forgetting about an atom is equivalent to nested forgetting about the positive and the negative literal with that atom. In accord with this observation, we technically consider the first argument of forgetting always as a *set of literals*, and regard an unsigned atom there as a shorthand representing both of its literals. For example, `[+grass_is_wet, shoes_are_wet]` is a shorthand for `[+grass_is_wet, +shoes_are_wet, -shoes_are_wet]`. Not just forgetting, but, as shown below, also other second-order operators have a set of literals as parameter. Hence, we refer to a set of literals in this context by a special name, as *scope*.

Projection. In many applications it is useful to make explicit not the scope that is “forgotten” about, but what is preserved. The *projection* [11] of formula F onto scope S, which can be defined for scopes S and formulas F as $\mathsf{project}_S(F) \equiv \mathsf{forget}_{\mathsf{ALL}-S}(F)$, where ALL denotes the set of all literals, serves this purpose. Vice versa, forgetting could be defined in terms of projection: $\mathsf{forget}_S(F) \equiv \mathsf{project}_{\mathsf{ALL}-S}(F)$. The call to `elim`/2 in the query (2) can equivalently be expressed with projection instead of forgetting by

```
elim(proj([shoes_are_wet, rained_last_night, sprinkler_was_on], F).
```
(4)

User Defined Logic Operators – An Example of Abduction. ToyElim allows the user to specify macros for use in the input formulas of `elim`/2. The following example extends the system by a logic operator `gwsc` for a variant of the weakest sufficient condition [8], characterized in terms of projection:

```
:- define_elim_macro(gwsc(S, F, G), ~proj(complements(S),(F, ~G))).
```
(5)

Here `complements(S)` specifies the set of the literal complements of the members of the scope specified by `S`. The term `gwsc(S, F, G)` is the system syntax for $\mathsf{gwsc}_S(F, G)$, the *globally weakest sufficient condition* [15] of formula G on scope S within formula F, which satisfies the following: A formula H is equivalent to $\mathsf{gwsc}_S(F, G)$ if and only if it holds that (1.) $H \equiv \mathsf{project}_S(H)$; (2.) $F \models H \rightarrow G$; (3.) For all formulas H' such that $H' \equiv \mathsf{project}_S(H')$ and $F \models H' \rightarrow G$ it holds that $H' \models H$. With the `gwsc` operator certain abductive tasks [3] can be expressed. The following query, for example, yields abductive explanations for `shoes_are_wet` in terms of {`rained_last_night`, `sprinkler_was_on`} with respect to the knowledge base (1):

```
?- kb1(F),
   elim(gwsc([rained_last_night, sprinkler_was_on], F, shoes_are_wet),
        G),
   ppp(G).
```
(6)

The predicate `ppp/1` serves, like `ppr/1`, to convert formulas to application adequate shape. It writes a prime implicate form of its input in list notation. In the example the output has two clauses, each representing an alternate explanation:

$$[[\mathtt{rained\_last\_night}], [\mathtt{sprinkler\_was\_on}]]. \qquad (7)$$

Scope-Determined Circumscription. A further second-order operator supported by ToyElim is *scope-determined circumscription* [15]. The corresponding functor `circ` has, like `proj` and `forg`, a scope specifier and a formula as arguments. It allows to express *parallel predicate circumscription with varied predicates* [5] (only propositional, since the system is based on propositional logic). The scope specifier controls the effect of circumscription: Atoms that occur just in a *positive* literal in the scope are minimized; symmetrically, atoms that occur just *negatively* are maximized; atoms that occur in *both polarities* are fixed; and atoms that do *not at all* occur in the scope are allowed to vary. For example, the scope specifier, `[+abnormal, bird]`, a shorthand for `[+abnormal, +bird, -bird]`, expresses that `abnormal` is minimized, `bird` is fixed, and all other predicates are varied.

Predicate Groups and Systematic Renaming. Semantics for knowledge representation sometimes involve what might be described as handling different occurrences of a predicate differently – for example depending on whether they are subject to negation as failure. If such semantics are to be modeled with classical logic, then these occurrences can be identified by using distinguished predicates, which are equated with the original ones when required. To this end, ToyElim supports the handling of *predicate groups*: The idea is that each predicate actually is represented by several *corresponding* predicates $p^0, \ldots, p^n$, where the superscripted index is called *predicate group*. In the system syntax, the predicate group of an atom is represented within its main functor: If the group is larger than 0, the main functor is suffixed by the group number; if it is 0, the main functor does not end in a number. For example $p(a)^0$ and $p(a)^1$ are represented by `p(a)` and `p1(a)`, respectively. In scope specifiers, a number is used as shorthand for the set of all literals whose atom is from the indicated group, and a number in a sign functor for the set of those literals which have that sign and whose atom is from the indicated group. For example, `[+(0), 1]` denotes the union of the set of all positive literals whose atom is from group 0 and of the set of all literals whose atom is from group 1. Systematic renaming of all atoms in a formula that have a specific group to their correspondents from another group can be expressed in terms of forgetting [13]. The ToyElim system provides the second-order operator `rename` for this. For example, `rename([1-0],` F`)` is equivalent to F after eliminating second-order operators, followed by replacing all atoms from group 1 with their correspondents from group 0.

An Example of Modeling a Logic Programming Semantics. Scope-determined circumscription and predicate groups can be used to express the characterization of the stable models semantics in terms of circumscription [7] (described also in [6,13]). Consider the following knowledge base:

```
kb2(((shoes_are_wet <-  grass_is_wet),
      (grass_is_wet <-  sprinkler_was_on, ~sprinkler_was_abnormal1),
     sprinkler_was_on)).
```
(8)

Group 1 is used here to indicate atoms that are subject to negation as failure: All atoms in (8) are from group 0, except for `sprinkler_was_abnormal1`, which is from 1. The user defined operator `stable` renders the stable models semantics:

```
:- define_elim_macro(stable(F),  rename([1-0],  circ([+(0),1], F))).
```
(9)

The following query then yields the stable models:

```
:- kb2(F),  elim(stable((F)),  G),  ppp(G).
```
(10)

The result is displayed with `ppp/1`, as in query (6). It shows here a DNF with a single clause, representing a single model. The positive members of the clause constitute the answer set

```
[[grass_is_wet, shoes_are_wet,  ~sprinkler_was_abnormal,
  sprinkler_was_on]].
```
(11)

If it is only of interest whether `shoes_are_wet` is a consequence of the knowledge base under stable models semantics, projection can be applied to obtain a smaller result. The query

```
:-  kb2(F),  elim(proj([shoes_are_wet],  stable(F)),  G),  ppp(G).
```
(12)

will effect that the DNF `[[shoes_are_wet]]` is printed.

3 Implementation

The ToyElim system is implemented in SWI-Prolog and can invoke external systems such as SAT and QBF solvers. It runs embedded in the Prolog environment, allowing for example to pass intermediate results between its components through Prolog variables, as exemplified by the queries shown above.

The implementation of the core predicate `elim`/2 maintains a formula which is gradually rewritten until it contains no more second-order operators. It is initialized with the input formula, preprocessed such that only two primitively supported second-order operators remain: forgetting and renaming. It then proceeds in a loop where alternately equivalence preserving simplifying rewritings are applied, and a subformula is picked and handed over for elimination to a specialized procedure. The simplifying rewritings include distribution of forgetting over subformulas and elimination steps that can be performed with low cost [12]. Rewriting of subformulas with the Shannon expansion enables low-cost elimination steps. It is performed at this stage if the expansion, combined with low-cost elimination steps and simplifications, does not lead to an increase of the formula size. The subformula for handing over to a specialized method is picked with the following priority: First, an application of forgetting upon the whole signature of a propositional argument, which can be reduced by a SAT solver to either `true` or `false`, is searched. Second, a subformula that can be reduced analogously by a QBF solver, and finally a subformula which properly requires elimination of forgetting. For the latter, ToyElim schedules a portfolio of different methods, where currently two algorithmic approaches are supported: Resolvent generation (SCAN, Davis-Putnam method) and rewriting of subformulas with the Shannon expansion [10,12]. Recent SAT preprocessors partially perform variable elimination by resolvent generation. *Coprocessor* [9] is such a preprocessor that is configurable such that it can be invoked by ToyElim for the purpose of performing the elimination of forgetting.

4 Conclusion

We have seen a prototype system for computation with logic as elimination of second-order operators. The system helped to concretize requirements on systems following this approach, concerning the user interface and the processing methods. In the long run, such a system should be based on more expressive logics than propositional logic. ToyElim is just a first pragmatic attempt, taking advantage of recent advances in SAT solving. A major difference in a first-order setting is that computations of elimination tasks then inherently do not terminate for all inputs.

Research on the improvement of elimination methods includes further consideration of techniques from SAT preprocessors, investigation of tableau and DPLL-like techniques [2,12], and, in the context of first-order logic, the so called *direct methods* [1]. In addition, it seems worth to investigate further types of output: incremental construction, like enumeration of model representations, and representations of proofs.

So far, the system has been applied in teaching and to investigate logic programming semantics. Some application examples are provided on its Web page. One of them shows generalizations of several logic programming semantics that allow to exempt specified predicates from the closed world assumption [14], another one shows how skeptical abduction with respect to the stable models semantics and to the three-valued partial stable models semantics can be expressed and implemented on the basis of the globally weakest sufficient condition. A third example includes a small toy knowledge base about a touristic real-world scenario to illustrate a range of further applications in a rudimentary way: Extraction of knowledge concerning a given signature and of knowledge at a particular level of abstraction, as well as certain forms of schema mapping, abduction, intensional answers, knowledge base modularization and data protection.

The approach of computation with logic by elimination leads to a system that provides a uniform user interface covering many tasks, like satisfiability checking, computation of abductive explanations and computation of models for various logic programming semantics. Variants of established concepts can be easily expressed on a clean semantic basis and made operational. The approach supports the co-existence of different knowledge representation techniques in a single system, backed by a single classical semantic framework. This seems a necessary precondition for logic libraries that accumulate knowledge independently of some particular application.

References

1. Gabbay, D.M., Schmidt, R.A., Szałas, A.: Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications. College Publications, London (2008)
2. Huang, J., Darwiche, A.: DPLL with a trace: from SAT to knowledge compilation. In: IJCAI-05, pp. 156–162. Morgan Kaufmann (2005)
3. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. J. Logic Comput. **2**(6), 719–770 (1993)
4. Lang, J., Liberatore, P., Marquis, P.: Propositional independence - formula-variable independence and forgetting. J. Artif. Intell. Res. **18**, 391–443 (2003)
5. Lifschitz, V.: Circumscription. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, pp. 298–352. Oxford University Press, Oxford (1994)
6. Lifschitz, V.: Twelve definitions of a stable model. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 37–51. Springer, Heidelberg (2008)
7. Lin, F.: A study of nonmonotonic reasoning. Ph.D. thesis, Stanford University (1991)
8. Lin, F.: On strongest necessary and weakest sufficient conditions. Artif. Intell. **128**, 143–159 (2001)
9. Manthey, N.: Coprocessor 2.0 – a flexible CNF simplifier. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 436–441. Springer, Heidelberg (2012)

10. Murray, N.V., Rosenthal, E.: Tableaux, path dissolution and decomposable negation normal form for knowledge compilation. In: Mayer, M.C., Pirri, F. (eds.) TABLEAUX 2003. LNCS (LNAI), vol. 2796, pp. 165–180. Springer, Heidelberg (2003)
11. Wernhard, C.: Literal projection for first-order logic. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 389–402. Springer, Heidelberg (2008)
12. Wernhard, C.: Tableaux for projection computation and knowledge compilation. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 325–340. Springer, Heidelberg (2009)
13. Wernhard, C.: Circumscription and projection as primitives of logic programming. In: Technical Communications of the ICLP 2010. LIPIcs, vol. 7, pp. 202–211 (2010)
14. Wernhard, C.: Forward human reasoning modeled by logic programming modeled by classical logic with circumscription and projection. Technical Report Knowledge Representation and Reasoning 11-07, Technische Universität Dresden (2011)
15. Wernhard, C.: Projection and scope-determined circumscription. J. Symb. Comput. **47**, 1089–1108 (2012)

Coprocessor – a Standalone SAT Preprocessor

Norbert Manthey(✉)

Knowledge Representation and Reasoning Group,
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

Abstract. In this work a stand-alone preprocessor for SAT is presented that is able to perform most of the known preprocessing techniques. Preprocessing a formula in SAT is important for performance since redundancy can be removed. The preprocessor is part of the SAT solver *riss* [9] and is called *Coprocessor*. Not only *riss*, but also *MiniSat 2.2* [11] benefit from it, because the *SatELite* preprocessor of *MiniSat* does not implement recent techniques. By using more advanced techniques, *Coprocessor* is able to reduce the redundancy in a formula further and improves the overall solving performance.

1 Introduction

In theory SAT problems with n variables have a worst case execution time of $O(2^n)$ [2]. Reducing the number of variables results in a theoretically faster search. However, in practice the number of variables does not correlate with the runtime. The number of clauses highly influences the performance of unit propagation, which consumes almost 90 percent of the search time of modern CDCL solvers. Preprocessing helps to reduce the size of the formula by removing variables and clauses that are redundant.

During the last years many preprocessing techniques have been invented and implemented in separate tools or only within a solver itself. Coprocessor tries to make these techniques available without being shipped with a solver.[1] Preprocessing techniques can be classified into two categories: Techniques, which change a formula so that a model for the preprocessed formula is not a model for the original formula, are called *satisfiability-preserving techniques*. For these techniques undo information has to be stored. For the second category, which is called *equivalence-preserving techniques*, this information is not required, because the preprocessed and original formula are equivalent.

This paper is structured in the following way. An overview of the implemented techniques is given in Sect. 2. Details on *Coprocessor*, a format for storing the undo information and a comparison to *SatELite*, one of the currently mostly used preprocessors, is given in Sect. 3. Finally, a conclusion is given in Sect. 4.

[1] The tool is available at http://tools.computational-logic.org.

DOI: 10.1007/978-3-642-41524-1_18, © Springer-Verlag Berlin Heidelberg 2013

2 Preprocessor Techniques

The notation used to describe the preprocessor is the following: variables are numbers and literals are positive or negated variables, e.g. 2 and $\neg 2$. A clause C is a disjunction of a set of literals, denoted by $[l_1, \dots, l_n]$. A formula is a conjunction of clauses. The original formula will be referred to as F, the preprocessed formula is called F'. Unit propagation on F is denoted by $\mathrm{BCP}(l)$, where l is the literal that is assigned to *true*.

2.1 Basic Preprocessing Rules

Techniques as *Boolean Constraint Propagation*, *Subsumption* and *Resolution* are considered basic techniques. All of them preserve the equvalence of the formula. Given a formula F, where a unit clause $[l] \in F$ is found. Hence, to safisfy F the literal l has to be assigned true. Next, the formula F can be simplified accordingly: all satisfied clauses are removed and from all the remaining clauses all occurrences of $\neg l$ are removed.

Subsumption can be applied to two clauses C and D of the formula, if the following property holds: all literals of C also appear in D. In this case, the D is always satisfied when C is satisfied. Therefore, the clause D can be removed from the formula.

A special form of resolution is *self-subsuming resolution*, which is also called *strengthening*. In general, if there are two clauses $C = [l, c_1, \dots, c_n]$ and $D = [\neg l, d_1, \dots, d_m]$ the resolvent $E = C \otimes D$ is defined as follows:

$$D = [c_1,\ dots, c_n, d_1, \dots, d_m].$$

A property of this resolvent is that whenever both C and D are satisfied, also E is satisfied. Thus, adding resolvents to a formula preserves equvalence. The special case of self-subsuming resolution is the following: C and D are resolved to produce E. Next, E is added to the formula and afterwards it either C or D can be removed because they are subsumed by E. Usually, E is only added to the formula if it subsumes on of the other two clauses.

2.2 Satisfiability-Preserving Techniques

The following techniques change F in a way, that models of F' are no model for F. Therefore, undo information is required. Undoing of these methods has to be done carefully, because the order influences the resulting assignment. All the elimination steps have to be undone in the opposite order they have been applied before [6].
Variable Elimination (VE) [3] is a technique to remove variables from the formula by resolving the according clauses in which the variable occurs. Given two sets of clauses: C_x with the literal x and $C_{\overline{x}}$ with $\overline{x}$. Let G be the union of these two sets $G \equiv C_x \cup C_{\overline{x}}$. Resolving C_x and $C_{\overline{x}}$ results in a new set of clauses G', where tautologies are not included. It is shown in [3] that G can be replaced

by G' without changing the satisfiability of the formula. If a model is needed for F, then the partial model can be extended using the original clauses F to assign variable x. Usually, applying VE to a variable results in a larger number of clauses. In state-of-the-art preprocessors VE is only applied to a variable if the number of clauses decreases. The resulting formula depends on the order of the eliminated variables. *Pure literal elimination* is a special case of VE.
Blocked Clause Elimination (BCE) [7] removes redundant *blocked clauses*. A clause C is blocked if it contains a blocking literal l. A literal l is a blocking literal, if l is part of C, and for each clause $C' \in F$ with $\bar{l} \in C'$ the resolvent $C \otimes_l C'$ is a tautology [4,7]. Removing a blocked clause from F changes the satisfying assignments [4]. BCE is confluent [7].
Equivalence Elimination (EE) [5] removes a literal l if it is equivalent to another literal l'. Only one literal per equivalence class is kept. Equivalent literals can be found by finding strongly connected components in the binary implication graph (BIG). The BIG represents all implications in the formula by directed edges $l \rightarrow l'$ between literals that occur in a clause $[\ \bar{l}, l'\ ]$. If a cycle $a \rightarrow b \rightarrow c \rightarrow a$ is found, there is also a cycle $\overline{a} \rightarrow \overline{b} \rightarrow \overline{c} \rightarrow \overline{a}$ and therefore $a \equiv b \equiv c$ can be shown and applied to F by replacing b, and c by a. Finally, double literal occurrences and tautologies are removed.

2.3 Equivalence-Preserving Techniques

Equivalence-preserving techniques can be applied in any order, because the preprocessed formula is equivalent to the original one. By combining the following techniques with satisfiability-preserving techniques the order of the applied techniques has to be stored, to be able to undo all changes correctly.
Hidden Tautology Elimination (HTE) [4] is based on the clause extension *hidden literal addition* (HLA). After the clause C is extended by HLA, C is removed if it is tautology. The HLA of a clause C with respect to a formula F is computed as follows: Let l be a literal of C and $[l', l] \in F \setminus \{C\}$. If such a literal l' can be found, C is extended by $C := C \cup \overline{l'}$. This extension is applied until fix point. HLA can be regarded as the opposite operation of self subsuming resolution. The algorithm is linear time in the number of variables [4].
Probing [8] is a technique to simplify the formula by propagating variables in both polarities l and $\bar{l}$ separately and comparing their implications or by propagating all literals of a clause $C = [l_1, \ldots, l_n]$, because it is known that one of the candidates has to be satisfied.

Probing a single variable can find a conflict and thus finds a new unit. The following example illustrates the other cases:

$$BCP(1) \Rightarrow 2, 3, 4, \neg 5, \neg\ 7$$
$$BCP(\overline{1}) \Rightarrow 2, \neg\ 4, 6, 7$$

To create a complete assignment, variable *1* has to be assigned and both possible assignments imply 2, so that 2 can be set to *true* immediately. Furthermore, the

equivalences $4 \equiv 1$ and $\overline{7} \equiv 1$ can be found. These equivalences can also be eliminated. Probing all literals of a clause can find only new units.
Vivification (also called *Asymmetric Branching*) [12] reduces the length of a clause by propagating the negated literals of a clause $C = [l_1, \ldots, l_n]$ iteratively until one of the following three cases occurs:

1. BCP($\{\overline{l_1}, \ldots, \overline{l_i}\}$) results in an empty clause for $i < n$.
2. BCP($\{\overline{l_1}, \ldots, \overline{l_i}\}$) implies another literal l_j of the C with $i < j < n$
3. BCP($\{\overline{l_1}, \ldots, \overline{l_i}\}$) implies another negated literal $\overline{l_j}$ of the C with $i < j \leq n$

In the first case, the unsatisfying partial assignment is disallowed by adding a clause $C' = [l_1, \ldots, l_i]$. The clause C' subsumes C. The implication $\overline{l_1} \wedge \cdots \wedge \overline{l_i} \rightarrow l_j$ in the second case results in the clause $C' = [l_1, \ldots, l_i, l_j]$ that also subsumes C. Formulating the third case into a clause $C' = [l_1, \ldots, l_i, \overline{l_j}]$ subsumes C by applying self subsumption to $C'' = C \otimes_{l_j} C' = [l_1, \ldots, l_{j-1}, l_{j+1}, \ldots, l_n]$.
Extended Resolution (ER) [1] introduces a new variables v to a formula that is equivalent to a disjunction of literals $v \equiv l \vee l'$. All clauses in F are updated by removing the pair and adding the new variable instead. It has been shown, that ER is good for shrinking the proof size for unsatisfiable formulas. Applying ER during search as in [1] resulted in a lower performance of *riss*, so that this technique has been put into the preprocessor and replaces the most frequent literal pairs. Still, no deep performance analysis has been done on this technique.

3 Coprocessor

The preprocessor of *riss*, *Coprocessor*, implements all the techniques presented in Sect. 2 and introduces many algorithm parameters. A description of these parameters can be found in the help of *Coprocessor*.[2] The techniques are executed in a loop on F, so that for example the result of HTE can be processed with VE and afterwards HTE tries to eliminate clauses again.

Coprocessor provides a black-list and a white-list of variables. Variables on the white-list are tabooed for any non-model-preserving techniques, thus their semantic is the same in F'. Variables on the blacklist are always removed by VE. Furthermore, the resulting formula can be compressed. Due to assigned and removed variables, the variables of the reduct of F' are usually not dense any more, resulting in unnecessary memory overhead. To overcome this weakness, *Coprocessor* fills these gaps with present variables and stores the already assigned variables for postprocessing the model. The compression cannot be combined with the white-list. Another transformation can be performed, namely the conversion from encoded CSP domains from the direct encoding to the regular encoding as described in [10].

[2] The source code can be found at http://www.ki.inf.tu-dresden.de/~norbert.

3.1 The Map File Format

A map file is used to store the undo information for postprocessing a model of F' such that it becomes a model for F again. The map file and the model for F' can be used to restore the model for F by giving this information to *Coprocessor*. The following information has to be stored:

Once	**Per elimination step**
Compression table	Variable elimination
Equivalence classes	Blocked clause elimination
	Equivalence elimination step

The map file is divided into two parts. A partial example file for illustration is given in Table 1. The format is described based on this example file. Each occurring case is also covered in the description. The first line has to state "original variables" (line 1). This number is specified in the next line (line 2). Next, the compression information is given by beginning with either "compress table" (line 3), if there is a table, or "no table", if there is no compression. Afterwards, the tables are given where each starts with a line "table k v" and k represents the number of the table and v is the number of variables before the applied compression (line 4). The next line gives the compression by simply giving a mapping that depends on the order: the ith number in the line is the variable that is represented by variable i in the compressed formula (line 5). The line is closed by a 0, so that a standard clause parser can be used. The next line introduces the assignments in the original formula by saying "units k" (line 6). The following 0-terminated line lists all the literals that have been assigned *true* in the original formula (line 7). The compression is completed with a line stating "end table" (line 8).

At the moment, only a single compression is supported, and thus, k is always 0. The compression is applied after all other techniques and therefore the following details are given with respect to the decompressed preprocessed formula F'. The next static information is the literals of the EE classes. They are introduced by a line "ee table" (line 9). The following lines represent the classes where the first element is the representative of the class that is in F'(line 10-12). Each class is ordered ascending, so that the EE information can be stored as a tree.

Table 1. Example map file

```
1 original variables                 9 ee table               17 bce 10623
2 30867                             10 1 -19 0                18 -10429 10623 -30296 0
3 compress tables                   11 2 -20 0                19 ...
4 table 0 30867                     12 ...                    20 ve 812 1
5 1 2 3 5 6 7 9 10 11 ... 0         13 postprocess stack      21 -812 -74 0
6 units 0                           14 ee                     22 ve 6587 4
7 -31 32 ... -30666 -30822 0        15 bce 523                23 6587 6615 0
8 end table                         16 -81 523 -6716 0        24 ...
```

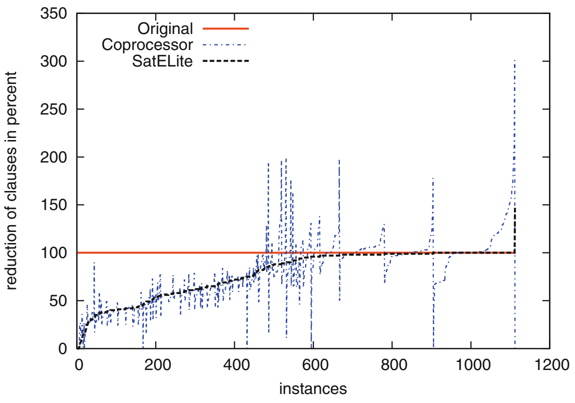


Fig. 1. Comparing the relative clause reduction of the preprocessors

Again, each class is terminated by a 0. Finally, the postprocess stack is given and preluded with a line "postprocess stack" (line 13). Afterwards the eliminations of BCE and VE are stored in the order they have been performed. BCE is prefaced with a line "bce l" where l is the blocking literal (line 15,17). The next line gives the according blocked clause (line 16,18). For VE the first line is "ve v n" where v is the eliminated variable and n is the number of clauses that have been replaced (line 20,22). The following n lines give the according clauses (line 21,23-26). Finally, for EE it is only stated that EE has been applied by writing a line "ee", because postprocessing EE depends also on the variables that are present at the moment (line 14). Some of the variables might already be removed at the point EE has been run, so that it is mandatory to store this information.

3.2 Preprocessor Comparison

A comparison of the relative formula reductions of *Coprocessor* and the current standard preprocessor SatELite [3] is given in Fig. 1 and has been performed on 1155 industrial and crafted instances from recent SAT Competitions and SAT Races[3]. Due to extended resolution, *Coprocessor* can increase the number of clauses, whereby the average length is still reduced. *Coprocessor* is also able to reduce the number of clauses more than SatELite. The instances are ordered by the reduction of SatELite so that the plot for *Coprocessor* produces peaks.

Since SatELite and *MiniSAT* [11] have been developed by the same authors, the run times of *MiniSAT* with the two preprocessors are compared in Fig. 2. Comparing these run times of *MiniSAT* (MS) combined with the preprocessors, it can be seen that by using a preprocessor the performance of the solver is much

[3] For more details see http://www.ki.inf.tu-dresden.de/~norbert/paperdata/WLP2011.html.

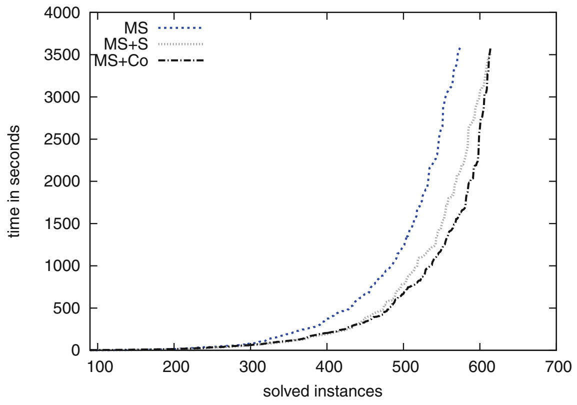


Fig. 2. Runtime comparison of MiniSAT and the preprocessors SatELite and Coprocessor

higher. Furthermore, the combination with *Coprocessor* (MS+Co) solves more instances than *SatELite* (MS+S) for most of the timeouts.

4 Conclusion and Future Work

This work introduces the SAT preprocessor *Coprocessor* that implements almost all known preprocessing techniques and additional features. Experiments showed that the default configuration of *Coprocessor* performs better than *SatELite* when combined with *MiniSAT 2.2*. *Coprocessor* provides many parameters for all its techniques that can be optimized for special use cases. Additionally, a map file format is presented that can be used to store the preprocessing information and to re-construct the model for the original formula if the model for the preprocessed formula is given. Future development of this preprocessor includes adding the latest techniques such as HLE and HLA [4,5] and to parallelize it to be able to use multi-core architectures.

Acknowledgment. The author would like to thank Marijn Heule for providing an implementation of HTE and discussing the dependencies of the algorithms.

References

1. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning sat solvers. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press (2010)
2. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (1971)
3. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)

4. Heule, M., Järvisalo, M., Biere, A.: Clause elimination procedures for CNF formulas. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 357–371. Springer, Heidelberg (2010)
5. Heule, M.J.H., Järvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 201–215. Springer, Heidelberg (2011)
6. Järvisalo, M., Biere, A.: Reconstructing solutions after blocked clause elimination. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 340–345. Springer, Heidelberg (2010)
7. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 129–144. Springer, Heidelberg (2010)
8. Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability. In: ICTAI, pp. 105–110. IEEE Computer Society (2003)
9. Manthey, N.: Solver Submission of riss 1.0 to the SAT Competition 2011. Tech. Rep. 1, Knowledge Representation and Reasoning Group, TU Dresden, Dresden, Germany (2011)
10. Manthey, N., Steinke, P.: Quadratic Direct Encoding vs. Linear Order Encoding. Tech. rep, Knowledge Representation and Reasoning Group, TU Dresden, Dresden, Germany (2011)
11. Sörensson, N.: Minisat 2.2 and minisat++ 1.1. http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf (2010)
12. Piette, C., Hamadi, Y., Saïs, L.: Vivifying propositional clausal formulae. In: ECAI, pp. 525–529. IOS Press (2008)

The SeaLion has Landed: An IDE for Answer-Set Programming—Preliminary Report

Johannes Oetsch, Jörg Pührer[1(✉)], and Hans Tompits

Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, 1040 Vienna, Austria
{oetsch, puehrer, tompits}@kr.tuwien.ac.at

Abstract. We report about the current state and designated features of the tool SeaLion, aimed to serve as an integrated development environment (IDE) for answer-set programming (ASP). A main goal of SeaLion is to provide a user-friendly environment for supporting a developer to write, evaluate, debug, and test answer-set programs. To this end, new support techniques have to be developed that suit the requirements of the answer-set semantics and meet the constraints of practical applicability. In this respect, SeaLion benefits from the research results of a project on methods and methodologies for answer-set program development in whose context SeaLion is realised. Currently, the tool provides source-code editors for the languages of Gringo and DLV that offer syntax highlighting, syntax checking, refactoring functionality, and a visual program outline. Further implemented features are a documentation generator, support for external solvers, and visualisation as well as visual editing of answer sets. SeaLion comes as a plugin of the popular Eclipse platform and provides itself interfaces for future extensions of the IDE.

1 Introduction

Answer-set programming (ASP) is a well-known and fully declarative problem-solving paradigm based on the idea that solutions to computational problems are represented in terms of logic programs such that the models of the latter, referred to as their *answer sets*, provide the solutions of a problem instance (for an overview about ASP, we refer to a survey article by Gelfond and Leone [1] or to the well-known textbook by Baral [2]). In recent years, the expressibility of languages supported by answer-set solvers increased significantly [3]. As well, ASP solvers have become much more efficient; e.g., the solver Clasp proved to be competitive with state-of-the-art SAT solvers [4].

Despite these improvements in solver technology, a lack of suitable *engineering tools* for developing programs is still a handicap for ASP towards gaining widespread popularity as a problem-solving paradigm. This issue is clearly

This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

recognised in the ASP community, and work to fill this gap has started recently, addressing issues like debugging, testing, and the modularity of programs [5–13]. Additionally, in order to facilitate tool support as known for other programming languages, attempts to provide *integrated development environments* (IDEs) have been put forth. Work in this direction includes the systems `APE` [14], `ASPIDE` [15], and `iGROM` [16].

Following this endeavour, in this paper, we describe the current status and designated features of a further IDE, `SeaLion`, developed as part of an ongoing research project on methods and methodologies for developing answer-set programs [17].

`SeaLion` is designed as an Eclipse plugin, providing useful and intuitive features for ASP. Besides experts, the target audience for `SeaLion` are software developers new to ASP yet who are familiar with support tools as used in procedural and object-oriented programming. Our goal is to fully support the languages of the current state-of-the-art solvers `Clasp` (in conjunction with `Gringo`) [3,18] and `DLV` [19], which distinguishes `SeaLion` from the other IDEs mentioned above which support only a single solver. Indeed, `APE` [14], which is also an Eclipse plugin, supports only the language of `Lparse` [20] that is a subset of the language of `Gringo`, whilst `ASPIDE` [15], a recently developed standalone IDE, offers support for `DLV` programs only. Although `iGROM` provides basic functionality for the languages of both `Lparse` and `DLV` [16], it currently does not support the latest version of `DLV` or the full syntax of `Gringo`.

At present, `SeaLion` is in a beta version that implements important core functionality and some advanced features. In particular, the languages of `DLV` and `Gringo` are supported to a large extent. The individual parsers translate programs and answer sets to data structures that are part of a rich and flexible framework for internally representing program elements. Based on these structures, the editor provides syntax highlighting, syntax checks, error reporting, error highlighting, and automatic generation of a program outline. A handy implemented refactoring feature allows for uniform and safe renaming of predicates and terms throughout a program and even across multiple files. There is functionality to manage external tools such as answer-set solvers and to define arbitrary pipes between them (as needed when using separate grounders and solvers). Moreover, in order to run an answer-set solver on the created programs, launch configurations can be created in which the user can choose input files, a solver configuration, command line arguments for the solver, as well as output-processing strategies. Answer sets resulting from a launch can either be parsed and stored in a view for interpretations, or the solver output can be displayed unmodified in Eclipse's built-in console view.

Another key feature of `SeaLion` is the capability for the *visualisation* and *visual editing* of interpretations. This follows ideas from the visualisation tools `ASPVIZ` [21] and `IDPDraw` [22], where a visualisation program Π_V (itself being an answer-set program) is joined with an interpretation I that shall be visualised. Subsequently, the overall program is evaluated using an answer-set solver, and the visualisation is generated from a resulting answer set. However, the editing feature of SeaLion allows also to graphically manipulate the interpretations under consideration which is neither supported by ASPVIZ nor by IDPDraw. The visualisation functionality of SeaLion is itself represented as an Eclipse plugin, called Kara.[1] In this paper, however, we describe only the basic functionality of Kara; a full description is given in a companion paper [23].

SeaLion integrates the documentation generator ASPDoc for ASP that is based on LANA (Language for ANnotating Answer-set programs), an annotation language for structuring, documenting, and testing answer-set programs [24].

The remainder of the paper is outlined as follows. In the next section, we shortly review the ASP solver languages supported by SeaLion. We discuss the general structure of the IDE, design choices regarding the implementation, as well as how to obtain SeaLion in Sect. 3. Section 4 gives an overview about features that are already functional in SeaLion, whereas Sect. 5 provides an outlook over functionality that is planned to be integrated in the future. In Sect. 6, we discuss other systems related to SeaLion, and we conclude in Sect. 7.

2 Supported ASP Languages

As we focus on supporting ASP developers, we deal with concrete solver languages and refer the reader to the textbook by Baral [2] for a formal introduction to ASP.

SeaLion offers support for the two major ASP solver language families, viz. the input language of the grounding tool Gringo that extends the one of the Lparse grounder and the language of the DLV solver. Both share a common basic Prolog-style rule syntax. In brief, an answer-set program consists of rules of the form

$$a_1 \mid \cdots \mid a_l \ :- \ a_{l+1}, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_n,$$

where $n \geq m \geq l \geq 0$, "not" denotes *default negation*, all a_i are first-order literals (i.e., atoms possibly preceded by the *strong negation* symbol "$-$"), and "$|$" is the disjunction symbol (DLV additionally allows for denoting disjunction by the letter "v"). For a rule r as above, the expression left to the symbol "$:-$" is the *head* of r and the expression to the right of "$:-$" is the *body* of r. If $n = l$, r is a *fact*; if r contains no disjunction, r is *normal*; and if $l = 0$ and $n > 0$, r is a *constraint*. For facts, the symbol "$:-$" is usually omitted.

Despite the common basic rule syntax, the languages of Gringo and DLV differ substantially when it comes to extended features. For one thing, aggregation in Gringo is realised by weight constraint literals that assign weights to literals such that the sum of the weights of true literals must lie between given bounds. For example, consider the weight literal

```
2 [a=1, b=1, c=3] 4,
```

[1] The name derives, with all due respect, from "Kara Zor-El", the native Kryptonian name of *Supergirl*, given that Kryptonians have visual superpowers on Earth.

assigning atoms `a` and `b` weight 1 and atom `c` weight 3. The weight literal is true when the sum of the weights of true atoms is between 2 and 4, i.e., when `a` and `b` are true but `c` is not, or if `c` and at most one of `a` and `b` are true. Aggregates in `DLV`, on the other side, are based on functions over so-called *symbolic sets* that are pairs of (a list of) variables and a conjunction of literals in which these variables appear. For example, the aggregate

```
2 <= #sum{X : a(X)} <= 4
```

is true if the sum of all (integer) constants `c` such that `a(c)` is true is between 2 and 4. Hence, the aggregate is, e.g., true if `a(1)` and `a(3)` but no other atom of predicate `a` is true. As can be seen from the example, `DLV` aggregates require the use of variables but `Gringo` weight constraint literals assign weights to ground literals. Variables in weight constraints are handled using so-called *conditions* that can also be used for ordinary literals in `Gringo` but are not available in `DLV`. For example, during grounding, the literal `redEdge(X,Y):edge(X,Y):red(X):red(Y)` in the body of a rule is replaced by the list of all literals `redEdge(n1,n2)`, where `edge(n1,n2)`, `red(n1)`, and `red(n2)` can be derived.

Further syntactic differences between the languages of `Gringo` and `DLV` are related to finding optimal answer sets. `DLV` uses special rules, called *weak constraints*, for optimisation, while `Gringo` uses minimise and maximise statements. While filtering atoms in the output can be done by hide and show statements in the case of `Gringo`, command-line arguments are needed in the case of `DLV`. For a more detailed description of the solver languages, we refer to the respective user manuals [25,26].

3 Implementation Principles, Architecture, and Availability

A key aspect in the design of `SeaLion` is extensibility. That is, on the one hand, we want to have enough flexibility to handle further ASP languages such that previous features can deal with them with no or only little adaption. On the other hand, we want to provide a powerful API framework that can be used by future features. To this end, we defined a hierarchy of classes and interfaces that represent *program elements*, i.e., fragments of ASP languages. This is done in a way such that we can use common interfaces and base classes for representing similar program elements of different ASP languages. For instance, we have different classes for representing literals of the `Gringo` language and literals of the `DLV` language in order to be able to handle subtle differences. For example, as `DLV` is unaware of conditions, an object of class `DLVStandardLiteral` has no support for them, whereas a `GringoStandardLiteral` object keeps a list of condition literals. Substantial differences in other language features, like aggregates, optimisation, and filtering support, are also reflected by different classes for `Gringo` and `DLV`, respectively. However, whenever possible, these classes are

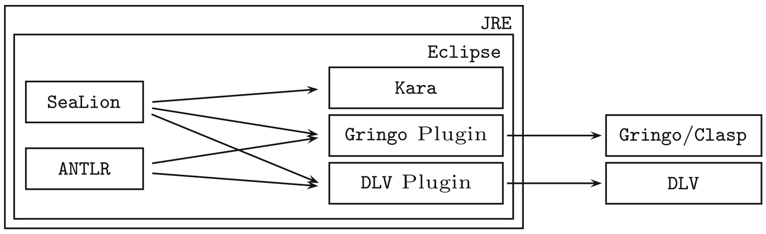


Fig. 1. Technology stack of SeaLion. An arrow indicates that a module is required by another.

derived from a common base class or share common interfaces. Therefore, plugins can, for example, use a general interface for aggregate literals to refer to aggregates of both languages. Hence, current and future feature implementations can make use of high-level interfaces and stay independent of the concrete ASP language to a large extent.

Also, within the SeaLion implementation, the aim is to have independent modules for different features, in form of Eclipse plugins, that ensure a well-structured code. Currently, there are the following plugins: the main plugin, a plugin that adapts the ANTLR parsing framework [27] to our needs, two solver plugins, one for supporting Gringo/Clasp and one for DLV, and the Kara plugin for answer-set visualisation and visual editing. Figure 1 depicts the technology stack of SeaLion, illustrating the embedding in Eclipse and the Java Runtime Environment (JRE), the aforementioned plugins, as well as the use of answer-set solvers as external applications.

It is a key aim to smoothly integrate SeaLion in the Eclipse platform and to make use of functionality the latter provides wherever suitable. The motivation is to exploit the rich platform as well as to ensure compatibility with upcoming versions of Eclipse.

The decision to build on Eclipse, rather than writing a stand-alone application from scratch, has many benefits. For one, we profit from software reuse as we can make use of the general GUI of Eclipse and just have to adapt existing functionality to our needs. Examples include the text editor framework, source-code annotations, problem reporting and quick fixes, project management, the undo-redo mechanism, the console view, the refactoring and the navigation framework (Outline, Project Explorer), and launch configurations. Moreover, much functionality of Eclipse can be used without any adaptions, e.g., workspace management, the possibility to define working sets, i.e., grouping arbitrary files and resources together, software versioning and revision control (e.g., based on SVN or CVS), and task management. Another clear benefit is the popularity of Eclipse among software developers, as it is a widely used standard tool for developing Java applications. Arguably, people who are familiar with Eclipse and basic ASP skills will easily adapt to SeaLion. Finally, choosing Eclipse for an IDE for ASP offers a chance for integration of development tools for hybrid languages, i.e., combinations of ASP and procedural languages. For instance, Gringo supports the use of functions written in the LUA scripting language [28]. As there is a LUA plugin for Eclipse available, one can at least use that in parallel with SeaLion. However, there is also potential for a tighter integration of the two plugins.

SeaLion is free software published under the GNU General Public License version 3. For more information on SeaLion and installation instructions we refer to the project web site

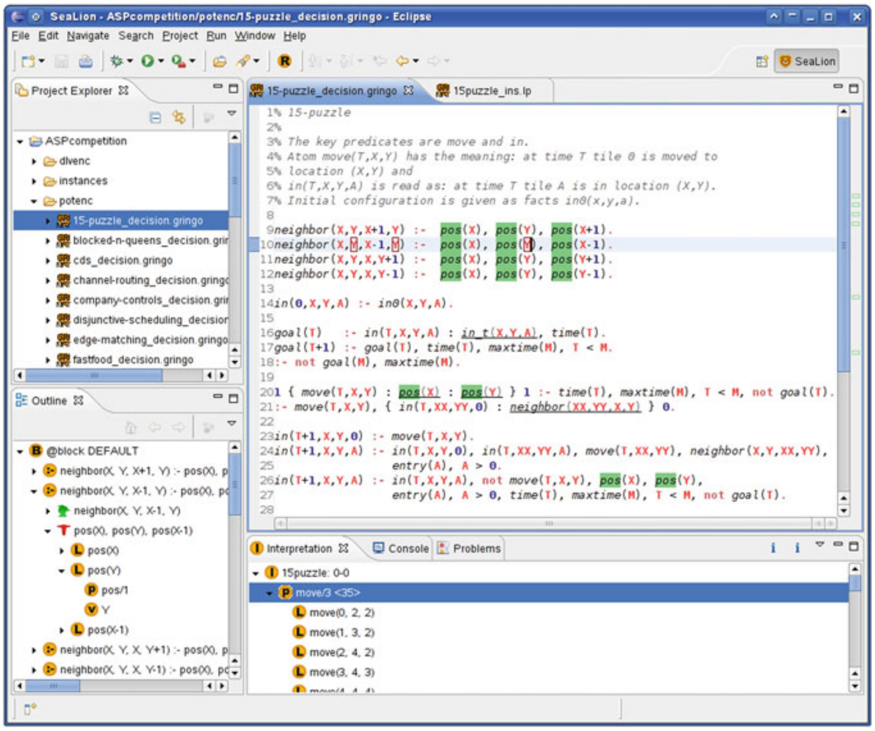
http://www.sealion.at.

4 Current Features

In this section, we describe the features that are already operational in SeaLion, including technical details on the implementation.

4.1 Source-Code Editor

The central element in SeaLion is the *source-code editor* for logic programs. For now, it comes in two variations, one for DLV and one for Gringo. A screenshot of a Gringo source file in SeaLion's editor is given in Fig. 2. By default, files


SeaLion - ASPcompetition/potenc/15-puzzle_decision.gringo - Eclipse
File Edit Navigate Search Project Run Window Help
SeaLion
Project Explorer
ASPcompetition
dlvenc
instances
potenc
15-puzzle_decision.gringo
blocked-n-queens_decision.grin
cds_decision.gringo
channel-routing_decision.gringo
company-controls_decision.grin
disjunctive-scheduling_decision
edge-matching_decision.gringo
fastfood_decision.gringo
Outline
@block DEFAULT
neighbor(X, Y, X+1, Y) :- pos(X), p
neighbor(X, Y, X-1, Y) :- pos(X), pc
neighbor(X, Y, X-1, Y)
pos(X), pos(Y), pos(X-1)
pos(X)
pos(Y)
pos/1
Y
pos(X-1)
neighbor(X, Y, X, Y+1) :- pos(X), p
neighbor(X, Y, X, Y-1) :- pos(X), pc
15-puzzle_decision.gringo
15puzzle_ins.lp
Interpretation
Console
Problems

```
% 15-puzzle
%
% The key predicates are move and in.
% Atom move(T,X,Y) has the meaning: at time T tile 0 is moved to
% location (X,Y) and
% in(T,X,Y,A) is read as: at time T tile A is in location (X,Y).
% Initial configuration is given as facts in0(x,y,a).

neighbor(X,Y,X+1,Y) :-  pos(X), pos(Y), pos(X+1).
neighbor(X,Y,X-1,Y) :-  pos(X), pos(Y), pos(X-1).
neighbor(X,Y,X,Y+1) :-  pos(X), pos(Y), pos(Y+1).
neighbor(X,Y,X,Y-1) :-  pos(X), pos(Y), pos(Y-1).

in(0,X,Y,A) :- in0(X,Y,A).

goal(T)   :- in(T,X,Y,A) : in_t(X,Y,A), time(T).
goal(T+1) :- goal(T), time(T), maxtime(M), T < M.
:- not goal(M), maxtime(M).

1 { move(T,X,Y) : pos(X) : pos(Y) } 1 :- time(T), maxtime(M), T < M, not goal(T).
:- move(T,X,Y), { in(T,XX,YY,0) : neighbor(XX,YY,X,Y) } 0.

in(T+1,X,Y,0) :- move(T,X,Y).
in(T+1,X,Y,A) :- in(T,X,Y,0), in(T,XX,YY,A), move(T,XX,YY), neighbor(X,Y,XX,YY),
                 entry(A), A > 0.
in(T+1,X,Y,A) :- in(T,X,Y,A), not move(T,X,Y), pos(X), pos(Y),
                 entry(A), A > 0, time(T), maxtime(M), T < M, not goal(T).
```

```
15puzzle: 0-0
  move/3 <35>
    move(0, 2, 2)
    move(1, 3, 2)
    move(2, 4, 2)
    move(3, 4, 3)
```

Fig. 2. A screenshot of SeaLion's editor, the program outline, and the interpretation view.

with names ending in "`.lp`", "`.lparse`", "`.gr`", or "`.gringo`" are opened in the `Gringo` editor, whereas files with extensions "`.dlv`" or "`.dl`" are opened in the `DLV` editor. Nevertheless, any file can be opened in either editor if required.

The editors provide *syntax highlighting*, which is computed in two phases. Initially, a fast syntactic check provides initial colouring and styling for comments and common tokens like dots concluding rules and the rule implication symbol. While editing the source code, after a few moments of user inactivity, the source code is parsed and data structures representing the program are computed and stored for various purposes. The second phase of syntax highlighting is already based on this program representation and allows for fine-grained highlighting depending not only on the type of the program element but also on its role. For instance, a literal that is used in the condition of another literal is highlighted in a different way than stand-alone literals.

The parsers used are based on the ANTLR framework [27] and are in some respect more lenient than the respective solver parsers. For one thing, they are more tolerant towards syntax errors. For instance, in many cases they accept terms of various types (constants, variables, aggregate terms) where a solver requires a particular type, like a variable. The errors will still be noticed, during building the program representation or afterwards, by means of explicit checks. This tolerance allows for more specific warning and error reporting than provided by the solvers. For example, the system can warn a user that a constant was used on the left-hand side of an assignment where only a variable is allowed. Another parsing difference is the handling of comments. The parser does not throw them away but collects them and associates them to the program elements in their immediate neighbourhood. One benefit is that the information contained in comments can be kept when performing automatic transformations on the program, like rule reorderings or translations to other logic programming dialects. Another advantage is that we can make use of comments for enriching the language with our own *meta statements* that do not interfere with the solver when running the file. We reserved the token "`\%!`" for initiating single-line meta commands and "`\%*!`" and "`*\%`" for the start and end of block meta commands in the `Gringo` editor, respectively. Currently, meta commands are used for assigning properties to program elements.

Example 1. In the following source code, a meta statement assigns the name "`r1`" to the rule it precedes.

```
%! name = r1;
a(X) :- c(X).
```

These names are currently used in an ancillary application of `SeaLion` for reifying disjunctive non-ground programs as used in a previous debugging approach [10]. Moreover, names assigned to program elements as above can be seen in Eclipse's *Outline View*. `SeaLion` uses this view to give an overview of the edited program in a tree-shaped graphical representation. The rules of the programs are represented by nodes of this tree. By expanding the descendant nodes of an individual rule node, one can see its elements, i.e., head, body, literals, predicates, terms, etc.

(cf. Fig. 2). Clicking on such an element selects the corresponding program code in the editor, and the programmer can proceed editing there. A similar outline is also available in Eclipse's "Project Explorer" as subtree under the program's source file.

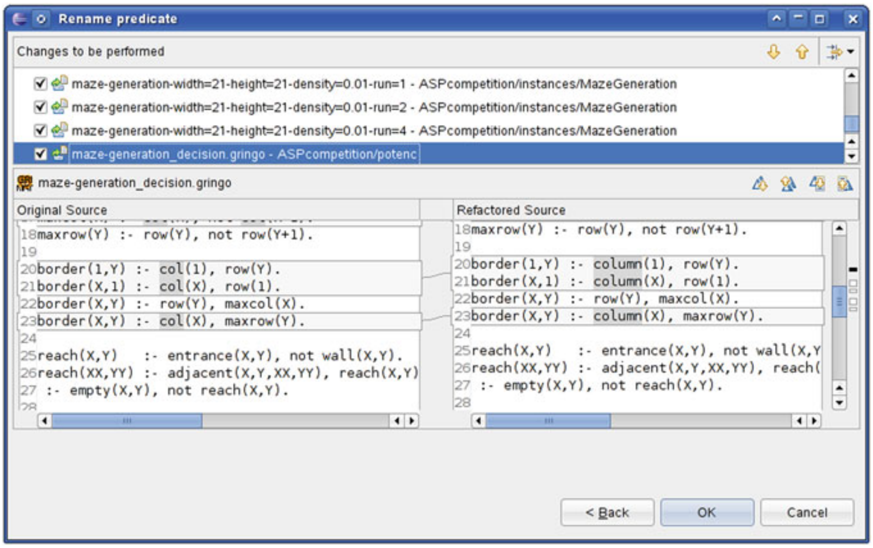
Another feature of the editor is the support for *Eclipse annotations*. These are means to temporarily highlight parts of the source code. For instance, SeaLion annotates occurrences of the program element under the text cursor. If the cursor is positioned over a literal, all literals of the same predicate are highlighted in the text and in a bar next to the vertical scrollbar that indicates the positions of all occurrences in the overall document. Likewise, when a constant or a variable in a rule is on the cursor position, their occurrences are detected within the whole source code or within the rule, respectively.

A particular application of Eclipse annotations is *problem reporting*. Syntax errors and warnings are displayed in two ways. First, they are marked in the source code with a zig-zag styled underline. Second, they are displayed in Eclipse's "Problem View" that collects various kinds of problems and allows for directly jumping to the problematic source code region upon a mouse click.

SeaLion offers initial functionality for *refactoring* answer-set programs. Refactoring is the process of improving the source code of a program, e.g., by enhancing its structure, reusability, or readability, without changing its external behaviour. In particular, we implemented functionality for uniform and safe renaming of predicates, constants, function symbols, and variables throughout a user-defined set of files containing answer-set programs. To initiate renaming, the user can either select the targeted program element in the Outline View or place the cursor on it within the editor and use the menu or a keyboard shortcut to open the renaming dialog. On the dialog's first page, the user can specify the new name for the program element and select the files in which renaming should take place. When renaming variables, however, the latter choice is not available because variables are only renamed within a rule and therefore within the same file in which the chosen variable appears. The motivation is that two variables with the same identifier in different rules often have a different meaning. The renaming dialog warns the user if the new name of the program element is already in use anywhere else in the selected programs. As such a renaming still could be intended, it is possible to perform renaming, nevertheless. Once the new name is chosen, the user has the possibility to directly apply the changes implied by renaming or revise them on a preview page. Here, one can inspect the effects file by file where the original as well as the new source code are displayed next to each other and all hypothetical changes are highlighted as depicted in Fig. 3.

4.2 Documentation Feature

SeaLion allows for automatically generating source code documentation for answer-set programs, similar as tools like JavaDoc or Doxygen do for other programming languages. For this purpose, the IDE incorporates the ASPDoc documentation generator, a recently developed tool that takes annotated ASP code as input and produces HTML files as output, based on the LANA annotation



☑ maze-generation-width=21-height=21-density=0.01-run=1 - ASPcompetition/instances/MazeGeneration
☑ maze-generation-width=21-height=21-density=0.01-run=2 - ASPcompetition/instances/MazeGeneration
☑ maze-generation-width=21-height=21-density=0.01-run=4 - ASPcompetition/instances/MazeGeneration
☑ maze-generation_decision.gringo - ASPcompetition/potenc

maze-generation_decision.gringo

Original Source

```
maxrow(Y) :- row(Y), not row(Y+1).

border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxcol(X).
border(X,Y) :- col(X), maxrow(Y).

reach(X,Y)   :- entrance(X,Y), not wall(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y)
 :- empty(X,Y), not reach(X,Y).
```

Refactored Source

```
maxrow(Y) :- row(Y), not row(Y+1).

border(1,Y) :- column(1), row(Y).
border(X,1) :- column(X), row(1).
border(X,Y) :- row(Y), maxcol(X).
border(X,Y) :- column(X), maxrow(Y).

reach(X,Y)   :- entrance(X,Y), not wall(X,Y
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(
 :- empty(X,Y), not reach(X,Y).
```

Fig. 3. Reviewing file changes implied by renaming predicate `col/2` to `column/2`.

language [24]. Lana is designed to support the development of answer-set programs even beyond documentation, allowing to group rules into coherent blocks and to specify language signatures, types, pre- and postconditions, as well as unit tests for such blocks. Similar to meta commands in `SeaLion`, these annotations are invisible to an ASP solver since they have the form of program comments, but they can be interpreted by specialised support tools, e.g., for testing and verification purposes or for eliminating sources of common programmer errors by realising syntax checking or code-completion features. The following example code demonstrates Lana annotations for grouping ASP code into blocks and describing predicates and their arguments using `@atom` and `@term` tags of Lana:

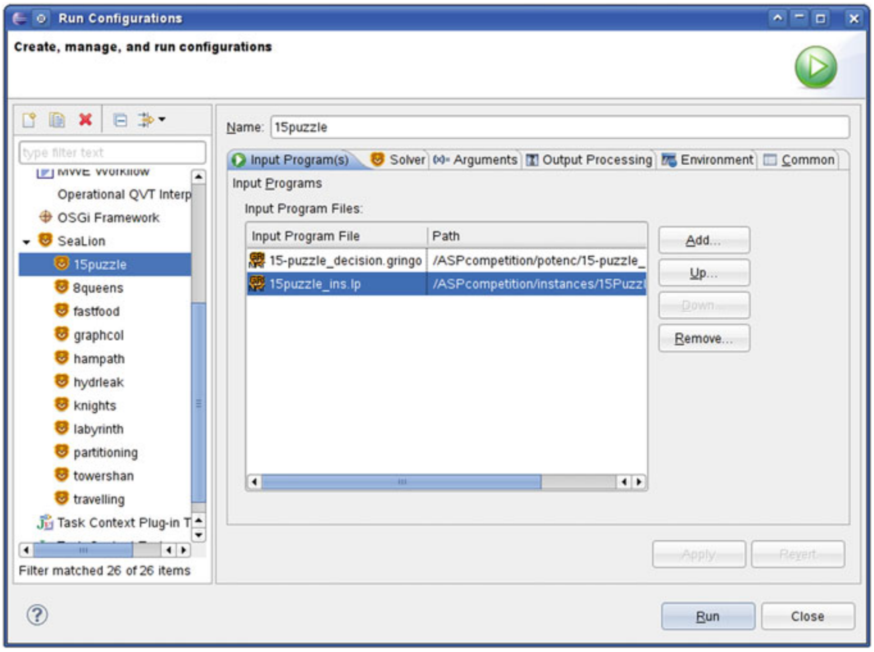
```
%* @block maze {
%*  This is the main block of the maze generation program.
%*  @atom entrance(R,C) gives the position of the maze entrance
%*  @term R is a row index
%*    @with 0 < R, R < 20
%*  @term C is a column index
%*    @with 0 < R, R < 20
%*  ...
    empty(R,C) | wall(R,C) :- row(R),col(C).
    ...
%* }
```

ASP documentation generation can be accessed through Eclipse's export menu. After selecting the ASP programs that should be documented and a target directory, different HTML files are created with `index.html` as the entry point as usual. The documentation contains descriptions of all blocks of the answer-set program, where sub-blocks are indented with respect to their parent blocks. Also, a summary of the block structure of the entire answer-set program is presented at the beginning of the documentation to provide an overview. For each block, descriptions of the used atoms and types of involved terms, as well as for pre- and postconditions are given. By default, hidden atoms, i.e., atoms never mentioned in a blocks input nor in its output signature, are displayed if the user does not decide otherwise. The documentation also includes HTML versions of the programs' source code, which can be particularly useful for sharing ASP code online. There are links from the documentation to the source code and vice versa. Likewise, rules for defining pre- and postconditions can be inspected by using respective links.

`SeaLion` can already parse ASP code annotated by the full Lana language. Besides the already implemented documentation functionality, it is planned to integrate further features based on Lana annotations as described in Sect. 5.

4.3 Support for External Tools

In order to interact with solvers and grounders from `SeaLion`, we implemented a mechanism for handling external tools. One can define *external tool configurations* that specify the path to an executable as well as default command-line



Name: 15puzzle

Input Program File	Path
15-puzzle_decision.gringo	/ASPcompetition/potenc/15-puzzle_
15puzzle_ins.lp	/ASPcompetition/instances/15Puzzl

Fig. 4. Selecting two source files for ASP solving in Eclipse's launch configuration dialog.

parameters. Arbitrary command-line tools are supported; however, there are special configuration types for some programs such as Gringo, Clasp, and DLV. For these, it is planned to have a specialised GUI that allows for a more convenient modification of command-line parameters. In addition to external command-line tools, one can also define tool configurations that represent pipes between external tools. This is needed when grounding and solving are provided by separate executables. For instance, one can define two separate tool configurations for Gringo and Clasp and define a piped tool configuration for using the two tools in a pipe. Pipes of arbitrary length are supported such that arbitrary pre- and post-processing can be done when needed. Default solvers for different languages can be set in the preference menu of SeaLion depending on file content types in the "Content Type Preferences" section.

For executing answer-set solvers, we make use of Eclipse's *launch configuration framework*. In our setting, a launch configuration defines which programs should be executed using which solver. Figure 4 shows the page of the launch configuration editor on which input files for a solver invocation can be selected.

Besides using the standard command-line parameters from the tool configurations, also customised parameters can be set for the individual program launches. Moreover, a launch configuration contains information how the output of the solver should be treated. One option is to print the solver output as it is in Eclipse's *console view*. The other option is to parse the resulting answer sets for further use in SeaLion. In this case, the user can specify the format in which the answer sets are expected from the solver (as there is no standardised form for displaying answer sets). Here, default strategies are preselected, depending on the chosen solver configuration.

Besides defining launch configurations, SeaLion also offers the possibility to invoke a solver right away on a selection of files in the workspace using the default settings of an external tool configuration. This is realised using the so-called *Launch Shortcuts* mechanism of Eclipse. The user selects the files that should be evaluated in the project explorer and select the SeaLion entry of their "Run As" context menu. The entry is available as soon as an external tool configuration is set as default solver for the selected file content type.

4.4 Interpretation Views

When the user decides to parse answer sets obtained from the solvers, they are stored in SeaLion's *interpretation view* as well as the *interpretation compare view* that is depicted in Fig. 5. In both, interpretations are visualised as expandable trees of depth 3. The root node is the interpretation (marked by an "I") and its children are the predicates (marked by a "p") appearing in the interpretation. Finally, each of these predicates is the parent node of the literals over the predicate that are contained in the interpretation (marked by an "L"). Compared to a standard textual representation, this way of visualising answer sets provides a well-arranged overview of the individual interpretations. We find it also more appealing than a tabular representation where only entries for a single predicate are visible at once. While the interpretation view lists interpretations

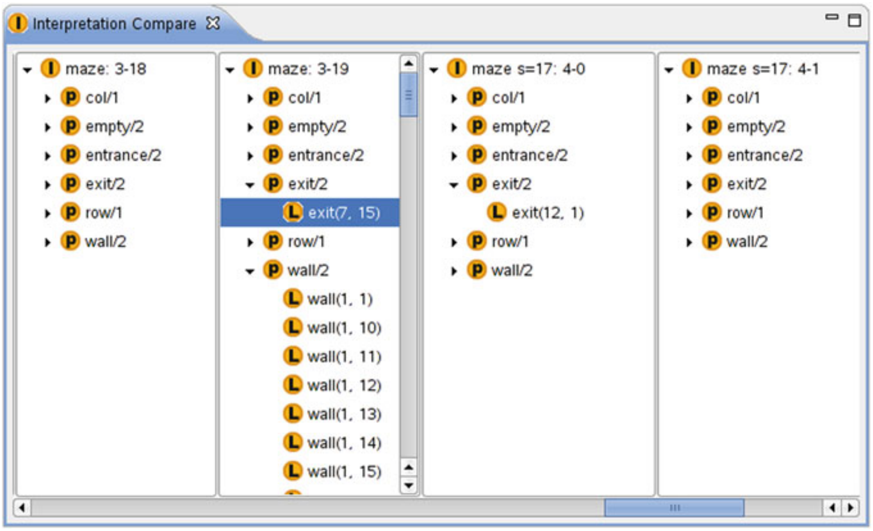


Fig. 5. `SeaLion`'s interpretation compare view.

in rows, the interpretation compare view places them in columns. By horizontally arranging trees for different interpretations next to each other, it is easy to compare two or more interpretations.

The two interpretation views are not only meant to provide a good visualisation of results but also serve as a starting point for ASP developing tools that depend on interpretations. One convenient feature is dragging interpretations or individual literals from the interpretation views and dropping them on the source-code editor. When released, these are transformed into facts of the respective ASP language.

4.5 Visualisation and Visual Editing

The plugin `Kara` [23] is a tool for the graphical visualisation and editing of interpretations. It is started from the interpretation view. One can select an interpretation for visualisation by right-clicking it in the view and choosing between a *generic visualisation* or a *customised visualisation*. The latter is specified by the user by means of a visualisation answer-set program. The former represents the interpretation as a labelled hypergraph.

In the generic visualisation, the nodes of the hypergraph are the individuals appearing in the interpretation. Each edge represents a literal in the interpretation, connecting the individuals appearing in the respective literal. Integer labels on the endings of an edge are used for expressing the argument position of the individual. In order to distinguish between different predicates, each edge has an additional label stating the predicate name. Moreover, edges of the same predicate are of the same colour. An example of a generic visualisation of a spanning

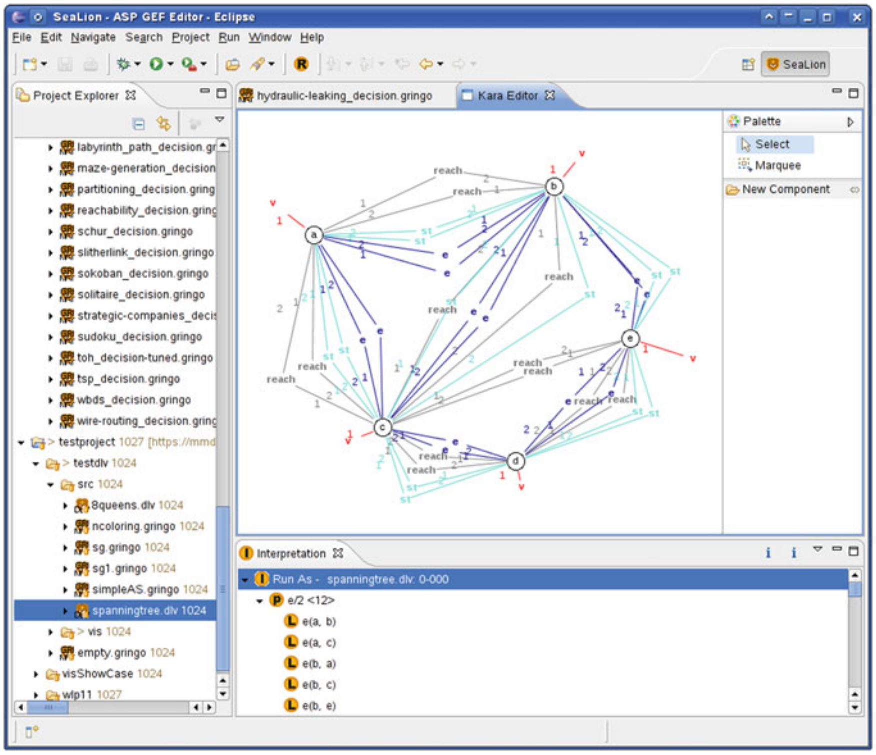
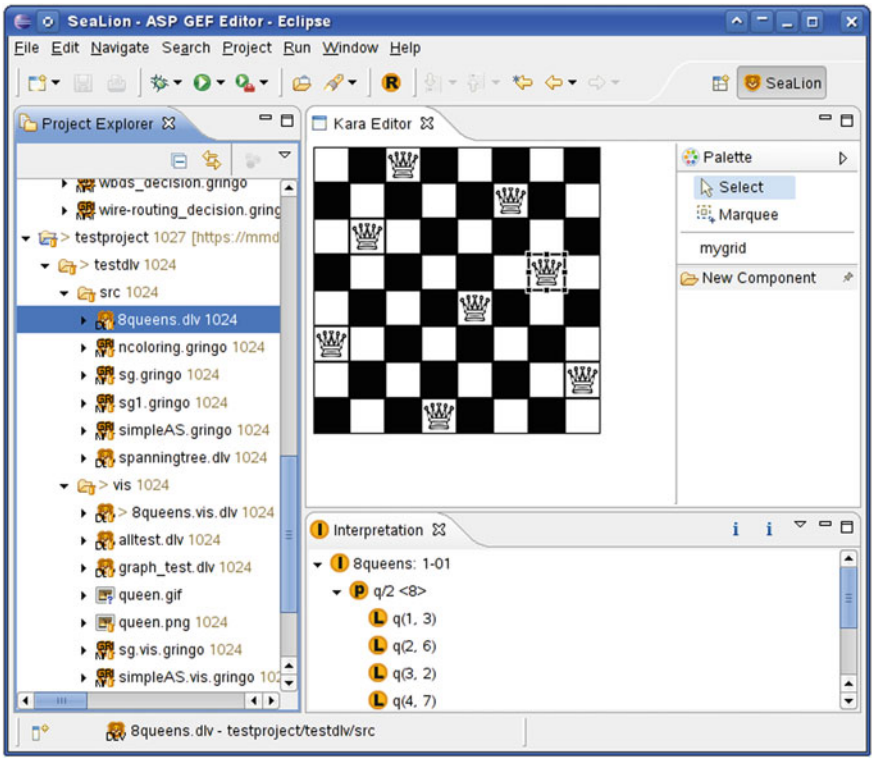


Fig. 6. A screenshot of SeaLion's visual interpretation editor.

tree interpretation is shown in Fig. 6 (the layout of the graph has been manually optimised in the editor).

The customised visualisation feature allows for specifying how the interpretation should be illustrated by means of an answer-set program that uses a powerful pre-defined visualisation vocabulary. The approach follows the ideas of ASPVIZ [21] and IDPDraw [22]: a visualisation program Π_V is joined with the interpretation I to be visualised (technically, I is considered to be a set of facts) and evaluated using an answer-set solver. One of the resulting answer sets is then interpreted by SeaLion for building the graphical representation of I. The vocabulary allows for using and positioning basic graphical elements such as lines, rectangles, polygons, labels, and images, as well as graphs and grids composed of such elements.

The resulting visual representation of an interpretation is shown in a graphical editor that also allows for manipulating the visualisation in many ways. Properties such as colours, IDs, and labels can be manipulated and graphical elements can be repositioned, deleted, or even created. Such manipulations are useful for two different purposes. First, for fine-tuning the visualisation before saving it as a scalable vector graphic (SVG) by means of our SVG export



```
8queens: 1-01
  q/2 <8>
    q(1, 3)
    q(2, 6)
    q(3, 2)
    q(4, 7)
```

Fig. 7. A customised visualisation of an 8-queens instance.

functionality. Second, modifying the visualisation can be used to obtain a modified version I' of the visualised interpretation I by abductive reasoning. In fact, we implemented a feature that allows for abducing an interpretation that would result in the modified visualisation. Modifications in the visual editor are automatically reflected in an adapted version I'_V of the answer set I_V representing the visualisation. We then construct an answer-set program $\lambda(I'_V, \Pi_V)$, depending on the modified visualisation answer set I'_V and the visualisation program Π_V, for obtaining the modified interpretation I' as a projected answer set of $\lambda(I'_V, \Pi_V)$. For more details, we refer to a companion paper [23]. An example for a customised visualisation for a solution to the 8-queens problem is given in Fig. 7.

5 Projected Features

In the following, we give an overview of further functionality that we plan to incorporate into SeaLion in the near future.

One core feature that is already under development is the support for *stepping-based debugging* of answer-set programs as introduced in recent work

[29]. Here, we aim for an intuitive and easy-to-handle user interface, which is clearly a challenge to achieve for reasons intrinsic to ASP. In particular, the discrepancy between developing programs at the non-ground level and obtaining solutions based on their groundings makes the realisation of practical debugging tools for ASP non-trivial.

As mentioned earlier, our goal is to develop more `SeaLion` features, besides the already implemented documentation generator, exploiting Lana annotations in answer-set programs. Here, one point is that we want to enrich `SeaLion` with support for *typed predicates* which can be specified using Lana. That is, the user can define the domain for a predicate. For instance, consider the predicate `age/2` stating the age of a person. Then, with typing, we can express that for every atom `age(p,a)`, the term `p` represents an element from a set of persons, whereas `a` represents an integer value. Two types of domain specifications will be supported, viz. direct ones, which explicitly state the names of the individuals of the domain, and indirect ones that allow for specifications in terms of the domain of other predicates. We expect multiple benefits from having this kind of information available. First, it is useful as a documentation of the source code. A programmer can clearly specify the intended meaning of a predicate and look it up in the type specifications. Moreover, type violations in the source code of the program can be automatically detected as illustrated by the following example.

Example 2. Assume we want to define a rule deriving atoms with predicate symbol `serves/3`, where `serves(R,D,P)` expresses that restaurant `R` serves dish `D` at price `P`. Furthermore, the two predicates `dishAvailable/2` and `price/3` state which dishes are currently available in which restaurants and the price of a dish in a restaurant, respectively. Moreover, assume we have type specifications stating that for `serves(R,D,P)` and `dishAvailable(D,R)`, `R` is of type `restaurant` and `D` is of type `dish`. Then, a potential type violation in the rule

```
serves(R,D,P) :- dishAvailable(R,D),price(R,D,P)
```

could be detected. This way, the programmer would notice that the order of variables in `dishAvailable(R,D)` was mixed up.

In order to avoid problems like in the above example in the first place, auto-completion functionality could be implemented such that variables and constants of correct types are suggested when writing the arguments of a literal in a rule.

The annotation language Lana allows for combining the typing system with functionality that allows for defining *program signatures*. One application of such signatures is for specifying the predicates and terms used for abducing a modified interpretation I' in our plugin for graphically editing interpretations. Moreover, input and output signatures can be defined for uniform problem encodings, i.e., answer-set programs that expect a set of facts representing a problem instance as input such that its answer sets correspond to the solutions for this instance. Then, such signatures can be used in our planned support for *assertions* that will allow for automatically checking pre- and postconditions of answer-set programs that are defined in Lana. Having a full specification for the input of a program, i.e., a typed signature and input constraints in the form of preconditions, one can automatically generate input instances for the program and use them, e.g., for random testing [12,30]. Also, more advanced testing and verification functionality can be realised, like the automatic generation of valid input (with respect to the preconditions) that violates a postcondition.

In order to reduce the amount of time a programmer has to spend for writing type and signature definitions, we want to explore methods for partially extracting them from the source code or from interpretations.

Besides assertions, it is also planned to offer further testing techniques. In particular, we aim at a unit testing system by integrating the recently developed command-line testing tool `ASPUnit` [24]. The idea is to formulate test cases in the form of ASP programs with Lana annotations that contain information about the expected results under a given reasoning mode when the test-case program is joined with the units under test. Here, units are understood as blocks of ASP rules that are defined using Lana. Multiple test cases can be combined to test suites according to the user's needs. When a test suite is evaluated, `SeaLion` shall give information about what conditions in which test cases failed and, if possible, provide information why.

Other projected features include typical amenities of Eclipse editors like auto-completion, pretty-printing, further means for refactoring, and providing quick-fixes for typical problems in the source code. Also, checks for errors and warnings that are not already done by the parser, e.g., detection of unsafe variables, need still to be implemented.

We also want to provide different kinds of program translations in `SeaLion`. To this end, we already implemented a flexible framework for transforming program elements to string representations following different strategies. In particular, we aim at translations between different solver languages at the non-ground level. Here, we first have to investigate strategies when and how transformations of, e.g., aggregates, can be applied such that a corresponding overall semantics can be achieved. Other specific program translations that we consider for implementation would be necessary for realising the import and export of rules in the Rule Interchange Format (RIF) [31], which is a W3C recommendation for exchanging rules in the context of the Semantic Web. Notably, a RIF dialect for ASP, called RIF-CASPD, has been proposed [32].

Further convenience improvements for using external tools in `SeaLion` include a specialised GUI for choosing the command-line parameters. For launch configurations, we want to add the possibility to directly write the output of a tool invocation into a file and to allow for exporting the launch configuration as native stand-alone scripts.

Finally, there are many possible ways to enhance the GUI of `SeaLion`. We want to extend the support for drag-and-drop operations such that, e.g., program elements in the outline can be dragged into the editor. Moreover, we plan to realise sorting and filtering features for the outline and interpretation view.

Regarding interpretations, we aim for supporting textual editing of interpretations directly in the view, besides visual editing, and a feature for comparing multiple interpretations by highlighting their differences.

6 Related Work

We next give a short overview of existing IDEs for core ASP languages. To begin with, the tool APE [14], developed at the University of Bath, is also based on Eclipse. It supports the language of Lparse and provides syntax highlighting, syntax checking, program outline, and launch configuration. Additionally, APE has a feature to display the predicate dependency graph of a program.

ASPIDE, a recent IDE for DLV programs [15], is a standalone tool that already offers many features as it builds on previous tools [33–35]. Some functionality we want to incorporate in SeaLion is already supported by ASPIDE, e.g., code completion, unit tests [36], and quick fixes. Further features of ASPIDE are support for code templates and a visual program editor. We do not aim for comprehensive visual source-code editing in SeaLion but consider the use of program templates that allow for expressing common programming patterns. In their current releases, neither APE nor ASPIDE support graphical visualisation or visual editing of answer sets as available in SeaLion. ASPIDE allows for displaying answer sets in a tabular form. This is an improvement compared to the standard textual representation but comes with the drawback that only entries for a single predicate are visible at once. Besides the graphical representation, SeaLion can display interpretations in a dedicated view that gives a good overview of the individual interpretations and allows also to compare different interpretations.

Concerning supported ASP languages, SeaLion is the first IDE to support the language of Gringo rather than its Lparse subset. Moreover, other proposed IDEs for ASP do only consider the language of either DLV or Lparse, with the exception of iGROM [16] that provides basic syntax highlighting and syntax checking for the languages of both, Lparse and DLV. Note that iGROM has been developed at our department independently from SeaLion as a student project. A speciality of iGROM is the support for the front-end languages for planning and diagnosis of DLV. There also exist proprietary IDEs for ASP related languages with support for object-oriented features, OntoStudio and OntoDLV [37,38].

Compared to the other visualisation tools, ASPVIZ [21] and IDPDraw [22], our plugin Kara [23] allows not only for visualisation of an interpretation but also for visually editing the graphical representation such that changes are reflected in the visualised interpretation. Moreover, Kara offers support for generic visualisations, special support for grids, and automatic layout of graph structures. The latter is also the goal of Lonsdaleite, a recent tool for visualising graph structures encoded in answer-sets [39]. It is realised as a lightweight Python script that maps the atoms in an answer set to the input format of the Graphviz utilities [40].

7 Conclusion

In this paper, we presented the current status of `SeaLion`, an IDE for ASP languages that is currently under development. We discussed general principles that we follow in our implementation and gave an overview of current features. `SeaLion` is an Eclipse plugin and is designed to be the first comprehensive IDE that supports the languages of both `Gringo` and `DLV`, which can currently be considered as the two most prominent implemented ASP languages.

As this is an intermediate report, we also discussed which features we plan to incorporate in future work. The most important step in the advancement of the IDE is the integration of an easy-to-use debugging system that is currently under development. Moreover, we want to implement features for defining types, signatures, pre- and postconditions, and unit tests based on the Lana annotation language into `SeaLion`. One advantage of using Lana is that, in addition to the graphical tools of the IDE, development support can also be provided by respective command-line tools supporting Lana.

References

1. Gelfond, M., Leone, N.: Logic programming and knowledge representation - The A-Prolog perspective. Artif. Intell. **138**(1–2), 3–38 (2002)
2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
3. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver *clasp*: Progress report. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 509–514. Springer, Heidelberg (2009)
4. SAT 2011 competition. http://www.satcompetition.org
5. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proceedings of ASP 2005. http://CEUR-WS.org (2005)
6. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theor. Pract. Logic Program. **9**(1), 1–56 (2009)
7. Syrjänen, T.: Debugging inconsistent answer-set programs. In: Proceedings of NMR 2006, pp. 77–83. Technische Universität Clausthal (2006)
8. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
9. Wittocx, J., Vlaeminck, H., Denecker, M.: Debugging for model expansion. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 296–311. Springer, Heidelberg (2009)
10. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: on debugging non-ground answer-set programs. Theor. Pract. Logic Program. **10**(4–5), 513–529 (2010)
11. Niemelä, I., Janhunen, T., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proceedings of ECAI 2010, pp. 951–956. IOS Press (2010)
12. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: Random vs. structure-based testing of answer-set programs: An experimental comparison. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 242–247. Springer, Heidelberg (2011)

13. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J. Artif. Intell. Res. **35**, 813–857 (2009)
14. Sureshkumar, A., De Vos, M., Brain, M., Fitch, J.: APE: An AnsProlog* environment. In: Proceedings of SEA 2007, pp. 71–85 (2007)
15. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated development environment for answer set programming. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 317–330. Springer, Heidelberg (2011)
16. iGROM. http://igrom.sourceforge.net/
17. Oetsch, J., Pührer, J., Tompits, H.: Methods and methodologies for developing answer-set Programs—Project description. In: Technical Communications of ICLP 2010, pp. 154–161. Leibniz-Zentrum für Informatik (2010)
18. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
19. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3), 499–562 (2006)
20. Syrjänen, T.: Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
21. Cliffe, O., De Vos, M., Brain, M., Padget, J.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 724–728. Springer, Heidelberg (2008)
22. Wittocx, J.: KRR Software: IDPDraw. https://dtai.cs.kuleuven.be/krr/software/visualisation
23. Kloimüllner, C., Oetsch, J., Pührer, J., Tompits, H.: Kara: A system for visualising and visual editing of interpretations for answer-set programs. In: Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A. (eds.) INAP/WLP 2011. LNCS, vol. 7773, pp. 325–344. Springer, Heidelberg (2013)
24. De Vos, M., Kısa, D.G., Oetsch, J., Pührer, J., Tompits, H.: Annotating answer-set programs in LANA. Theor. Pract. Logic Program. **12**(4–5), 619–637 (2012)
25. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. http://sourceforge.net/projects/potassco/files/potassco_guide
26. Bihlmeyer, R., Faber, W., Ielpa, G., Lio, V., Pfeifer, G.: DLV user manual. http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html
27. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers. Pragmatic Bookshelf, Frisco (2007)
28. Ierusalimschy, R.: Programming in Lua, 2nd edn. Lua.Org (2006)
29. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an answer-set program. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 134–147. Springer, Heidelberg (2011)
30. Oetsch, J., Prischink, M., Pührer, J., Schwengerer, M., Tompits, H.: On the small-scope hypothesis for testing answer-set programs. In: Proceedings of KR 2012, pp. 43–53. AAAI Press (2012)
31. Boley, H., Kifer, M. (eds.): RIF framework for logic dialects. W3C (2010) W3C Recommendation 22 June 2010
32. Kifer, M., Heymans, S.: RIF core answer set programming dialect. http://ruleml.org/rif/RIF-CASPD.html (2009)
33. Febbraro, O., Reale, K., Ricca, F.: A visual interface for drawing ASP programs. In: Proceedings of CILC 2010 (2010)

34. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proceedings of SEA 2009 (2009)
35. Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: spock: A debugging support tool for logic programs under the answer-set semantics. In: Seipel, D., Hanus, M., Wolf, A. (eds.) INAP 2007. LNCS, vol. 5437, pp. 247–252. Springer, Heidelberg (2009)
36. Febbraro, O., Leone, N., Reale, K., Ricca, F.: Unit testing in *ASPIDE*. In: Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A. (eds.) INAP/WLP 2011. LNCS, vol. 7773, pp. 345–364. Springer, Heidelberg (2013)
37. ontoprise GmbH: OntoStudio 3.0. http://help.ontoprise.de/ (2010)
38. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: An ASP-based system for enterprise ontologies. J. Logic Comput. **19**(4), 643–670 (2008)
39. Smith, A.: Lonsdaleite. https://github.com/rndmcnlly/Lonsdaleite (2011)
40. AT&T Labs Research and Contributors: Graphviz. http://www.graphviz.org/

Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs

Christian Kloimüllner[1], Johannes Oetsch[2], Jörg Pührer[2(✉)], and Hans Tompits[2]

[1] Forschungsgruppe für Industrielle Software (INSO),
Technische Universität Wien,
Favoritenstraße 9-11, 1040 Vienna, Austria
christian.kloimuellner@inso.tuwien.ac.at

[2] Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, 1040 Vienna, Austria
{oetsch, puehrer, tompits}@kr.tuwien.ac.at

Abstract. In answer-set programming (ASP), the solutions of a problem are encoded in dedicated models, called *answer sets*, of a logical theory. These answer sets are computed from the program that represents the theory by means of an ASP solver and returned to the user as sets of ground first-order literals. As this type of representation is often cumbersome for the user to interpret, tools like ASPVIZ and IDPDraw were developed that allow for visualising answer sets. The tool Kara, introduced in this paper, follows these approaches, using ASP itself as a language for defining visualisations of interpretations. Unlike existing tools that position graphic primitives according to static coordinates only, Kara allows for more high-level specifications, supporting graph structures, grids, and relative positioning of graphical elements. Moreover, generalising the functionality of previous tools, Kara provides modifiable visualisations such that interpretations can be manipulated by graphically editing their visualisations. This is realised by resorting to abductive reasoning techniques using ASP itself. Kara is part of SeaLion, an integrated development environment (IDE) for ASP.

1 Introduction

Answer-set programming (ASP) [1] is a well-known paradigm for declarative problem solving. Its key idea is that a problem is encoded in terms of a logic program such that dedicated models of it, called *answer sets*, correspond to the solutions of the problem. Answer sets are interpretations, usually represented by sets of ground first-order literals.

A problem often faced when developing answer-set programs is that interpretations returned by an ASP solver are cumbersome to read—in particular, in

This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

case of large interpretations which are spread over several lines on the screen or the output file. Hence, a user may have difficulties extracting the relevant information from the textual representation of an answer set. Related to this issue, there is one even harder practical problem: editing or writing interpretations by hand.

Although the general goal of ASP is to have answer sets computed automatically, we identify different situations during the development of answer-set programs in which it would be helpful to have adequate means to manipulate interpretations. First, in declarative debugging [2], the user has to specify the expected semantics in order for the debugging system to identify the causes for a mismatch with the actual semantics. In previous work [3], a debugging approach has been introduced that takes a program P and an interpretation I that is expected to be an answer set of P and returns reasons why I is not an answer set of P. Manually producing such an intended interpretation ahead of computation is a time-consuming task, however. Another situation in which the creation of an interpretation can be useful is testing post-processing tools. Typically, if answer-set solvers are used within other applications, they are embedded as a module in a larger context. The overall application delegates a problem to the solver by transforming it to a respective answer-set program, and the outcome of the solver is then processed further as needed by the application. In order to test post-processing components, which may be written by programmers unaware of ASP, it would be beneficial to have means to create mock answer sets as test inputs. Third, the same idea of providing test input applies to modular answer-set programming [4], when a module B that depends on another module A is developed before or separately from A. To test B, B can be joined with interpretations mocking answer sets from A.

In this paper, we describe the system `Kara` which allows for both visualising interpretations and editing them by manipulating their visualisations.[1] The visualisation functionality of `Kara` has been inspired by the existing tools `ASPVIZ` [5] and `IDPDraw` [6] for visualising answer sets. The key idea is to use ASP itself as a language for specifying how to visualise an interpretation. To this end, the user takes a dedicated answer-set program V—which we call a *visualisation program*—that specifies how the visualisation of an interpretation I should look like. That is, V defines how different graphical elements, such as rectangles, polygons, images, graphs, etc., should be arranged and configured to visually represent I.

`Kara` offers a rich visualisation language that allows for defining a superset of the graphical elements available in `ASPVIZ` and `IDPDraw`, e.g., providing support for automatically layouting graph structures, relative and absolute positioning, and support for grids of graphical elements. Moreover, `Kara` also offers a *generic mode* of visualisation, not available in previous tools, that does not require a domain-specific visualisation program, and visualises an answer set as

[1] The name "`Kara`" derives, with all due respect, from "Kara Zor-El", the native Kryptonian name of *Supergirl*, given that Kryptonians have visual superpowers on Earth.

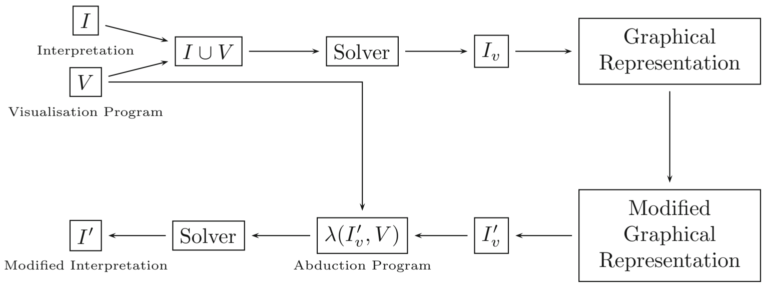


Fig. 1. Overview of the workflow (visualisation and abduction process).

a hypergraph whose set of nodes corresponds to the individuals occurring in the interpretation. A detailed overview of the differences concerning the visualisation capabilities of `Kara` with other tools is given in Sect. 5. A general difference to previous tools is that `Kara` does not just produce image files right away but presents the visualisation in form of modifiable graphical elements in a visual editor. The user can manipulate the visualisation in various ways, e.g., change size, position, or other properties of graphical elements, as well as copy, delete, and insert new ones. Notably, the created visualisations can also be used outside our editing framework as `Kara` offers an SVG export function that allows to save the possibly modified visualisation as a vector graphic. Besides fine-tuning exported SVG files, manipulation of the visualisation of an interpretation I can be done for obtaining a modified version I' of I by means of ASP-based abductive reasoning [7]. This gives the possibility to visually edit interpretations which is useful for debugging and testing purposes as described above. We will present a number of examples to illustrate the functionality of `Kara` and the ease of coping with a visualised answer set compared to interpreting its textual representation.

`Kara` is designed as a plugin of `SeaLion`, an Eclipse-based integrated development environment (IDE) for ASP [8] that is currently developed as part of a project on programming-support methods for ASP [9].

2 System Overview

We assume familiarity with the basic concepts of ASP (for a thorough introduction to the subject, cf. Baral [1]). In brief, an answer-set program consists of rules of the form

$$a_1 \vee \cdots \vee a_l \mathrel{:-} a_{l+1}, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n,$$

where $n \geq m \geq l \geq 0$, "not" denotes *default negation*, and all a_i are first-order literals (i.e., atoms possibly preceded by the *strong negation* symbol $\neg$). For a rule r as above, we define the *head* of r as $\mathrm{H}(r) = \{a_1, \ldots, a_l\}$ and the *body* as $\mathrm{B}(r) = \{a_{l+1}, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n\}$. We will also refer to the *positive body*, given as $\mathrm{B}^+(r) = \{a_{l+1}, \ldots, a_m\}$. If $n = l = 1$, r is a *fact*, and if $l = 0$, r is a *constraint.* For facts, we will usually omit the symbol "$:-$". The *grounding* of a program P relative to its Herbrand universe is defined as usual. An *interpretation* I is a finite and consistent set of ground literals, where consistency means that $\{a, \neg a\} \not\subseteq I$, for any atom a. An interpretation I is an *answer set* of a program P if it is a subset-minimal model of the grounding of the *reduct* of P relative to I (see Baral [1] for details).

The overall workflow of `Kara` is depicted in Fig. 1, illustrating how an interpretation I can be visualised in the upper row and how changing the visualisation can be reflected back to I such that we obtain a modified version I' of I in the lower row. In the following, we call programs that encode problems for which I and I' represent solution candidates *domain programs.*

2.1 Visualisation of Interpretations

As discussed in the introduction, we use ASP itself as a language for specifying how to visualise an interpretation. In doing so, we follow a similar approach as the tools `ASPVIZ` [5] and `IDPDraw` [6]. We next describe this method on an abstract level.

Assume we want to visualise an interpretation I that is defined over a first-order alphabet $\mathcal{A}$. We join I, interpreted as a set of facts, with a visualisation program V that is defined over $\mathcal{A}' \supset \mathcal{A}$, where $\mathcal{A}'$ may contain auxiliary predicates and function symbols, as well as predicates from a fixed set $\mathcal{P}_v$ of reserved *visualisation predicates* that vary for the three tools.[2]

The rules in V are used to derive different atoms with predicates from $\mathcal{P}_v$, depending on I, that control the individual graphical elements of the resulting visualisation including their presence or absence, position, and all other properties. An actual visualisation is obtained by post-processing an answer set I_v of $V \cup I$ that is projected to the predicates in $\mathcal{P}_v$. We refer to I_v as a *visualisation answer set* for I. Note that since V is an arbitrary answer-set program it might be non-deterministic in the sense that multiple visualisation answer sets may exist. In the current implementation only one of them is used for visualisation. The process is depicted in the upper row of Fig. 1. An exhaustive list of visualisation predicates available in `Kara` is given in Appendix A.

Example 1. Assume we deal with a domain program whose answer sets correspond to arrangements of items on two shelves. Consider the interpretation

$$I = \{book(s_1, 1), book(s_1, 3), book(s_2, 1), globe(s_2, 2)\}$$

stating that two books are located on shelf s_1 in positions 1 and 3 and that there is another book and a globe on shelf s_2 in positions 1 and 2, respectively. The goal is to create a simple graphical representation of this and similar interpretations,

[2] Technically, in `ASPVIZ`, V is not joined with I but with a domain program P such that I is an answer set of P.

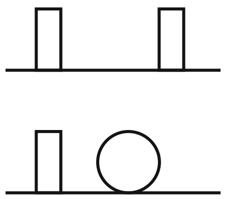


Fig. 2. The visualisation of interpretation I from Example 1.

depicting the two shelves as two lines, each book as a rectangle, and globes as circles. Consider the following visualisation program:

$$\begin{aligned}
&\mathit{visline}(\mathit{shelf}_1, 10, 40, 80, 40, 0), && (1)\\
&\mathit{visline}(\mathit{shelf}_2, 10, 80, 80, 80, 0), && (2)\\
&\mathit{visrect}(f(X,Y), 20, 8) :- \mathit{book}(X,Y), && (3)\\
&\mathit{visposition}(f(s_1,Y), 20 * Y, 20, 0) :- \mathit{book}(s_1,Y), && (4)\\
&\mathit{visposition}(f(s_2,Y), 20 * Y, 60, 0) :- \mathit{book}(s_2,Y), && (5)\\
&\mathit{visellipse}(f(X,Y), 20, 20) :- \mathit{globe}(X,Y), && (6)\\
&\mathit{visposition}(f(s_1,Y), 20 * Y, 20, 0) :- \mathit{globe}(s_1,Y), && (7)\\
&\mathit{visposition}(f(s_2,Y), 20 * Y, 60, 0) :- \mathit{globe}(s2,Y). && (8)
\end{aligned}$$

Rules (1) and (2) create two lines with the identifiers shelf_1 and shelf_2, representing the top and bottom shelf. The second to fifth arguments of *visline*/6 represent the origin and the target coordinates of the line.[3] The last argument of *visline*/6 is a z-coordinate determining which graphical element is visible in case two or more overlap. Rule (3) generates the rectangles representing books, and Rules (4) and (5) determine their position depending on the shelf and the position given in the interpretation. Likewise, Rules (6) to (8) generate and position globes. The resulting visualisation of I is depicted in Fig. 2. □

Note that the first argument of each visualisation predicate is a unique identifier for the respective graphical element. By making use of function symbols with variables, like $f(X,Y)$ in Rule (3) above, these labels are not limited to constants in the visualisation program but can be generated on the fly, depending on the interpretation to visualise. While some visualisation predicates, like *visline*, *visrect*, and *visellipse*, define graphical elements, others, e.g., *visposition*, are used to change properties of the elements, referring to them by their respective identifiers.

Kara also offers a *generic visualisation* that visualises an arbitrary interpretation without the need for defining a visualisation program. In such a case, the interpretation is represented as a labelled hypergraph. Its nodes are the

[3] The origin of the coordinate system is at the top-left corner of the illustration window with the x-axis pointing to the right and the y-axis pointing down.

individuals appearing in the interpretation and the edges represent the literals in the interpretation, connecting the individuals appearing in the respective literal. Integer labels on the endings of the edge are used for expressing the term position of the individual. To distinguish between different predicates, each edge has an additional label stating the predicate. Edges of the same predicate are of the same colour. A generic visualisation is presented in Example 4 in Sect. 4.

2.2 Editing of Interpretations

We next describe how we can obtain a modified version I' of an interpretation I corresponding to a manipulation of the visualisation of I. We follow the steps depicted in the lower row of Fig. 1, using abductive reasoning. Recall that abduction is the process of finding hypotheses that explain given observations in the context of a theory. Intuitively, in our case, the theory is the visualisation program, the observation is the modified visualisation of I, and the desired hypothesis is I'.

In `Kara`, the visualisation of I is created using the Graphical Editing Framework (GEF) [10] of Eclipse. It is displayed in a graphical editor which allows for various kinds of manipulation actions such as moving, resizing, adding or deleting graphical elements, adding or removing edges between them, editing their properties, or changing grid values. Each change in the visual editor of `Kara` is internally reflected by a modification to the underlying visualisation answer set I_v. We denote the visualisation interpretation that results from editing I_v as I'_v. From that and the visualisation program V, we construct a logic program $\lambda(I'_v, V)$ such that the visualisation of any answer set I' of $\lambda(I'_v, V)$ using V corresponds to the modified one.

The idea is that $\lambda(I'_v, V)$, which we refer to as the *abduction program* for I'_v and V, guesses a set of *abducible atoms*. On top of these atoms, the rules of V are used in $\lambda(I'_v, V)$ to derive a hypothetical visualisation answer set I''_v for I'. Finally, constraints in the abduction program ensure that I''_v coincides with the targeted visualisation interpretation I'_v on a set $\mathcal{P}_i$ of selected predicates from $\mathcal{P}_v$, which we call *integrity predicates*. Hence, a modified interpretation I' can be obtained by computing an answer set of $\lambda(I'_v, V)$ and projecting it to the guessed atoms. To summarise, the abduction problem underlying the described process can be stated as follows:

($*$) Given the interpretation I'_v, determine an interpretation I' such that I'_v coincides with each answer set of $V \cup I'$ on $\mathcal{P}_i$.

Naturally, depending on V and I'_v it is possible that no such solution I' exists. Visualisation programs must be written in a way such that manipulated visualisation interpretations could indeed be the outcome of the visualisation program for some input. This is not the case for arbitrary visualisation programs, but usually it is easy to write an appropriate visualisation program that allows for abducing interpretations.

$$\begin{aligned}
\mathsf{dom}(I'_v, V) = \{&\mathit{nonRecDom}(t) \mathrel{:-} v(\boldsymbol{t'}) \mid r \in V, v/m \in \mathcal{P}_v, v(\boldsymbol{t'}) \in \mathrm{H}(r),\\
&\quad a(\boldsymbol{t}) \in \mathrm{B}^+(r), \boldsymbol{t} = t_1, \dots, t, \dots, t_n, a/n \notin \mathcal{P}_v,\\
&\quad \mathit{VAR}(t) \neq \emptyset,\ \mathit{VAR}(t) \subseteq \mathit{VAR}(\boldsymbol{t'})\} \cup\\
\{&\mathit{dom}(t) \mathrel{:-} v(\boldsymbol{t'}), \mathit{nonRecDom}(X_1), \dots, \mathit{nonRecDom}(X_l) \mid r \in V,\\
&\quad v/m \in \mathcal{P}_v, v(\boldsymbol{t'}) \in \mathrm{H}(r), a(\boldsymbol{t}) \in \mathrm{B}^+(r), \boldsymbol{t} = t_1, \dots, t, \dots, t_n,\\
&\quad a/n \notin \mathcal{P}_v,\ \mathit{VAR}(t) \cap \mathit{VAR}(\boldsymbol{t'}) \neq \emptyset,\\
&\quad \mathit{VAR}(t) \setminus \mathit{VAR}(\boldsymbol{t'}) = \{X_1, \dots, X_l\}\} \cup\\
\{&\mathit{dom}(X) \mathrel{:-} \mathit{nonRecDom}(X)\}.
\end{aligned}$$

$$\begin{aligned}
\mathsf{guess}(V) = \{&a(X_1, \dots, X_n) \mathrel{:-} \mathrm{not}\ \neg a(X_1, \dots, X_n), \mathit{dom}(X_1), \dots, \mathit{dom}(X_n),\\
&\neg a(X_1, \dots, X_n) \mathrel{:-} \mathrm{not}\ a(X_1, \dots, X_n), \mathit{dom}(X_1), \dots, \mathit{dom}(X_n) \mid\\
&\quad a/n \notin \mathcal{P}_v, a(t_1, \dots, t_n) \in \textstyle\bigcup_{r \in V} \mathrm{B}(r),\\
&\quad \{a(t'_1, \dots, t'_n) \mid a(t'_1, \dots, t'_n) \in \mathrm{H}(r), r \in V\} = \emptyset\}.
\end{aligned}$$

$$\begin{aligned}
\mathsf{check}(I'_v) = \{&\mathrel{:-} \mathrm{not}\ v(t_1, \dots, t_n),\ \mathrel{:-} v(X_1, \dots, X_n), \mathrm{not}\ v'(X_1, \dots, X_n),\\
&v'(t_1, \dots, t_n) \mid v(t_1, \dots, t_n) \in I'_v, v/n \in \mathcal{P}_i\}.
\end{aligned}$$

Fig. 3. Elements of the abduction program $\lambda(I'_v, V)$.

The following problems have to be addressed for realising the sketched approach:

– determining the predicates and domains of the abducible atoms, and
– choosing the integrity predicates among the visualisation predicates.

For solving these issues, we rely on pragmatic choices that seem useful in practice. We obtain the set $\mathcal{P}_a$ of predicates of the abducible atoms from the visualisation program V. The idea is that every predicate that is relevant to the solution of a problem encoded in an answer set has to occur in the visualisation program if the latter is meant to provide a complete graphical representation of the solution. Moreover, we restrict $\mathcal{P}_a$ to those non-visualisation predicates in V that occur in the body of a rule but not in any head atom in V. The assumption is that atoms defined in V are most likely of auxiliary nature and not contained in a domain program.

An easy approach for generating a domain $\mathcal{D}_a$ of the abducible atoms would be to extract the terms occurring in I'_v. We follow, however, a more fine-grained approach that takes the introduction and deletion of function symbols in the rules in V into account. Assume V contains the rules

$$\mathit{visrect}(f(\mathit{Street}, \mathit{Num}), 9, 10) \mathrel{:-} \mathit{house}(\mathit{Street}, \mathit{Num}) \quad \text{and}$$
$$\mathit{visellipse}(\mathit{sun}, \mathit{Width}, \mathit{Height}) \mathrel{:-} \mathit{property}(\mathit{sun}, \mathit{size}(\mathit{Width}, \mathit{Height})),$$

and I'_v contains $\mathit{visrect}(f(\mathit{bakerstreet}, 221b), 9, 10)$ and $\mathit{visellipse}(\mathit{sun}, 10, 11)$. Then, when extracting the terms in I'_v, the domain includes $f(\mathit{bakerstreet}, 221b)$, *bakerstreet*, $221b$, 9, 10, *sun*, and 11 for the two rules. However, the functor f is solely an auxiliary concept in V and not meant to be part of domain programs.

Moreover, the term 9 is introduced in V and is not needed in the domain for I'. Also, the terms 10 and 11 as standalone terms and *sun* are not needed in I' to derive I'_v. Even worse, the term $\mathit{size}(10, 11)$, that has to be contained in I' such that I'_v can be a visualisation answer set for I', is missing in the domain. Hence, we derive $\mathcal{D}_a$ in $\lambda(I'_v, V)$ not only from I'_v but also consider the rules in V. Using our translation detailed below, we obtain *bakerstreet*, $221b$, and $\mathit{size}(10, 12)$ as domain terms from the rules above.

For the choice of $\mathcal{P}_i$, i.e., of the predicates on which I'_v and the actual visualisation answer sets of I' need to coincide, we exclude visualisation predicates that require a high preciseness in visual editing by the user in order to match exactly a value that could result from the visualisation program. For example, we do not include predicates determining position and size of graphical elements, since in general it is hard to position and scale an element precisely such that an interpretation I' exists with a matching visualisation. Note that this is not a major restriction, as in general it is easy to write a visualisation program such that aspects that the user wants to be modifiable are represented by graphical elements that can be elegantly modified visually. For example, instead of representing a Sudoku puzzle by labels whose exact position is calculated in the visualisation program, the language of `Kara` allows for using a logical grid such that the value of each cell can be easily changed in the visual editor.

We next give the details of the abduction program.

Definition 1. *Let I'_v be an interpretation over predicates in $\mathcal{P}_v$, V a (visualisation) program, and $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates. Moreover, let $\mathit{VAR}(T)$ denote the variables occurring in T, where T is a term or a list of terms. Then, the* abduction program *$\lambda(I'_v, V)$ with respect to I'_v and V is given by*

$$\lambda(I'_v, V) = \mathsf{dom}(I'_v, V) \cup \mathsf{guess}(V) \cup V \cup \mathsf{check}(I'_v),$$

where $\mathsf{dom}(I'_v, V)$, $\mathsf{guess}(V)$, and $\mathsf{check}(I'_v)$ are given in Fig. 3, and $\mathit{nonRecDom}/1$, $\mathit{dom}/1$, and v'/n, for all $v/n \in \mathcal{P}_i$, are fresh predicates.

The idea of $\mathsf{dom}(I'_v, V)$ is to consider non-ground terms t contained in the body of a visualisation rule that share variables with a visualisation atom in the head of the rule and to derive instances of these terms when the corresponding visualisation atom is contained in I'_v. In case less variables occur in the visualisation atom than in t, we avoid safety problems by restricting their scope to parts of the derived domain. Here, the distinction between predicates *dom* and *nonRecDom* is necessary to prevent infinite groundings of the abduction program. The next part of the abduction program is $\mathsf{guess}(V)$, where the atoms of the domain program P are guessed, i.e., the abducible atoms. The output of the guessing part is I'. Finally, $\mathsf{check}(I'_v)$ contains all constraints and auxiliary facts.

Note that in general it is not guaranteed that the domain we derive contains all necessary elements for abducing an appropriate interpretation I'. For instance, consider the case that the visualisation program contains a rule

$$\mathit{visrect}(\mathit{id}, 5, 5) :- \mathit{foo}(X),$$

and V, together with the constraints in $\mathsf{check}(I'_v)$, require that for every term t of a domain that can be obtained from I'_v and V, $foo(t)$ must not hold. Then, there is no interpretation that will trigger the rule using this domain, although an interpretation with a further term t' might exist that results in the desired visualisation. Hence, we added an editor to Kara that allows for changing and extending the automatically generated domain as well as the set of abducible predicates.

The following two results characterise the answer sets of the abduction program.

Theorem 1. *Let I'_v be an interpretation with atoms over predicates in $\mathcal{P}_v$, V a (visualisation) program, and $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates. Then, any answer set I''_v of $\lambda(I'_v, V)$ coincides with I'_v on the atoms over predicates from $\mathcal{P}_i$.*

Note that since the stated abduction problem $(*)$ requires I'_v to coincides with each answer set of $V \cup I'$ on $\mathcal{P}_i$, a solution is only given in case the visualisation program deterministically derives a visualisation, as expressed in Theorem 2.

Theorem 2. *Let I'_v be an interpretation with atoms over $\mathcal{P}_v$, $\mathcal{P}_i \subseteq \mathcal{P}_v$ the fixed set of integrity predicates, and V a (visualisation) program such that, for every I' with atoms over $\mathcal{P}_d$, where*

$$\mathcal{P}_d = \{a/n \mid \textit{there are terms } t_1, \ldots, t_n \textit{ such that } a(t_1, \ldots, t_n) \in \bigcup_{r \in V} \mathrm{B}(r) \textit{ but there are no terms } t'_1, \ldots, t'_n \textit{ such that } a(t'_1, \ldots, t'_n) \in \bigcup_{r \in V} \mathrm{H}(r)\} \setminus \mathcal{P}_v,$$

every two answer sets I_1 and I_2 of $V \cup I'$ do not differ on $\mathcal{P}_i$. Then, for any answer set I''_v of $\lambda(I'_v, V)$, a solution I' of the abduction problem $(*)$ is obtained by projecting I''_v to the predicates in $\mathcal{P}_d$.

3 Implementation

Kara is written in Java and integrated in the Eclipse-plugin SeaLion [8] for developing answer-set programs. SeaLion is an IDE that offers functionality

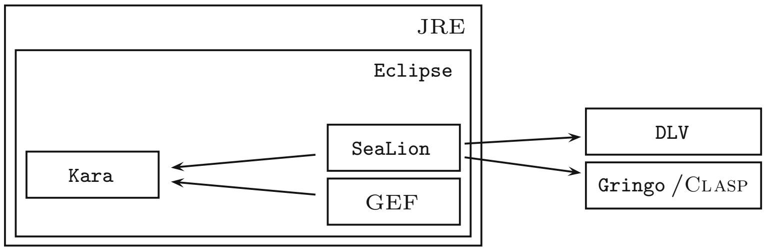
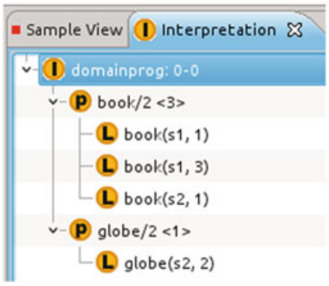


Fig. 4. Technology stack of the Kara system.



```
domainprog: 0-0
  book/2 <3>
    book(s1, 1)
    book(s1, 3)
    book(s2, 1)
  globe/2 <1>
    globe(s2, 2)
```

Fig. 5. Sample output in the *interpretation view* of `SeaLion`.

to execute external ASP solvers on answer-set programs and thus realises the interface between `Kara` and ASP solvers. The overall technology stack of `Kara` is depicted in Fig. 4. Currently, programs in the languages of `Gringo` and `DLV` are supported.

Next we describe how to use `Kara` in `SeaLion`. The ASP developer may invoke ASP solver calls in `SeaLion` using Eclipse's launch-configuration framework in a similar fashion as Java programs are started from within Eclipse. Answer sets resulting from a solver call can be parsed by the IDE and displayed as expandable tree structures in a dedicated Eclipse view for interpretations as shown in Fig. 5. Starting from there, the user can invoke `Kara` by choosing a pop-up menu entry of the interpretation that should be visualised. Here, one can select between the generic mode of visualisation or a mode using a visualisation program created by the user. In the latter case, a customised run configuration dialog will open that allows for choosing a file containing the visualisation program and for setting the solver configuration, including selection of the solver and command-line arguments, to be used by `Kara`. Then, the visual editor opens with the generated visualisation. It allows for changing the visualisation in various ways, including repositioning, rescaling, and renaming of graphical elements, as well as creation, deletion, and making copies of elements. The current state of the visualisation can always be exported in an SVG file. Moreover, the process for abducing an interpretation that reflects the modifications to the visualisation can be started from a pop-up menu of the visual editor. If a respective interpretation exists, it will be added to the interpretation view of `SeaLion`.

`Kara` and `SeaLion` can be downloaded and installed from within Eclipse, using the following update site:

http://sealion.at/update.

For more information and installation instructions, we refer to the project web site

http://www.sealion.at.

$$\mathit{visgrid}(\mathit{maze}, MR, MC, MR{*}20{+}5, MC{*}20{+}5) :- \mathit{maxC}(MC), \mathit{maxR}(MR). \tag{9}$$
$$\mathit{visposition}(\mathit{maze}, 0, 0, 0). \tag{10}$$

`% A cell with a wall on it.`

$$\mathit{visrect}(\mathit{wall}, 20, 20). \tag{11}$$
$$\mathit{visbackgroundcolor}(\mathit{wall}, \mathit{black}). \tag{12}$$

`% An empty cell.`

$$\mathit{visrect}(\mathit{empty}, 20, 20). \tag{13}$$
$$\mathit{visbackgroundcolor}(\mathit{empty}, \mathit{white}). \tag{14}$$
$$\mathit{viscolor}(\mathit{empty}, \mathit{white}). \tag{15}$$

`% Entrance and exit.`

$$\mathit{visimage}(\mathit{entrance}, \text{“}\mathit{entrance.jpg}\text{”}). \tag{16}$$
$$\mathit{visscale}(\mathit{entrance}, 18, 18). \tag{17}$$
$$\mathit{visimage}(\mathit{exit}, \text{“}\mathit{exit.png}\text{”}). \tag{18}$$
$$\mathit{visscale}(\mathit{exit}, 18, 18). \tag{19}$$

`% Filling the cells of the grid.`

$$\mathit{visfillgrid}(\mathit{maze}, \mathit{empty}, R, C) :- \mathit{empty}(C, R), \text{not } \mathit{entrance}(C, R), \text{not } \mathit{exit}(C, R). \tag{20}$$
$$\mathit{visfillgrid}(\mathit{maze}, \mathit{wall}, R, C) :- \mathit{wall}(C, R), \text{not } \mathit{entrance}(C, R), \text{not } \mathit{exit}(C, R). \tag{21}$$
$$\mathit{visfillgrid}(\mathit{maze}, \mathit{entrance}, R, C) :- \mathit{entrance}(C, R). \tag{22}$$
$$\mathit{visfillgrid}(\mathit{maze}, \mathit{exit}, R, C) :- \mathit{exit}(C, R). \tag{23}$$

`% Vertical and horizontal lines.`

$$\mathit{visline}(v(0), 5, 5, 5, MR * 20 + 5, 1) :- \mathit{maxR}(MR). \tag{24}$$
$$\mathit{visline}(v(C), C{*}20{+}5, 5, C{*}20{+}5, MR{*}20{+}5, 1) :- \mathit{col}(C), \mathit{maxR}(MR). \tag{25}$$
$$\mathit{visline}(h(0), 5, 5, MC * 20 + 5, 5, 1) :- \mathit{maxC}(MC). \tag{26}$$
$$\mathit{visline}(h(R), 5, R * 20 + 5, MC * 20 + 5, R * 20 + 5, 1) :- \mathit{row}(R), \mathit{maxC}(MC). \tag{27}$$

`% Define possible grid values for editing.`

$$\mathit{vispossiblegridvalues}(\mathit{maze}, \mathit{wall}). \tag{28}$$
$$\mathit{vispossiblegridvalues}(\mathit{maze}, \mathit{empty}). \tag{29}$$
$$\mathit{vispossiblegridvalues}(\mathit{maze}, \mathit{entrance}). \tag{30}$$
$$\mathit{vispossiblegridvalues}(\mathit{maze}, \mathit{exit}). \tag{31}$$

Fig. 6. Visualisation program for Example 2.

4 Examples

In this section, we provide examples that give an overview of the functionality of Kara. We first illustrate the use of logic grids and the visual editing feature.

Example 2. Maze generation is a benchmark problem from the second ASP competition [11]. The task is to generate a two-dimensional grid, where each cell is either a wall or empty, that satisfies certain constraints. There are two dedicated empty cells, being the maze's entrance and its exit, respectively. The following facts represent a sample answer set of a maze-generation encoding restricted to interesting predicates:

$col(1..5).$ $row(1..5).$ $maxC(5).$ $maxR(5).$ $wall(1,1).$ $empty(1,2).$ $wall(1,3).$ $wall(1,4).$ $wall(1,5).$ $wall(2,1).$ $empty(2,2).$ $empty(2,3).$ $empty(2,4).wall(2,5).$ $wall(3,1).$ $wall(3,2).$ $wall(3,3).$ $empty(3,4).$ $wall(3,5).$ $wall(4,1).$ $empty(4,2).$

$empty(4,3).$ $empty(4,4).$ $wall(4,5).$ $wall(5,1).wall(5,2).$ $wall(5,3).$ $empty(5,4).$ $wall(5,5).$ $entrance(1,2).$ $exit(5,4).$

Predicates *col*/1 and *row*/1 define indices for the rows and columns of the maze, while *maxC*/1 and *maxR*/1 give the maximum column and row index, respectively. The predicates *wall*/2, *empty*/2, *entrance*/2, and *exit*/2 determine the positions of walls, empty cells, the entrance, and the exit in the grid, respectively. One may use the visualisation program from Fig. 6 for maze-generation interpretations of this kind.

In Fig. 6, Rule (9) defines a logic grid with identifier *maze* having *MR* rows and *MC* columns. The fourth and fifth parameter define the height and width of the grid in pixel. Rule (10) is a fact that defines a fixed position for the maze. The next step is to define the graphical objects to be displayed in the grid. Because these objects are fixed (i.e., they are used more than once), they can be defined as facts. A wall is represented by a rectangle with black background and foreground colour[4] (Rules (11) and (12)) whereas an empty cell is rendered as a rectangle with white background and foreground colour (Rules (13) to (15)). The entrance and the exit are represented by two images (Rules (16 to (19)). Then, these graphical elements are assigned to the respective cell of the grid (Rules (20) to (23)). Rules (24) to (27) render vertical and horizontal lines to better distinguish between the different cells. Rules (28) to (31) are not needed for visualisation but define possible values for the grid that we want to be available in the visual editor.

Once the grid is rendered, the user can replace the value of a cell with a value defined by using predicate *vispossiblegridvalues*/2 (e.g., replacing an empty cell

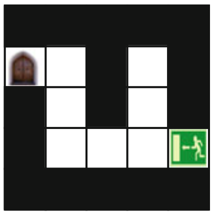


Fig. 7. Visualisation output for the maze-generation program.

[4] Black foreground colour is default and need not be set explicitly.

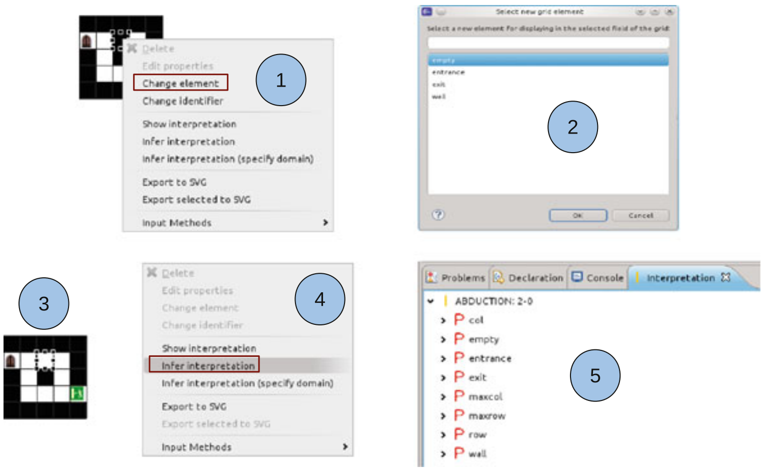


Fig. 8. Abduction steps in the plugin.

with a wall). The visualisation of the sample interpretation using this program is depicted in Fig. 7. Evidently, the visual representation of the answer set is more accessible than the textual representation of the answer set given in the beginning of the example.

Next, we demonstrate how to use the visual editing feature of Kara to obtain a modified interpretation; the respective steps are illustrated by Fig. 8. Suppose we want to change the cell $(3, 2)$ from being a wall to an empty cell. The user can select the respective cell and open a pop-up menu that provides an item for changing grid-values. A dialog opens that allows for choosing among the values that have been defined in the visualisation program, using the *vispossiblegridvalues*/2 predicate. When the user has finished editing the visualisation, the abduction process for inferring the new interpretation can be started. After an interpretation is derived, it is added to SeaLion's interpretation view. □

Kara supports absolute and relative positioning of graphical elements. If for any visualisation element the predicate *visposition*/4 is defined, then fixed positioning is used. Otherwise, the element is positioned automatically. Then, by default, the elements are randomly positioned on the graphical editor. However, the user can define the position of an element *relative* to another element. This is done by using the predicates *visleft*/2, *visright*/2, *visabove*/2, *visbelow*/2, and *visinfrontof*/2.

Example 3. The following visualisation program makes use of relative positioning for sorting elements according to their label.

$$visrect(a, 50, 50). \tag{32}$$

$$vislabel(a, laba). \tag{33}$$

$$vistext(laba, 3). \tag{34}$$

$$vispolygon(b, 0, 20, 1). \tag{35}$$
$$vispolygon(b, 25, 0, 2). \tag{36}$$
$$vispolygon(b, 50, 20, 3). \tag{37}$$
$$vislabel(b, labb). \tag{38}$$
$$vistext(labb, 10). \tag{39}$$
$$visellipse(c, 30, 30). \tag{40}$$
$$vislabel(c, labc). \tag{41}$$
$$vistext(labc, 5). \tag{42}$$
$$element(X) :- \ visrect(X, \_, \_). \tag{43}$$
$$element(X) :- \ vispolygon(X, \_, \_, \_). \tag{44}$$

$$element(X) :- \ visellipse(X, \_, \_). \tag{45}$$
$$\begin{aligned} visleft(X, Y) :- \ & element(X), element(Y), vislabel(X, LABX), \\ & vistext(LABX, XNUM), vislabel(Y, LABY), \\ & vistext(LABY, YNUM), XNUM < YNUM. \end{aligned} \tag{46}$$

The program defines three graphical objects, a rectangle, a polygon, and an ellipse. In Rules (32) to (34), the rectangle together with its label, 3, is generated. The shape of the polygon (Rules (35) to (37)) is defined by a sequence of points relative to the polygon's own coordinate system using the *vispolygon*/4 predicate. The order in which these points are connected with each other is given by the predicate's fourth argument. Rules (38) and (39) generate the label for the polygon and specify its text. Rules (43) to (45) state that every rectangle, polygon, and ellipse is an element. The relative position of the three elements is determined by Rule (46). For two elements E_1 and E_2, E_1 has to appear to the left of E_2 whenever the label of E_1 is smaller than the one of E_1.

The output of this visualisation program is shown in Fig. 9. Note that the visualisation program does not make reference to predicates from an interpretation to visualise, hence the example illustrates that `Kara` can also be used for creating arbitrary graphics. □

The last example demonstrates the support for graphs in `Kara`. Moreover, the generic visualisation feature is illustrated.

Example 4. Suppose we want to visualise answer sets of an encoding of a graph-colouring problem. Assume we have the following interpretation that defines nodes and edges of a graph as well as a colour for each node:

$$\begin{aligned}\{&node(1),\ node(2),\ node(3),\ node(4),\ node(5),\ node(6),\ edge(1,2),\ edge(1,3), \\ &edge(1,4),\ edge(2,4),\ edge(2,5),\ edge(2,6),\ edge(3,1),\ edge(3,4),\ edge(3,5), \\ &edge(4,1),\ edge(4,2),\ edge(5,3),\ edge(5,4),\ edge(5,6),\ edge(6,2),\ edge(6,3), \\ &edge(6,5),\ colour(1, lightblue),\ colour(2, yellow),\ colour(3, yellow), \\ &colour(4, red), colour(5, lightblue),\ colour(6, red)\}.\end{aligned}$$

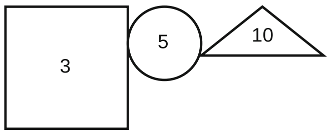


Fig. 9. Output of the visualisation program in Example 3.

We make use of the following visualisation program:

% `Generate a graph.`

$$visgraph(g). \tag{47}$$

% `Generate the nodes of the graph.`

$$visellipse(X, 20, 20) :- node(X). \tag{48}$$

$$visisnode(X, g) :- node(X). \tag{49}$$

% `Connect the nodes (edges of the input).`

$$visconnect(f(X, Y), X, Y) :- edge(X, Y). \tag{50}$$

$$vistargetdeco(X, arrow) :- visconnect(X, \_, \_). \tag{51}$$

% `Generate labels for the nodes.`

$$vislabel(X, l(X)) :- node(X). \tag{52}$$

$$vistext(l(X), X) :- node(X). \tag{53}$$

$$visfontstyle(l(X), bold) :- node(X). \tag{54}$$

% `Colour the node according to the solution.`

$$visbackgroundcolor(X, COL) :- node(X), color(X, COL). \tag{55}$$

In Rule (47), a graph, g, is defined and a circle for every node from the input interpretation is created (Rule (48)). Rule (49) states that each of these circles is logically considered a node of graph g. This has the effect that they will be considered by the algorithm layouting the graph during the creation of the visualisation. The edges of the graph are defined using the $visconnect/3$ predicate (Rule (50)). It can be used to connect arbitrary graphical elements with a line, also if they are not nodes of some graph. As we deal with a directed graph, an arrow is set as target decoration for all the connections (Rule (51)). Labels for the nodes are set in Rules (52) to (54). Finally, Rule (55) sets the colour of the node according to the interpretation. The resulting visualisation is depicted in Fig. 10. Moreover, the generic visualisation of the graph colouring interpretation is given in Fig. 11. □

5 Related Work

The visualisation method realised in `Kara` follows the approach of the previous systems `ASPVIZ` [5] and `IDPDraw` [6], which also use ASP for defining how

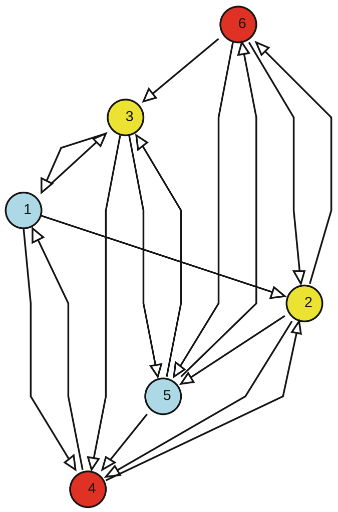


Fig. 10. Visualisation of a coloured graph.

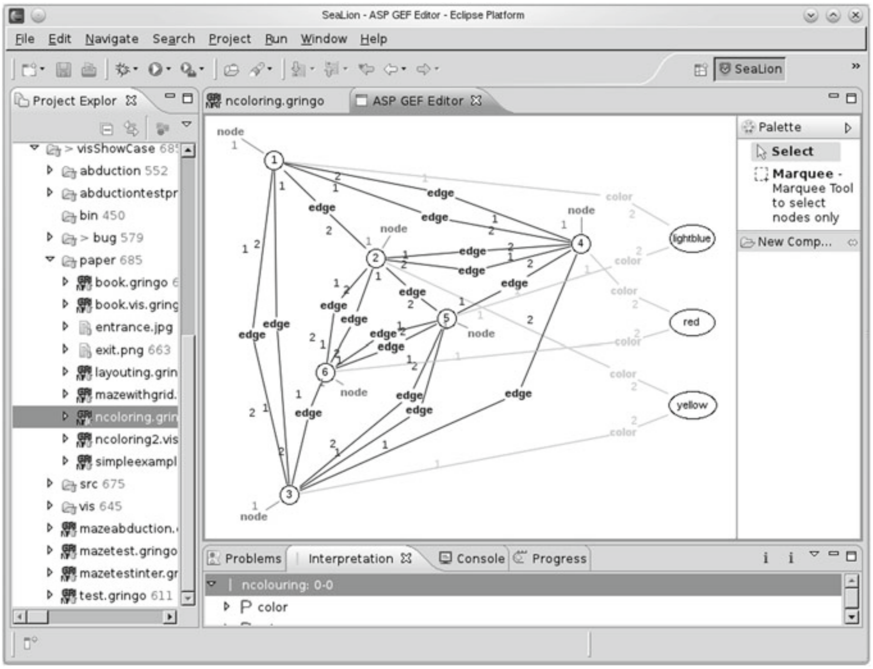


Fig. 11. `SeaLion`'s visual interpretation editor showing a generic visualisation of the graph colouring interpretation of Example 4 (the layout has been manually optimised).

interpretations should be visualised. Besides the features beyond visualisation, viz. the framework for editing visualisations and the support for multiple ASP solvers, there are also differences between Kara and these tools regarding visualisation aspects.

Kara allows to write more high-level specifications for positioning the graphical elements of a visualisation. While IDPDraw and ASPVIZ require the use of absolute coordinates, Kara additionally supports relative positioning and automatic layouting for graph and grid structures. Note that, technically, the former is realised by using ASP for guessing positions of the individual elements and adding respective constraints to ensure the correct layout, while the latter is realised by using a standard graph layouting algorithm which is part of the Eclipse framework. In Kara, as well as in IDPDraw, each graphical element has a unique identifier that can be used, e.g., to link elements or to set their properties (e.g., colour or font style). That way, programs can be written in a clear and elegant way since not all properties of an element have to be specified within a single atom. Here, Kara exploits that the latest ASP solvers support function symbols that allow for generating new identifiers from terms of the interpretation to visualise. IDPDraw does not support function symbols, however. Instead, for compound identifiers, IDPDraw uses predicates of variable length (e.g., $idp\_polygon(id_1, id_2, ...)$). A disadvantage of this approach is that some solvers, like DLV, do not support predicates of variable length. ASPVIZ does not support identifiers for graphical objects at all. However, it does allow for defining named properties used for drawing, e.g., brushes of different colours and sizes can be assigned to constants that are then needed to be set in the command for drawing lines.

The support for a z-axis to determine which object should be drawn over others is available in Kara and IDPDraw but missing in ASPVIZ. However, a new version of ASPVIZ supports visualisations by 3-dimensional objects. Both Kara and ASPVIZ support the export of visualisations as vector graphics in the SVG format, which is not possible with IDPDraw. A feature that is supported by ASPVIZ and IDPDraw, however, is creating animations which is not possible with Kara so far. Note that an export feature for animations could be easily integrated in Kara as well.

Kara and ASPVIZ are written in Java and depend only on a Java Virtual Machine. IDPDraw, on the other hand, is written in C++ and depends on the qt libraries. Finally, Kara is embedded in an IDE, whereas ASPVIZ and IDPDraw are stand-alone tools.

Recently, a further visualisation tool for answer-sets that uses ASP for specifying visualisations has been written [12]. This system, called Lonsdaleite, is a lightweight Python script for visualising graph structures by mapping the atoms in an answer set to the input format of the graphviz utilities [13]. Thus, it only supports rendering a problem in a graph, but not other elements available in ASPVIZ, IDPDraw, and Kara.

A related approach from software engineering is the `Alloy Analyzer`, a tool to support the analysis of declarative software models [14]. Models are formulated in a first-order based specification language. The user may define object signatures, which can have properties (e.g., relationships to other objects). The tool can find satisfying instances of a model using translations to SAT. Since the approach is based on finding models for declarative specifications, it can be regarded as a form of ASP in a broader sense. The derived model instances are first-order structures that can be automatically visualised as graphs, where objects are represented as nodes and their relationships are represented as edges. Hence, visualisation in `Alloy` is closely related to the generic visualisation mode of `Kara` where also no dedicated visualisation program is needed. Finally, a useful feature of `Alloy` is filtering predicates and arguments away of the graph.

6 Conclusion

We presented the tool `Kara` for visualising and visual editing of interpretations in ASP. It supports generic as well as customised visualisations. A powerful language for defining a visualisation by means of ASP is provided, supporting, e.g., graph layouting, grids of graphical elements, and relative positioning. The editing feature uses abductive reasoning, inferring a new interpretation as hypothesis to explain a modified visualisation.

In future work, we want to add support for defining input and output signatures for programs in `SeaLion`. Then, the abduction framework of `Kara` could be easily extended such that one can derive inputs for a domain program such that one of its answer sets corresponds to a modified visualisation. We also consider adding a feature similar to the filtering mode of Alloy for getting a clearer generic visualisation. Moreover, we want to investigate model-driven development for ASP involving domain models that allow for obtaining generic visualisations that take structural information into account.

A Predefined Visualisation Predicates in Kara

Atom	Intended meaning
$visellipse(id, height, width)$	Defines an ellipse with specified height and width.
$visrect(id, height, width)$	Defines a rectangle with specified height and width.
$vispolygon(id, x, y, ord)$	Defines a point of a polygon. The ordering defines in which order the defined points are connected with each other.
$visimage(id, path)$	Defines an image given in the specified file.
$visline(id, x_1, y_1, x_2, y_2, z)$	Defines a line between the points (x_1, y_1) and (x_2, y_2).
$visgrid(id, rows, cols, height, width)$	Defines a grid with the specified number of rows and columns; *height* and *width* determine the grid size.
$visgraph(id)$	Defines a graph.
$vistext(id, text)$	Defines a text element.
$vislabel(id_g, id_t)$	Sets the text element id_t as a label for graphical element id_g. Labels are supported for the following elements: *visellipse*/3, *visrect*/3, *vispolygon*/4, and *visconnect*/3.
$visisnode(id_n, id_g)$	Adds the graphical element id_n as a node to a graph id_g for automatic layouting. The following elements are supported as nodes: *visrect*/3, *visellipse*/3, *vispolygon*/4, *visimage*/2.
$visscale(id, height, weight)$	Scales an image to the specified height and width.
$visposition(id, x, y, z)$	Puts an element *id* on the fixed position (x, y, z).
$visfontfamily(id, ff)$	Sets the specified font *ff* for a text element *id*.
$visfontsize(id, size)$	Sets the font size *size* for a text element *id*.
$visfontstyle(id, style)$	Sets the font style for a text element *id* to bold or italics.
$viscolor(id, color)$	Sets the foreground colour for the element *id*.
$visbackgroundcolor(id, color)$	Sets the background colour for the element *id*.
$visfillgrid(id_g, id_c, row, col)$	Puts element id_c in cell (row, col) of the grid id_g.
$visconnect(id_c, id_{g_1}, id_{g_2})$	Connects two elements, id_{g_1} and id_{g_2}, by a line such that id_{g_1} is the source and id_{g_2} is the target of the connection.
$vissourcedeco(id, deco)$	Sets the source decoration for a connection.
$vistargetdeco(id, deco)$	Sets the target decoration for a connection.
$visleft(id_l, id_r)$	Ensures that the x-coordinate of id_l is less than that of id_r.
$visright(id_r, id_l)$	Ensures that the x-coordinate of id_r is greater than that of id_l.
$visabove(id_t, id_b)$	Ensures that the y-coordinate of id_t is smaller than that of id_b.
$visbelow(id_b, id_t)$	Ensures that the y-coordinate of id_b is greater than that of id_t.
$visinfrontof(id_1, id_2)$	Ensures that the z-coordinate of id_1 is greater than that of id_2.
$vishide(id)$	Hides the element *id*.
$visdeletable(id)$	Defines that the element *id* can be deleted in the visual editor.
$viscreatable(id)$	Defines that the element *id* can be created in the visual editor.
$vischangable(id, prop)$	Defines that the property *prop* can be changed for the element *id* in the visual editor.
$vispossiblegridvalues(id, id_e)$	Defines that graphical element id_e is available as possible grid value for a grid *id* in the visual editor.

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Shapiro, E.Y.: Algorithmic program debugging. Ph.D. thesis, Yale University, New Haven, CT, USA (1982)
3. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: on debugging nonground answer-set programs. Theor. Pract. Logic Program. **10**(4–5), 513–529 (2010)
4. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J Artif. Intell. Res. **35**, 813–857 (2009)
5. Cliffe, O., De Vos, M., Brain, M., Padget, J.: ASPVIZ: declarative visualisation and animation using answer set programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 724–728. Springer, Heidelberg (2008)
6. Wittocx, J.: KRR Software: IDPDraw. https://dtai.cs.kuleuven.be/krr/software/visualisation
7. Peirce, C.S.: Abduction and induction. In: Buchler, J. (ed.) Philosophical Writings of C.S Peirce, Chapter 11, pp. 150–156. Dover, New York (1955)
8. Oetsch, J., Pührer, J., Tompits, H.: The SeaLion has landed: An IDE for answer-set programming–preliminary report. In: Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A. (eds.) INAP/WLP 2011. LNCS, vol. 7773, pp. 305–324. Springer, Heidelberg (2013)
9. Oetsch, J., Pührer, J., Tompits, H.: Methods and methodologies for developing answer-set programs - Project description. In: Leibniz International Proceedings in Informatics (LIPIcs) of Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010), Dagstuhl, Germany, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, vol. 7, pp. 154–161 (2010)
10. The Eclipse Foundation: GEF (Graphical Editing Framework). http://www.eclipse.org/gef/
11. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
12. Smith, A.: Lonsdaleite. https://github.com/rndmcnlly/Lonsdaleite
13. AT&T Labs Research and Contributors: Graphviz. http://www.graphviz.org/
14. Jackson, D.: Software Abstractions–Logic, Language, and Analysis. MIT Press, Cambridge (2006)

Unit Testing in *ASPIDE*

Onofrio Febbraro[1], Nicola Leone[2], Kristian Reale[2](✉), and Francesco Ricca[2]

[1] DLVSystem s.r.l. - P.zza Vermicelli, Polo Tecnologico,
87036 Rende, Italy
`febbraro@dlvsystem.com`

[2] Dipartimento di Matematica, Università della Calabria,
87036 Rende, Italy
{`leone, reale, ricca`}`@mat.unical.it`

Abstract. Answer Set Programming (ASP) is a declarative logic programming formalism, which is employed nowadays in both academic and industrial real-world applications. Although some tools for supporting the development of ASP programs have been proposed in the last few years, the crucial task of *testing* ASP programs received less attention and it is an Achilles' heel of the available programming environments. In this paper we present a language for specifying and running *unit tests* on ASP programs. The testing language was implemented in *ASPIDE*, a comprehensive IDE for ASP, which supports the entire life cycle of ASP development with a collection of user-friendly graphical tools for program composition, *testing*, debugging, profiling, solver execution configuration, and output handling.

1 Introduction

Answer Set Programming (ASP) [1] is a declarative logic programming formalism proposed in the area of non-monotonic reasoning. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find those solutions [2]. The language of ASP [1] supports a number of modeling constructs including disjunction in rule heads, nonmonotonic negation [1], (weak and strong) constraints [3], aggregate functions [4], and more. These features make ASP very expressive [5], and suitable for developing advanced real-world applications. ASP is employed in several fields, from Artificial Intelligence [6–11] to Information Integration [12], and Knowledge Management [13,14], Bioinformatics [15], Software Packaging [16], etc. Interestingly, these applications of ASP recently have stimulated some interest also in industry [14,17,18].

On the one hand, the effective application of ASP in real-world scenarios was made possible by the availability of efficient ASP systems [6,19,20]. On the other hand, the adoption of ASP can be further boosted by offering effective programming tools capable of supporting the programmers in managing large and complex projects [21].

In the last few years, a number of tools for developing ASP programs have been proposed, including editors and debuggers [22–32]. Among them, *ASPIDE* [32], which stands for Answer Set Programming Integrated Development Environment, is one of the most complete development tools.[1] *ASPIDE* features a cutting-edge editing tool (offering dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, etc.) and a collection of user-friendly graphical tools for program composition, debugging, profiling, DBMS access, solver execution configuration and output handling. Although so many tools for developing ASP programs have been proposed up to now, the crucial task of *testing* ASP programs received less attention [33,34], and it is an Achilles' heel of the available programming environments. Indeed, the majority of available graphic programming environments for ASP does not provide the user with a testing tool (see [32]), and also the one present in the first versions of *ASPIDE* is far from being effective.

In this paper we present a pragmatic solution for testing ASP programs. In particular, we define the notion of *unit test* for an ASP program, and present a new language inspired by the JUnit [35] framework for specifying and running unit tests on ASP programs. Unit testing is a white-box testing technique that requires to assess separately subparts of a source code called *units* to verify whether they behave as intended. The testing language allows the developer to specify the rules composing one or several units, specify one or more inputs and assert a number of conditions on both expected outputs and the expected behavior of sub-programs. The obtained test case specification can be run by exploiting an ASP solver, and the assertions are automatically verified by analyzing the output of the execution. Notably, the test case specification language herein presented is general and applicable to any variant/dialect of ASP. The testing language was implemented in *ASPIDE*, which also provides the user with some graphic tools that make the development of test cases simpler. The testing tool described in this work extends significantly the one formerly available in *ASPIDE*, by: (i) extending the language by more expressive (non-ground) assertions and the support for weak-constraints, (ii) supporting the notion of DLP-function [36] in the specification of test-cases, and, (iii) introducing in *ASPIDE* a graphical test suite management interface.

As far as related work is concerned, the task of testing ASP programs was approached for the first time, to the best of our knowledge, in [33,34] where the notion of structural testing for ground normal ASP programs is defined and methods for automatically generating tests is introduced. The results presented in [33,34] are, somehow, orthogonal to the contribution of this paper. Indeed,

[1] For an exhaustive feature-wise comparison with existing environments for developing logic programs we refer the reader to [32].

no language/implementation is proposed in [33,34] for specifying/automatically-running the produced test cases. Whereas, the language presented in this paper can be used for encoding the output of a test case generator based on the methods proposed in [33]. Finally, it is worth noting that, testing approaches developed for other logic languages, like PROLOG [37–39], cannot be straightforwardly ported to ASP because of the differences between the languages.[2]

This paper is organized as follows: in Sect. 2 we introduce ASP; in Sect. 3 we introduce the notion of a unit test for ASP programs and we present a language for specifying unit tests; in Sect. 4 we describe the user interface components of *ASPIDE* conceived for creating and running tests; finally, in Sect. 5 we draw the conclusion.

2 Answer Set Programming

In this section we first present some preliminaries notions concerning ASP [1], then we recall some relevant modularity properties of ASP programs that are exploited in the testing language described in the following sections; finally, we briefly introduce some common language extensions that we use in our examples.

Hereafter, we assume the reader to be familiar with logic programming, and refer to both [13,40] for complementary introductory material on ASP.

Syntax. A variable or a constant is a *term*. An *atom* is of the form $p(t_1, ..., t_n)$, where p is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* $not\ p$, where p is an atom. A *(disjunctive) rule* r has the following form:

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \ {:}{-} \ \ b_1, \cdots, b_k, \ not \ \ b_{k+1}, \cdots, \ not \ \ b_m.$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms. The disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while the conjunction of literals $b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m$ is the *body* of r. We denote by $At(r) = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ all the atoms occurring in r (both in the head and in the body). A rule having precisely one head literal (i.e., $n = 1$) is called a *normal rule*. If the body is empty (i.e., $k = m = 0$), it is called a *fact*, and we usually omit the $:-$ sign. A rule without head literals (i.e., $n = 0$) is usually referred to as an *integrity constraint*. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. A *not* -free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it. Given a ground program $\mathcal{P}$ and

[2] As an example, note that the semantics of a Prolog program changes if we modify the rule's order in the program, but this is not true in ASP where rule order is immaterial. Thus, meaningful unit tests in ASP can be obtained by collecting rules without taking care of their "position" in the program, the same does not hold for Prolog.

a set of atoms A, $Def_{\mathcal{P}}(A)$ denotes the set of rules that define A, i.e., all the rules $r \in \mathcal{P}$ such that some atom form A occurs in the head of r.

Semantics. Given a program $\mathcal{P}$, the *Herbrand Universe* $U_{\mathcal{P}}$ and the *Herbrand Base* $B_{\mathcal{P}}$ are defined in the usual way.

An *interpretation* for a program $\mathcal{P}$ is a subset I of $B_{\mathcal{P}}$. A ground positive literal a is true (resp., false) w.r.t. I if $a \in I$ (resp., $a \notin I$). A ground negative literal *not* a is true w.r.t. I if a is false w.r.t. I, otherwise *not* a is false w.r.t. I.

Given a program $\mathcal{P}$ its *ground instantiation* $Ground(\mathcal{P})$ is the set of (ground) rules obtained by applying to each rule $r \in \mathcal{P}$ all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. Given a program $\mathcal{P}$, its answer sets are defined using $Ground(\mathcal{P})$. Let r be a ground rule, r is *satisfied* (or true) w.r.t. I if some atom from the head of r is true in I and all body literals of r are true w.r.t. I. A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in Ground(\mathcal{P})$ is true w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if there is no model N for $\mathcal{P}$ such that N is a proper subset of M.

Given a *ground* program Q and an interpretation I, the (Gelfond-Lifschitz) *reduct* of Q w.r.t. I [1] is the positive ground program Q^I obtained from Q by (i) deleting all $r \in Q$ whose negative body is false w.r.t. I, and (ii) deleting the negative body from the remaining rules. An answer set of a program $\mathcal{P}$ is a model I of $\mathcal{P}$ such that I is a minimal model of $Ground(\mathcal{P})^I$. The set of all answer sets for $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$.

Example 1. Given the program $\mathcal{P} = \{a \vee b \;{:}\!-\; c.,\ b \;{:}\!-\; not\ a, not\ c.,\ a \vee c \;{:}\!-\; not\ b.\}$ and $I = \{b\}$, the reduct $\mathcal{P}^I$ is $\{a \vee b \;{:}\!-\; c., b.\}$. I is a model for $\mathcal{P}$ and is also a minimal model of $\mathcal{P}^I$, and for this reason it is an answer set of $\mathcal{P}$.

Splitting Sets. An important result presented in [41] allows to split an ASP program $\mathcal{P}$ in two modules that have a clear interface.

Definition 1 (Splitting Set). *A* splitting set *[41] of a program $\mathcal{P}$ is any subset U of atoms in $\mathcal{P}$ such that, for each rule $r \in \mathcal{P}$, if $H(r) \cap U \neq \emptyset$ then $At(r) \subset U$.*

In this case we say that U splits $\mathcal{P}$ in two distinct sub-programs $\mathcal{P}_b$ and $\mathcal{P}_t$. $\mathcal{P}_b$, called the *bottom* of $\mathcal{P}$ w.r.t. U, consists of all the rules that satisfy the property of Definition 1, and $\mathcal{P}_t$, called the *top* of $\mathcal{P}$ w.r.t. U, is composed of all the rules contained in $\mathcal{P} \setminus \mathcal{P}_b$. A consequence is the fact that all the atoms contained in the head of some rule of $\mathcal{P}_t$ are not contained in U.

Theorem 1 (Splitting Theorem). *Let a program $\mathcal{P}$ with a splitting set U of $\mathcal{P}$, let $\mathcal{P}_b$ the* bottom *of $\mathcal{P}$ w.r.t. U, and $\mathcal{P}_t$ the* top *of $\mathcal{P}$ w.r.t. U. A set M of atoms is a* consistent answer set *for $\mathcal{P}$ if and only if $M = X \cup Y$ where X is an answer set of $\mathcal{P}_b$ and Y is an answer set of $\mathcal{P}_t \cup X$.*

In other words, this theorem says that an answer set of $\mathcal{P}$ can be found by calculating an answer set X of $\mathcal{P}_b$ and giving X as input to $\mathcal{P}_t$ for calculating an answer set of $\mathcal{P}_t$.

The splitting theorem by Lifschitz and Turner [41] was also generalized to the non-ground case [42] by considering, as splitting sets, predicates of the program instead of ground atoms and taking in consideration the predicate dependency graph of the program.

Some Language Extension. The basic ASP language recalled above can be used to solve complex search problems, but it has been extended with constructs for specifying aggregate functions on sets of literals and solving optimization problems. These constructs are supported in DLV as well as in other ASP systems.

Aggregates. An *aggregate function* in DLV [43] is of the form $f(S)$, where S is a set term of the form $\{Vars : Conj\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard atoms, and f is an *aggregate function symbol.* The most common aggregate functions compute the number of terms, the sum of non-negative integers, and minimum/maximum term in a set. As an example, the following rule counts the number of true instances of predicate p: $numP(X)$:– $\#countX : p(X) = X$.

Weak constraints [3] allow us to express desiderata, that is, conditions that *should* be satisfied. A weak constraint starts by $\leftrightsquigarrow$. The informal meaning of a $\leftrightsquigarrow B$ is "try to falsify B," or "B should preferably be false." Intuitively, the semantics coincides with the answer sets minimizing the number of violated (unsatisfied) weak constraints.

3 Unit Testing in ASP

In software engineering, the task of testing and validating programs is a crucial part of the life cycle of the software development process. Software testing [44] is an activity aimed at evaluating the behavior of a program by verifying whether it produces the required output for a particular input. The goal of testing is not to provide means for establishing whether the program is totally correct. Conversely testing is a pragmatic and cheap way of finding errors by executing some tests.

One of the most diffused white-box[3] testing techniques is *unit testing.* The idea of unit testing is to assess an entire software by testing its subparts called *units*, which correspond to small testable parts of a program. In a software implemented using imperative object-oriented languages, unit testing corresponds

[3] A test conceived for verifying some functionality of an application without knowing the code internals is said to be a black-box test. A test conceived for verifying the behavior of a specific part of a program is called a white-box test. White box testing is an activity usually carried out by developers and it is a key component of agile software development [44].

to assessing separately portions of the code like class methods. Our testing methodology is inspired by the JUnit [35] framework. Given an ASP program the developer can select the program unit, specify one or more inputs, and assert a number of conditions on the expected output. The obtained test case specification can be run, and the assertions are automatically verified by calling an ASP solver and checking its output. We first introduce the notion of a unit test for a given ASP program, and then we present a language for specifying and running unit tests.

Definition 2 (Unit Test). *Given a program $\mathcal{P}$, a* unit test $\mathcal{T}$ *for $\mathcal{P}$ is a triple $\mathcal{T} = \langle \mathcal{U}, \mathcal{I}, \mathcal{A} \rangle$ where $\mathcal{U} \subseteq \mathcal{P}$ is the program unit to be tested, $\mathcal{I}$ the input program, and $\mathcal{A}$ is a set of assertions modeling properties that have to be verified by $ANS(\mathcal{U} \cup \mathcal{I})$. A test case $\mathcal{T}$* passes *if all the assertions in $\mathcal{A}$ are satisfied, and* fails *otherwise.*[4] *A* test suite *for a program $\mathcal{P}$ is a set of test cases.*

Basically, a unit test $\mathcal{T}$ focuses on a portion of the ASP program to be tested denoted by $\mathcal{U}$ and called program unit. The inputs for $\mathcal{U}$ are specified by an additional program $\mathcal{I}$, which, in the simplest scenario, can be made of input facts. Since $\mathcal{I}$ can be specified by means of an ASP program, ASP itself can be exploited for modeling different inputs in the same unit test. In order to specify a significant test case both $\mathcal{U}$ and $\mathcal{I}$ have to be specified with care. In particular, the program unit to be tested should "act as a module" so that its behavior can be effectively tested outside the original program. Moreover, $\mathcal{I}$ should only be exploited for specifying the test inputs for $\mathcal{U}$, and should not interfere with the usual "behavior" of $\mathcal{U}$. To this end, our framework provides a way for exploiting the notion of *splitting set* [41] both for specifying the unit program and checking that $\mathcal{I}$ only provides an input for $\mathcal{U}$ and not "interfering" with its execution. In addition, we also give the possibility to the programmer to enforce checking whether a test case satisfies the more fine grained notion of *DLP-Function* [36], so that more precise modularity of units can be exploited if needed.[5]

For example, our testing language that is described in the following allows to specify the program unit as the bottom program [41] of a given splitting set.

The output of the test case are the answer sets of $\mathcal{U} \cup \mathcal{I}$, and $\mathcal{A}$ contains the specification of a number of properties that have to be satisfied to pass the test. For example, one might require that a given ground atom a is contained in all answer sets $\mathcal{U} \cup \mathcal{I}$, or that the unit program is expected to have a given number of answer sets.

[4] Note that this definition of test case is more general than the one of [33]. Indeed, all test cases in [33] are such that: $\mathcal{U} = \mathcal{P}$, $\mathcal{I}$ is the set of ground inputs, and $\mathcal{A}$ contains the assertions stating that the set of expected outputs is in $ANS(\mathcal{U} \cup \mathcal{I})$.

[5] Note that the notion of DLP-Function does not allow to split the program in a unique deterministic way, thus we pragmatically decided to support this notion only for checking units.

Testing Language. Test cases are specified in our framework by means of text files. A test file can be written according to the following grammar[6]:

```
1 : invocation("invocationName" [ ,"solverPath", "options" ]?);
2 : [ [ input("program"); ] | [ inputFile("file"); ] ]*
3 : [
4 : testCaseName([ SELECTED_RULES | SPLIT_PROGRAM | PROGRAM ])
6 : {
7 : [newOptions("options");]?
8 : [ [ input("program"); ] | [ inputFile("file"); ] ]*
9 : [ [ excludeInput("program"); ]
10 : | [ excludeInputFile("file"); ] ]*
11 : [
12 : [ filter | pfilter | nfilter ]
13 : [ [ (predicateName [,predicateName ]* ) ]
14 : | [SELECTED_RULES] ] ;
15 : ]?
16 : [checkModularity( [ SPLITTING_SET | DLP_FUNCTION ] [,"atoms" ]? ); ]*
17 : [ [ selectRule("ruleName"); ]
18 : | [ selectRulesWithPredicateInHead("predicateName"); ]
19 : | [ selectRulesWithPredicateInBody("predicateName"); ]
20 : | [ selectRulesWithPredicateInPositiveBody("predicateName"); ]
21 : | [ selectRulesWithPredicateInNegativeBody("predicateName"); ]
22 : | [ selectRulesWithPredicateInAggregates("predicateName"); ]
23 : ]*
24 : [ [ assertName( [ intnumber, ]? [ [ "atoms" ] | [ "constraint" ] ]?); ]
25 : | [ assertBestModelCost(intcost [, intlevel ]? ); ] ]*
26 : }
27 : ]*
28 : [ [ assertName( [ intnumber, ]? [ [ "atoms" ] | [ "constraint" ] ]? ); ]
29 : | [ assertBestModelCost(intcost [, intlevel ]? ); ] ]*
```

A test file might contain a single test or a test suite (a set of tests) including several test cases for the same program to be tested. Each test case includes one or more assertions on the results.

The *invocation* statement (line 1) sets the global invocation settings, that apply to all tests specified in the same file (name, solver, and execution options). The invocation name might correspond to an *ASPIDE* run configuration (see Sect. 4.1). In this latter case, both the solver path and invocation options are automatically imported from the corresponding run configuration.

The user can specify the program to be tested by writing one or more *input* and *inputFile* statements (line 2). The first kind of statement allows one for writing the program to be tested; the second statement indicates a file that contains some input program in ASP format.

A unit test declaration (line 4 and 5) is composed of a name and an optional parameter that allows one to choose if the unit program corresponds to the entire program (option PROGRAM), or is made of exactly the selected rules (option SELECTED_RULES), or if the unit program to consider corresponds to the splitting set containing the atoms occurring on the selected rules (option SPLIT_PROGRAM). In this last case, the "interface" between two splitting sets can be tested (e.g., one can assert some expected properties on the candidates produced by the guessing part of a program by excluding the effect of some constraints in the checking part). Note that, this feature of the language allows

[6] Non-terminals are in bold face, token specifications are omitted for simplicity.

for specifying in a declarative way which parts of the tested program has to be considered in a unit test.

The user can specify particular solver options (line 7), as well as certain inputs (line 8) which are valid in a given unit test. Moreover, global inputs of the test suite can be excluded by exploiting *excludeInput* and *excludeInputFile* statements (lines 9 and 10). The optional statements *filter*, *pfilter* and *nfilter* (lines 12, 13, and 14) are used to filter out output predicates from the test results (i.e., specified predicate names are filtered out from the results when the assertion is executed).[7]

The statement *checkModularity* (line 16) can be added to a unit test to require to verify that the selected rules compose either a correct DLP-Function [36] or correspond to splitting set for the tested program. The list of atoms, that can be specified by the user, represent either the input signature if the rules define a DLP-Function (that can be joined with the remaining part of the program for a given choice of input/output) or the splitting set if the rules define a split of the program.[8]

The statement *selectRule* (line 17) allows one for selecting rules among the ones composing the tested program. A rule r to be selected must be identified by a name, which is expected to be specified in the input program in a comment appearing in the row immediately preceding r. Actually, in the implementation rule names are added automatically as comments. The selection of the rules can be done also by using predicate names; in particular the statements (lines 18/22) allow one to select rules where a given predicate appears in the head, in the body, in the positive body, in the negative body and in some aggregate atom. Note that, this feature is very useful for selecting in an easy way the rules composing the definition of a predicate.

The expected output of a test case is expressed in terms of assertion statements (lines 24/29). The assertions supported by the language are:

– *assertTrue*(*"atomList"*)/*assertCautiouslyTrue*(*"atomList"*). Asserts that all atoms of the atom list must be true in any answer sets;
– *assertBravelyTrue*(*"atomList"*). Asserts that all atoms of the atom list must be true in at least one answer set;
– *assertTrueIn*(*number, "atomList"*). Asserts that all atoms of the atom list must be true in exactly *number* answer sets;
– *assertTrueInAtLeast*(*number, "atomList"*). Asserts that all atoms of the atom list must be true in at least *number* answer sets;
– *assertTrueInAtMost*(*number, "atomList"*). Asserts that all atoms of the atom list must be true in at most *number* answer sets;
– *assertConstraint*(*":-constraint."*). Asserts that all answer sets must satisfy the specified constraint;

[7] *pfilter* excludes the strongly negated ones, while *nfilter* has opposite behavior.

[8] DLP-Functions offer a fine way of decomposing a program in modules that can be joined together to construct $\mathcal{P}$. The interested reader is referred to [36] for a formal definition.

- *assertConstraintIn*(*number, ":-constraint."*). Asserts that exactly *number* answer sets must satisfy the specified constraint;
- *assertConstraintInAtLeast*(*number, ":-constraint."*). Asserts that at least *number* answer sets must satisfy the specified constraint;
- *assertConstraintInAtMost*(*number, ":-constraint."*). Asserts that at most *number* answer sets must satisfy the specified constraint;
- *assertBestModelCost*(*intcost*) and *assertBestModelCost*(*intcost, intlevel*). In case of execution of programs with weak constraints, they assert that the cost of the best model with level *intlevel* must be *intcost*;
- *assertAnswerSetsNumber*(*number*). Asserts that the tested program is expected to generate exactly *number* answer sets.
- *assertNoAnswerSet.* Asserts that the tests is expected to have no answer set.

together with the corresponding negative assertions: *assertFalse*, *assertCautiouslyFalse*, *assertBravelyFalse*, *assertFalseIn*, *assertFalseInAtLeast*, *assertFalseInAtMost.*

The *atomList* specifies a list of atoms that can be ground or non-ground; in the case of non-ground atoms the assertion is true if some ground instance matches in some/all answer sets. Assertions can be global (line 20-21) or local to a single test (line 16-17). Note that, the set of supported assertions is redundant; actually, this is on purpose, indeed having different possibilities for asserting the same property eases the task of test case specification for the programmer, who can specify its requirements in the way she prefers.

In the following we report a test case example.

Test case example. The maximum clique is a classical hard problem in graph theory requiring to find the largest clique (i.e., a complete subgraph of maximal size) in an undirected graph. Suppose that the graph *G* is specified by using facts over predicates *node* (unary) and *edge* (binary), then the following program solves the problem.

```
% Guess the clique
%@name = r1
inClique(X1) v outClique(X1) :- node(X1).
% Order edges in order to reduce checks
%@name = r2
uedge(X1,X2) :- edge(X1,X2), X1 < X2.
%@name = r3
uedge(X2,X1) :- edge(X1,X2), X2 < X1.
% Ensure property.
%@name = r4
:- inClique(X1), inClique(X2), not uedge(X1,X2), X1 < X2.
%@name = r5
:~ outClique(X2).
```

The disjunctive rule ($\mathtt{r}_1$) guesses a subset *S* of the nodes to be in the clique, while the rest of the program checks whether *S* constitutes a clique, and the weak constraint ($\mathtt{r}_5$) maximizes the size of *S* (since it prefers interpretations in which the number of true `outClique` instances is minimized). Here, an auxiliary

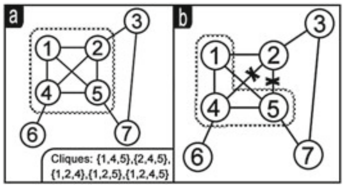


Fig. 1. Input graphs.

predicate *uedge* exploits an ordering for reducing the time spent in checking. Suppose that the encoding is stored in a file named *clique.dl* and suppose also that the graph instance, composed of facts { *node(1). node(2). node(3). node(4). node(5). node(6). node(7). edge(1,2). edge(2,3). edge(2,4). edge(1,4). edge(1,5). edge(4,5). edge(2,5). edge(4,6). edge(5,7). edge(3,7).*}, is stored in the file named *graphInstance.dl* (the corresponding graph is depicted in Fig. 1a). The following is a simple test suite specification for the above-reported ASP program:

```
invocation("MaximalClique", "/usr/bin/dlv", "");
inputFile("clique.dl");
inputFile("graphInstance.dl");
maximalClique() {
assertBestModelCost(3);
}
constraintsOnCliques() {
excludeInput(":~ outClique(X2).");
assertConstraintInAtLeast(1,":- inClique(1),
          inClique(4).");
assertConstraintIn(5,":- #count{ X1: inClique(X1) } < 3.");
}
checkNodeOrdering(SELECTED_RULES) {
inputFile("graphInstance.dl");
selectRule("r2");
selectRule("r3");
assertFalse("uedge(2,1).");
}
guessClique(SPLIT_PROGRAM) {
selectRule("r1");
assertFalseInAtMost(1,"inClique(X).");
assertBravelyTrue("inClique(X).");
}
```

Here, we first set the invocation parameters by indicating DLV as solver, then we specify the file to be tested *clique.dl* and the input file *graphInstance.dl*, by exploiting a global input statement. Then, we add the test case *maximalClique*, in which we assert that the best model is expected to have a cost (i.e., the number of weak constraint violations corresponding to the vertexes out of the clique) of 3 (*assertBestModelCost(3)*). In the second test case, named *constraintsOnCliques*, we require that (i) vertexes 1 and 4 do not belong to at least one clique, and (ii) for exactly five answer sets the size of the corresponding clique is greater than 2. (The weak constraint is removed to ensure the computation of all cliques by DLV.)

In the third test case, named *checkNodeOrdering*, we select rules r_2 and r_3, and we require to test selected rules in isolation, discarding all the other statements of the input. We are still interested in considering ground facts that are included locally (i.e., we include the file *graphInstance.dl*). In this case we assert that *uedge(2,1)* is false, since edges should be ordered by rules r_2 and r_3.

Test case *guessClique* is run in *SPLIT_PROGRAM* modality, which requires to test the sub-program containing all the rules belonging to the splitting set corresponding to the selection (i.e., {*inClique, outClique, node*}). In this test case the sub-program that we are testing is composed of the disjunctive rule and by the facts of predicate *node* only. Here we require that there is at most one answer set modeling the empty clique, and there is at least one answer set modeling a non-empty clique.

In a language expressive and concise like ASP also testing small modules (and even one single rule) in isolation may help the programmer to detect bugs. Unit test cases may help the programmer in fixing bugs as well as understanding/correcting requirements. It might happen that both test cases and programs may change if the requirements (i.e., the problem specification) are not well understood, but, in most cases, buggy rules (referenced by name) can be updated without the need of updating the corresponding unit tests.

As an example, suppose that the first time we have written the ASP program of our example we would have introduced a bug, and in particular a typo on the guessing rule:

```
%@name = r1
inClique(X1) v outClique(X1) :- nod(X1).
```

Note that rule *r1* has a bug since, in the body, *node* is written without the final *e*. When the test case *guessClique* is run it fails since the extension of *nod* (without *e*) is empty.

After fixing the rule *r1* the test case can be now re-run and will pass. In the above example, selecting one rule and testing it in isolation helped to fix the program without the need for updating the test case. Not that, the possibility of testing parts (i.e., units) of the original program (up to one single rule at time) helps in detecting by exploiting simple tests which specific part of the program does not behave as expected.

The test file described in this section can be created and executed in *ASPIDE* as described in the next section. The results can be inspected using the results window that marks the atoms that have participated to the passing/failure of test cases. Actually, in our IDE test cases can be also created graphically, i.e., without the need of writing by hand (textual) test case specifications. This can be done, starting from the results of an execution, by selecting and marking wanted and unwanted results in an intuitive and "by example" test case creation interface. This is another distinguishing feature of the approach of ASPIDE to unit testing, which is described in the following.

4 Implementation in *ASPIDE*

In this section, after overviewing the *ASPIDE* [32] development environment, we describe the graphic tools conceived for developing and running test cases.

4.1 *ASPIDE*

ASPIDE is an Integrated Development Environment (IDE) for ASP, which features a rich *editing tool* with a collection of user-friendly *graphical tools* for ASP program development. *ASPIDE* is inspired by Eclipse, one of the most diffused programming environments. The main features of *ASPIDE* are described in the following.

Workspace management. The system allows one to organize ASP programs in projects, which are collected in the workspace. Workspace components, such as projects and files, are graphically represented by a tree structure called *Workspace Explorer*.

Advanced text editor. The editing of ASP files is simplified by an advanced text editor. Currently, the system is able to load and store ASP programs in the syntax of the ASP system DLV [45], and supports the `ASPCore` language profile employed in the ASP System Competition 2011 [46]. *ASPIDE* can also manage *TYP files* specifying a mapping between program predicates and database tables in the DLV^{DB} syntax [47]. Besides the core functionality that basic text editors offer (like code line numbering, find/replace, undo/redo, copy/paste, etc.), *ASPIDE* offers other advanced functionalities, like: *Automatic completion*, *Dynamic code templates*, *Quick fix*, and *Refactoring*. Indeed, the system is able to complete (on request) predicate names, as well as variable names. Predicate names are both learned while writing, and extracted from the files belonging to the same project; variables are suggested by taking into account the rule we are currently writing. When several possible alternatives for completion are available the system shows a pop-up dialog. Moreover, the writing of repeated programming patterns (like transitive closure or disjunctive rules for guessing the search space) is assisted by advanced auto-completion with code templates, which can generate several rules at once according to a known pattern. Note that code templates can also be user defined by writing DLT [48] files. The refactoring tool allows one to modify in a guided way, among others, predicate names and variables (e.g., variable renaming in a rule is done by considering bindings of variables, so that variables/predicates/strings occurring in other expressions remain unchanged). Reported errors or warnings can be automatically fixed by selecting (on request) one of the system's suggested quick fixes, which automatically change the affected part of code.

Outline navigation. *ASPIDE* creates an outline view which graphically represents program elements. Each item in the outline can be used to quickly access the corresponding line of code (a very useful feature when dealing with long files), and also provides a graphical support for building rules in the visual editor (see below).

Dynamic code checking and error highlighting. Syntax errors and relevant conditions (like safety) are checked *while typing programs*: portions of code containing errors or warnings are immediately highlighted. Note that the checker considers the entire project; for example, the system issues a warning if there are predicates defined with different arity in different files. This condition is usually revealed only when programs divided in multiple files are run together.

Dependency graph. The system is able to display several variants of the dependency graph associated to a program (e.g., depending on whether both positive and negative dependencies are considered).

Debugger and profiler. Semantic error detection as well as code optimization can be done by exploiting graphic tools. In particular, we developed a graphical user interface for embedding in *ASPIDE* the debugging tool *spock* [24] (properly extended for dealing with the syntax of the DLV system). Regarding the profiler, we have fully embedded the graphical interface presented in [49].

Unit testing. The user can define unit tests and verify the behavior of program units. The testing tools are described in detail in the following sections.

Configuration of the execution. This feature allows one to configure and manage input programs and execution options (called *run configurations*).

Presentation of results. The output of the program (either answer sets, or query results) are visualized in a tabular representation or in a text-based console. The result of the execution can be also saved in text files for subsequent analysis.

Visual editor. The users can *draw* logic programs by exploiting a full graphical environment that offers a QBE-like tool for building logic rules [50]. The user can switch, every time he needs, from the text editor to the visual one (and vice-versa) thanks to a reverse-engineering mechanism from text to graphical format.

Interaction with databases. Interaction with external databases is useful in several applications (e.g., [12]). *ASPIDE* provides a fully graphical import/export tool that automatically generates mappings by following the DLV^{DB} TYP file specifications [47]. Text editing of TYP mappings is also assisted by syntax coloring and auto-completion. Database oriented applications can exploit DLV^{DB} as solver.

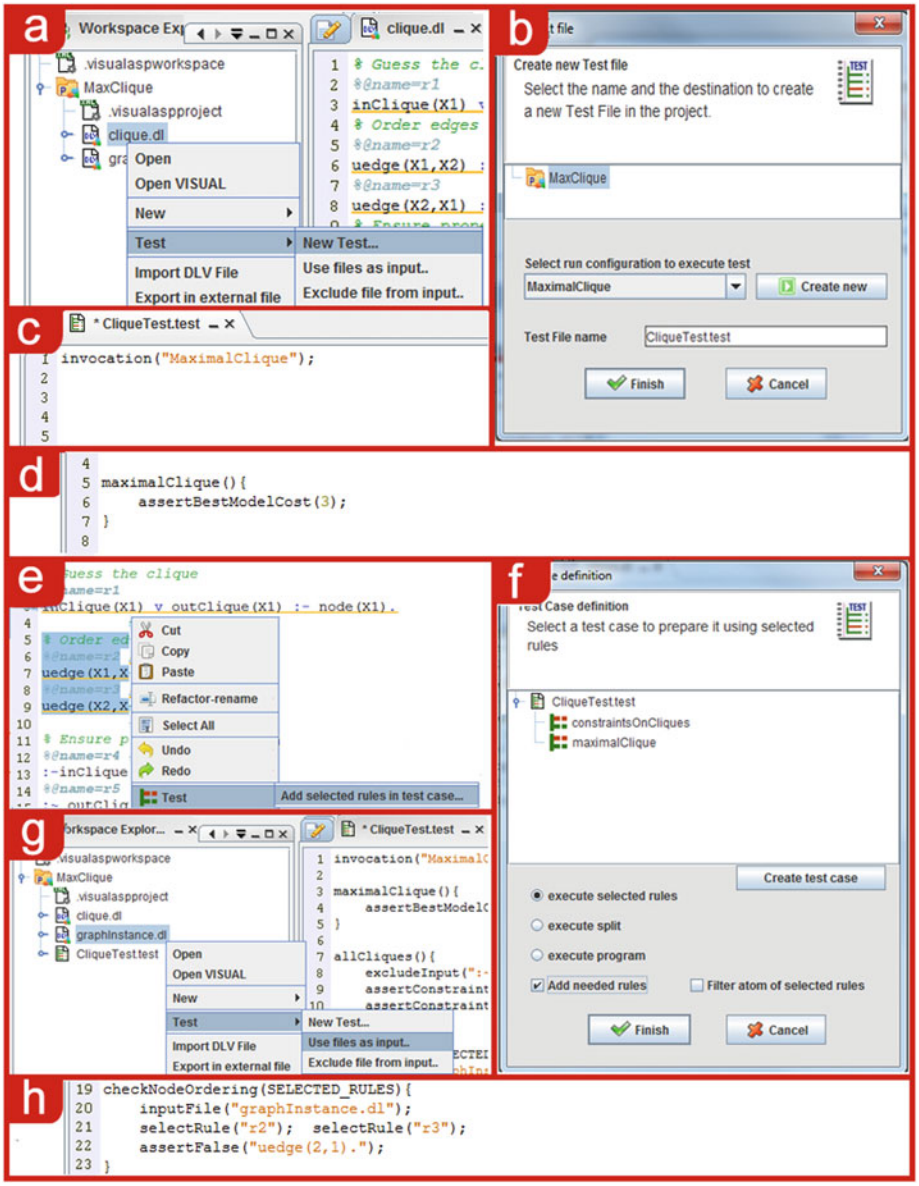
Implementation and availability. *ASPIDE* is written in Java and runs on the most diffused operating systems (Microsoft Windows, Linux, and Mac OS) and can connect to any database supporting Java DataBase Connectivity (JDBC).

For a more detailed description of *ASPIDE*, as well as for a complete comparison with competing tools, we refer the reader to [32] and to the online manual published in the system web site: http://www.mat.unical.it/ricca/aspide.

4.2 Unit Testing in *ASPIDE*

We have implemented an extension of *ASPIDE* that allows the user to create and execute test suites specified in the language presented in Sect. 3. The user can both manually edit test case files, as well as he or she can create test cases by exploiting a number of visual tools. In order to provide a description that immediately gives an idea about the capabilities of the visual testing interface of *ASPIDE* and it is easy to use, we describe step by step how to implement the example illustrated in the previous section.[9] Suppose that we have created in *ASPIDE* a project named MaxClique, which contains the files *clique.dl* and *graphInstance.dl* storing the encoding of the maximal clique problem and a graph instance, respectively. Moreover we assume that both input files are included in a run configuration named *MaximalClique*, and we assume that the DLV system is the solver of choice in this run configuration. Since the file that we want to test in our example is *clique.dl*, we select it in the *workspace explorer*, then we click the right button of the mouse and select *New Test* from the popup menu (Fig. 2a). The system shows the test creation dialog (Fig. 2b), which allows one for both setting the name of the test file and selecting a previously-defined run configuration (storing execution options and input files). By clicking on the *Finish* button, the new test file is created (see Fig. 2c) where a statement regarding the selected run configuration is added automatically. We add the first unit test (called *maximalClique*) by exploiting the text editor (see Fig. 2d), whereas we build the remaining ones (working on some selected rules) by exploiting the logic program editor. After opening the *clique.dl* file, we select rules r_2 and r_3 inside the text editor, we right-click on them and we select *Add selected rules in test case* from the menu item *Test* of the popup menu (Fig. 2e). The system opens a dialog window where we indicate the test file in which we want to add the new test case (Fig. 2f). We click on the *Create test case* button. The system will ask for the name of the new test case and we write *guessClique*. After that, on the window, we select the option *execute selected rules* and click on the *Finish* button. The system will add the test case *guessClique* filled with the *selectRule* statements indicating the selected rules. To add project files as input of the test case, we select them from the *workspace explorer* and click on *Use file as input* in the menu item *Test* (Fig. 2g). We complete the test case specification by adding the assertion, thus the test created up to now is shown in Fig. 2h. Following an analogous procedure we create the remaining test cases (see Fig. 3a). To execute

[9] Space constraints prevent us from providing a complete description of all the usage scenarios and commands; rather, we focus on the testing interface presenting a specific example.



```
% Guess the c
%@name=r1
inClique(X1) v
% Order edges
%@name=r2
uedge(X1,X2) :
%@name=r3
uedge(X2,X1) :
```

Create new Test file

Select the name and the destination to create a new Test File in the project.

Select run configuration to execute test: MaximalClique

Test File name: CliqueTest.test

```
invocation("MaximalClique");
```

```

maximalClique(){
     assertBestModelCost(3);
}

```

```
uess the clique
ame=r1
nClique(X1) v outClique(X1) :- node(X1).

% Order ed
%@name=r2
uedge(X1,X
%@name=r3
uedge(X2,X

% Ensure p
%@name=r4
:-inClique
%@name=r5
```

Test Case definition

Select a test case to prepare it using selected rules

[✓ execute selected rules] execute split execute program

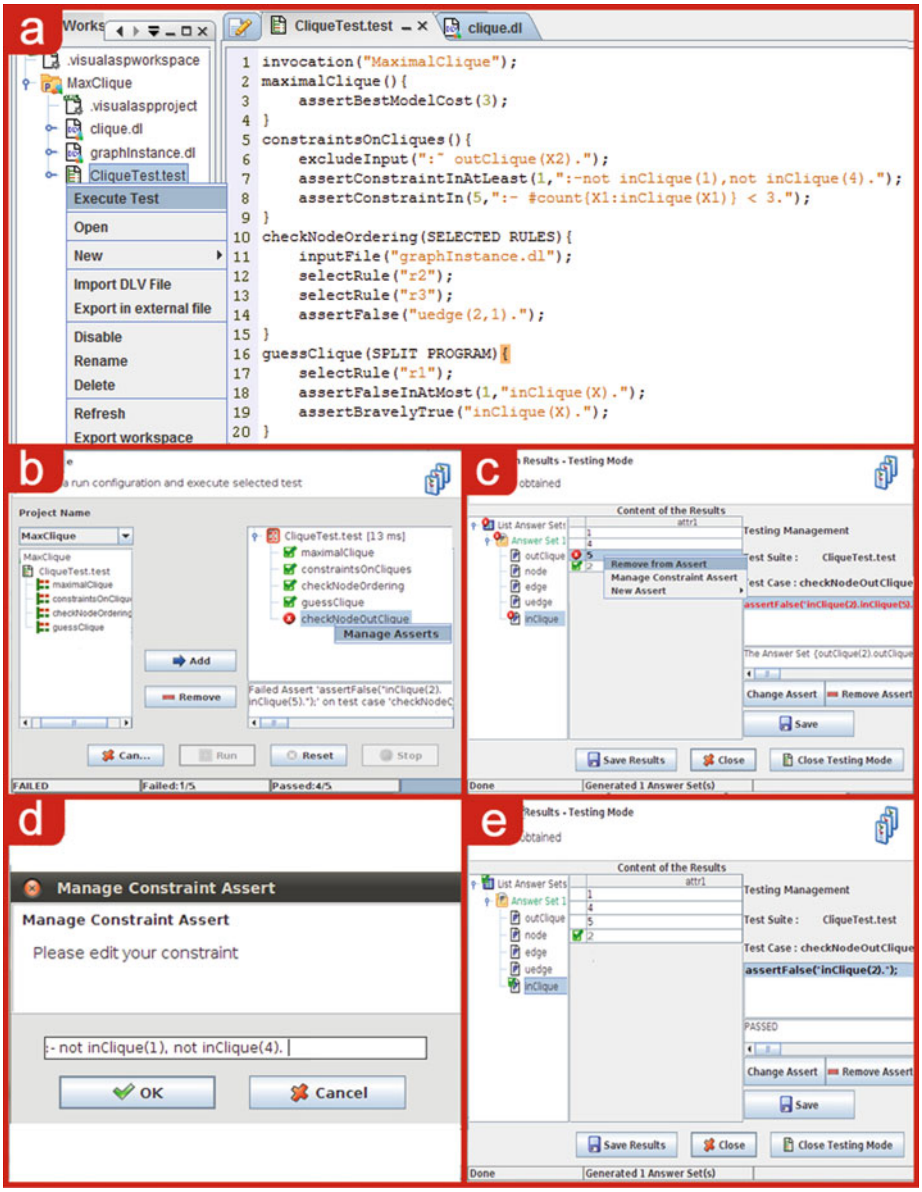
[✓ Add needed rules] Filter atom of selected rules

```
invocation("MaximalC

maximalClique(){
     assertBestModelC
}

allCliques(){
     excludeInput(":-
     assertConstraint
     assertConstraint
```

```
checkNodeOrdering(SELECTED_RULES){
     inputFile("graphInstance.dl");
     selectRule("r2");  selectRule("r3");
     assertFalse("uedge(2,1).");
}
```

Fig. 2. Test case creation.

our tests, we right-click on the test file and select *Execute Test*. The *Test Execution Dialog* appears and the results are shown to the programmer (see Fig. 3b). Failing tests are indicated by a red icon, while green icons indicate passing tests. At this point we add the following additional test:



```
invocation("MaximalClique");
maximalClique(){
    assertBestModelCost(3);
}
constraintsOnCliques(){
    excludeInput(":- outClique(X2).");
    assertConstraintInAtLeast(1,":-not inClique(1),not inClique(4).");
    assertConstraintIn(5,":- #count{X1:inClique(X1)} < 3.");
}
checkNodeOrdering(SELECTED RULES){
    inputFile("graphInstance.dl");
    selectRule("r2");
    selectRule("r3");
    assertFalse("uedge(2,1).");
}
guessClique(SPLIT PROGRAM){
    selectRule("r1");
    assertFalseInAtMost(1,"inClique(X).");
    assertBravelyTrue("inClique(X).");
}
```

b: a run configuration and execute selected test

CliqueTest.test [13 ms]

Failed Assert 'assertFalse("inClique(2). inClique(5).");' on test case 'checkNodeC

FAILED | Failed:1/5 | Passed:4/5

c: Test Suite : CliqueTest.test
Test Case : checkNodeOutClique
assertFalse("inClique(2).inClique(5).
The Answer Set {outClique(2).outClique

attr1: 1, 4, 5, 2

Done | Generated 1 Answer Set(s)

d: Manage Constraint Assert
Please edit your constraint
:- not inClique(1), not inClique(4).

e: Test Suite : CliqueTest.test
Test Case : checkNodeOutClique
assertFalse("inClique(2).");
PASSED

attr1: 1, 4, 5, 2

Done | Generated 1 Answer Set(s)

Fig. 3. Test case execution and assertion management.

```
checkNodeOutClique() {
excludeInput("edge(2,4).edge(2,5).");
assertFalse("inClique(2). inClique(5).");
}
```

This additional test (purposely) fails, this can be easily seen by looking at Fig. 1b. The reason for this failure is indicated (see Fig. 3b) in the test execution dialog. In order to know which literals of the solution do not satisfy the assertion, we right-click on the failed test and select *Manage Asserts* from the menu. A dialog showing the outputs of the test appears where, in particular, predicates and literals matching correctly the assertions are marked in green, whereas the ones violating the assertion are marked in red (gray icons may appear to indicate missing literals which are expected to be in the solution). In our example, the assertion is *assertFalse("inClique(2). inClique(5).")*; however, in our instance, node 5 is contained in the maximal clique composed of nodes *1, 4, 5*; this is the reason for the failing test. Assertions can be modified graphically, and, in this case, we act directly on the result window (Fig. 3c). We remove the node 5 from the assertion by selecting it. Moreover we right-click on the instance of *inClique* that specifies the node 5 and we select *Remove from Assert*. The atom *node(5)* will be removed from the assertion and the window will be refreshed marking the test case as passed (see Fig. 3e). The same window can be used to manage constraint assertions. In particular, by clicking on *Manage Constraint Assert* of the popup menu, a window appears that allows the user to set/edit constraints (see Fig. 3d).

5 Conclusion

This paper presents a pragmatic environment for testing ASP programs. In particular, we propose a new language, inspired by the JUnit [35] framework, for specifying and running *unit tests* on ASP programs. The testing language is general and suits both the DLV [45] and clasp [51] ASP dialects. The testing language has been implemented in *ASPIDE* together with some graphic tools for easing both the development of tests and the analysis of test execution.

As far as future work is concerned, we plan to extend *ASPIDE* by improving/ introducing additional dynamic editing instruments, and graphic tools like VIDEAS [52]. Moreover, we plan to further improve the testing tool by supporting (semi)automatic test case generation based on the structural testing techniques proposed in [33,34]. We have contacted the authors of [33,34] for including them in our system. These issues are, indeed, orthogonal to our approach (unit tests are usually defined by programmers), but since our definition is more general, this integration requires to implement/adapt the existing a test case generators in such a way that they produce specifications encoded in our test case specification language.

Acknowledgments. This work has been partially supported by the Calabrian Region under PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) project DLVSYSTEM approved in BURC n. 20 parte III del 15/05/2009 - DR n. 7373 del 06/05/2009.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. NGC **9**, 365–385 (1991)
2. Lifschitz, V.: Answer set planning. In: ICLP'99, pp. 23–37. MIT Press (1999)
3. Buccafurri, F., Leone, N., Rullo, P.: Enhancing disjunctive datalog by constraints. IEEE TKDE **12**(5), 845–860 (2000)
4. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. AI **175**(1), 278–298 (2011). Special Issue: John McCarthy's Legacy
5. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM TODS **22**(3), 364–418 (1997)
6. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 3–17. Springer, Heidelberg (2007)
7. Balduccini, M., Gelfond, M., Watson, R., Nogueira, M.: The USA-Advisor: a case study in answer set planning. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 439–442. Springer, Heidelberg (2001)
8. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 257–279. Kluwer, Dordrecht (2000)
9. Baral, C., Uyan, C.: Declarative specification and solution of combinatorial auctions using logic programming. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 186–199. Springer, Heidelberg (2001)
10. Franconi, E., Palma, A.L., Leone, N., Perri, S.: Census data repair: a challenging application of disjunctive logic programming. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 561–578. Springer, Heidelberg (2001)
11. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog decision support system for the space shuttle. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 169–183. Springer, Heidelberg (2001)
12. Leone, N., et al.: The INFOMIX system for advanced integration of incomplete and inconsistent data. In: SIGMOD 2005, pp. 915–917. ACM Press (2005)
13. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
14. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV applications for knowledge management. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 591–597. Springer, Heidelberg (2009)
15. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. TPLP **11**(2–3), 323–360 (2011)
16. Gebser, M., Kaminski, R., Schaub, T.: aspcud: a linux package configuration tool based on answer set programming. In: LoCoCo, pp. 12–25 (2011)
17. Ricca, F., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: Logic-based system for e-Tourism. FI **105**(1–2), 35–55 (2010)
18. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the Gioia-Tauro seaport. TPLP. http://dx.doi.org/10.1017/S147106841100007X (2011)
19. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)

20. Calimeri, F., et al.: The third answer set programming competition: preliminary report of the system competition track. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 388–403. Springer, Heidelberg (2011)
21. Dovier, A., Erdem, E.: Report on application session @lpnmr09. http://www.cs.nmsu.edu/ALP/2010/03/report-on-application-session-lpnmr09/ (2009)
22. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: SEA 07, pp. 86–100 (2007)
23. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: an AnsProlog* environment. In: SEA 07, pp. 101–115 (2007)
24. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: That is illogical captain! The debugging support tool spock for answer-set programs: system description. In: SEA 07, pp. 71–85 (2007)
25. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In: ASP'05, Bath, UK (2005)
26. El-Khatib, O., Pontelli, E., Son, T.C.: Justification and debugging of answer set programs in ASP. In: Proceedings of Automated Debugging, California, USA. ACM (2005)
27. Oetsch, J., Pührer, J., Tompits, H.: Catching the ouroboros: on debugging non-ground answer-set programs. In: Proceedings of the ICLP'10 (2010)
28. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
29. De Vos, M., Schaub, T., (eds.): SEA'07: Software Engineering for Answer Set Programming, vol. 281. CEUR. http://CEUR-WS.org/Vol-281/ (2007)
30. De Vos, M., Schaub, T., (eds.): SEA'09: Software Engineering for Answer Set Programming, vol. 546. CEUR. http://CEUR-WS.org/Vol-546/ (2009)
31. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based system for enterprise ontologies. JLC **19**(4), 643–670 (2009)
32. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: integrated development environment for answer set programming. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 317–330. Springer, Heidelberg (2011)
33. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proceeding of the conference on ECAI 2010: 19th European Conference on Artificial Intelligence, pp. 951–956. IOS Press, Amsterdam (2010)
34. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: Random vs. structure-based testing of answer-set programs: an experimental comparison. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 242–247. Springer, Heidelberg (2011)
35. JUnit.org community: JUnit, Resources for Test Driven Development. http://www.junit.org/
36. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J. Artif. Intell. Res. (JAIR) **35**, 813–857 (2009)
37. Jack, O.: Software Testing for Conventional and Logic Programming. Walter de Gruyter & Co., Hawthorne (1996)
38. Wielemaker, J.: Prolog Unit Tests. http://www.swi-prolog.org/pldoc/package/plunit.html
39. Cancinos, C.: Prolog Development Tools - ProDT. http://prodevtools.sourceforge.net
40. Gelfond, M., Leone, N.: Logic programming and knowledge representation - the A-Prolog perspective. AI **138**(1–2), 3–38 (2002)

41. Lifschitz, V., Turner, H.: Splitting a logic program. In: ICLP'94, pp. 23–37. MIT Press (1994)
42. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artif. Intell. **172**, 1495–1539 (2008)
43. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV. In: IJCAI 2003, Acapulco. Mexico, pp. 847–852 (2003)
44. Sommerville, I.: Software Engineering. Addison-Wesley, Harlow (2004)
45. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TCL **7**(3), 499–562 (2006)
46. Calimeri, F., Ianni, G., Ricca, F.: The third answer set programming system competition. https://www.mat.unical.it/aspcomp2011/ (2011)
47. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **8**, 129–165 (2008)
48. Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C.: Answer set programming with templates. In: ASP'03, Messina, Italy, pp. 239–252. http://CEUR-WS.org/Vol-78/ (2003)
49. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proceedings of SEA'09, Potsdam, Germany (2009)
50. Febbraro, O., Reale, K., Ricca, F.: A visual interface for drawing ASP programs. In: Proceedings of CILC2010, Rende(CS), Italy (2010)
51. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007, pp. 386–392 (2007)
52. Oetsch, J., Pührer, J., Seidl, M., Tompits, H., Zwickl, P.: VIDEAS: a development tool for answer-set programs based on model-driven engineering technology. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 382–387. Springer, Heidelberg (2011)

Author Index