Answer Set Programming
and
Other Computing Paradigms

Fifth International Workshop
September 4th 2012, Budapest, Hungary

Michael Fink
Yuliya Lierler (Eds.)
Preface

Since its introduction in the late 1980s, answer set programming (ASP) has been widely applied to various knowledge-intensive tasks and combinatorial search problems. ASP was found to be closely related to Boolean satisfiability, which has led to a new method of computing answer sets using SAT solvers and techniques adapted from SAT. While this has been the most studied relationship which is currently extended towards satisfiability modulo theories (SMT), the relationship of ASP to other computing paradigms, such as constraint satisfaction, quantified boolean formulas (QBF), first-order logic (FOL), or FO(ID) logic is also the subject of active research. New methods of computing answer sets are being developed based on the relation between ASP and other paradigms, such as the use of pseudo-Boolean solvers, QBF solvers, FOL theorem provers, and CLP systems.

Furthermore, the practical applications of ASP also foster work on multi-paradigm problem-solving, and in particular language and solver integration. The most prominent examples in this area currently are the integration of ASP with description logics (in the realm of the Semantic Web) and constraint satisfaction.

This volume contains the papers presented at the fifth workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2012) held on September 4th, 2012 in Budapest, co-located with the 28th International Conference on Logic Programming (ICLP 2012). It thus continues a series of previous events co-located with ICLP, aiming at facilitating the discussion about crossing the boundaries of current ASP techniques in theory, solving, and applications, in combination with or inspired by other computing paradigms.

Eleven papers have been accepted for presentation and constitute the technical contributions to this proceedings. They cover a wide range of research topics including theoretical aspects such as generalizing foundational principles of answer set semantics to more expressive settings, as well as practical aspects such as applications of the answer set programming and related computing paradigms in various domains. Each submission was reviewed by three program committee members in a blind review process. On top of that the program featured two invited talks by Marc Denecker (Catholic University of Leuven, Belgium) and Wolfgang Faber (University of Calabria, Italy).

We would like to take this opportunity to thank all authors, reviewers, speakers, and participants for making this workshop happen. We appreciated the support of the ICLP General Chair Péter Szeredi and the ICLP Workshop Chair Mats Carlsson. Moreover, we are grateful to Tung Tran for his help in creating this proceedings, and eventually, we like to acknowledge the EasyChair system which we used for organizing the submission and review processes.

August 2012

Michael Fink
Yuliya Lierler
(Organizers ASPOCP 2012)
Organisation

Executive Committee

Workshop Chairs: Michael Fink  
(Vienna University of Technology, Austria)

Yuliya Lierler  
(University of Nebraska at Omaha, USA)

Programme Committee

Marcello Balduccini  
Kodak Research Laboratories

Gerhard Brewka  
Leipzig University

Pedro Cabalar  
University of Coruña

Sandeep Chintabathina  
University of Arkansas at Pine Bluff

Wolfgang Faber  
University of Calabria

Michael Fink  
Vienna University of Technology

Martin Gebser  
University of Potsdam

Giovambattista Ianni  
University of Calabria

Joohyung Lee  
Arizona State University

Joao Leite  
Universidade Nova de Lisboa

Yuliya Lierler  
University of Kentucky

Vladimir Lifschitz  
University of Texas at Austin

Marco Maratea  
University of Genova

Alessandro Mosca  
Free University of Bozen-Bolzano

Emilia Oikarinen  
Aalto University

David Pearce  
Universidad Rey Juan Carlos

Axel Polleres  
Siemens AG Österreich / DERI Galway

Peter Schüller  
Vienna University of Technology

Guillermo Simari  
Universidad Nacional del Sur

Tran Cao Son  
New Mexico State University

Mirek Truszczyński  
University of Kentucky

Joost Vennekens  
K.U. Leuven

Stefan Woltran  
Vienna University of Technology

Fangkai Yang  
University of Texas at Austin

Jia-Huai You  
University of Alberta, Edmonton

Additional Reviewers

Torsten Schaub  
University of Potsdam
# Table of Contents

## Papers

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translating NP-SPEC into ASP</td>
<td>3</td>
</tr>
<tr>
<td>( M.\ \text{Alviano} \ (\text{University of Calabria}) ) and ( W.\ \text{Faber} \ (\text{University of Calabria}) )</td>
<td></td>
</tr>
<tr>
<td>( \text{SPARC} – ) Sorted ASP with Consistency Restoring Rules</td>
<td>19</td>
</tr>
<tr>
<td>( E.\ \text{Balai} \ (\text{Kyrgyz-Russian Slavic University}), \ M.\ \text{Gelfond} \ (\text{Texas Tech University}), ) and ( Y.\ \text{Zhang} \ (\text{Texas Tech University}) )</td>
<td></td>
</tr>
<tr>
<td>Language ASP( {f} ) with Arithmetic Expressions and Consistency-Restoring Rules</td>
<td>35</td>
</tr>
<tr>
<td>( M.\ \text{Balduccini} \ (\text{Eastman Kodak Company}) ) and ( M.\ \text{Gelfond} \ (\text{Texas Tech University}) )</td>
<td></td>
</tr>
<tr>
<td>Utilizing ASP for Generating and Visualizing Argumentation Frameworks</td>
<td>51</td>
</tr>
<tr>
<td>( G.\ \text{Charwat} \ (\text{Vienna University of Technology, Austria}), \ J.\ \text{Wallner} \ (\text{Vienna University of Technology, Austria}), ) and ( S.\ \text{Woltran} \ (\text{Vienna University of Technology, Austria}) )</td>
<td></td>
</tr>
<tr>
<td>Planning and Scheduling in Hybrid Domains Using Answer Set Programming</td>
<td>67</td>
</tr>
<tr>
<td>( S.\ \text{Chintabathina} \ (\text{University of Arkansas at Pine Bluff}) )</td>
<td></td>
</tr>
<tr>
<td>Eliminating Unfounded Set Checking for ( \text{HEX})-Programs</td>
<td>83</td>
</tr>
<tr>
<td>( T.\ \text{Eiter} \ (\text{Vienna University of Technology, Austria}), \ M.\ \text{Fink} \ (\text{Vienna University of Technology, Austria}), \ T.\ \text{Krennwallner} \ (\text{Vienna University of Technology, Austria}), \ \text{C. Redl} \ (\text{Vienna University of Technology, Austria}), ) and ( \text{P. Schüller} \ (\text{Vienna University of Technology, Austria}) )</td>
<td></td>
</tr>
<tr>
<td>Backdoors to Normality for Disjunctive Logic Programs</td>
<td>99</td>
</tr>
<tr>
<td>( J.\ \text{Fichte} \ (\text{Vienna University of Technology, Austria}) ) and ( S.\ \text{Szeider} \ (\text{Vienna University of Technology, Austria}) )</td>
<td></td>
</tr>
<tr>
<td>Answer Set Programming for Stream Reasoning</td>
<td>115</td>
</tr>
<tr>
<td>( M.\ \text{Gebser} \ (\text{University of Potsdam, Germany}), \ T.\ \text{Grote} \ (\text{University of Potsdam, Germany}), \ \text{R. Kaminski} \ (\text{University of Potsdam, Germany}), \ \text{P. Obermeier} \ (\text{DERI Galway, Ireland}), \ \text{O. Sabuncu} \ (\text{University of Potsdam, Germany}), ) and ( \text{T. Schaub} \ (\text{University of Potsdam, Germany}) )</td>
<td></td>
</tr>
<tr>
<td>Two New Definitions of Stable Models of Logic Programs with Generalized Quantifiers</td>
<td>131</td>
</tr>
<tr>
<td>( J.\ \text{Lee} \ (\text{Arizona State University}) ) and ( Y.\ \text{Meng} \ (\text{Arizona State University}) )</td>
<td></td>
</tr>
<tr>
<td>Lloyd-Topor Completion and General Stable Models</td>
<td>145</td>
</tr>
<tr>
<td>( V.\ \text{Lifschitz} \ (\text{University of Texas at Austin}) ) and ( F.\ \text{Yang} \ (\text{University of Texas at Austin}) )</td>
<td></td>
</tr>
</tbody>
</table>
Extending FO(ID) with Knowledge Producing Definitions: Preliminary Results

J. Vennekens (University of Leuven, Belgium) and M. Denecker (University of Leuven, Belgium)

Author Index
Part I

Papers
Translating NP-SPEC into ASP

Mario Alviano and Wolfgang Faber
Department of Mathematics
University of Calabria
87030 Rende (CS), Italy
mario@alviano.com, wf@wfaber.com

Abstract. NP-SPEC is a language for specifying problems in NP in a declarative way. Despite the fact that the semantics of the language was given by referring to Datalog with circumscription, which is very close to ASP, so far the only existing implementations are by means of ECLiPSe Prolog and via Boolean satisfiability solvers. In this paper, we present translations from NP-SPEC into various forms of ASP and analyze them. We also argue that it might be useful to incorporate certain language constructs of NP-SPEC into mainstream ASP.

1 Introduction

NP-SPEC is a language that was proposed in [4, 2] in order to specify problems in the complexity class NP in a simple, clear, and declarative way. The language is based on Datalog with circumscription, in which some predicates are circumscribed, while others are not and are thus “left open”. Some practical features are added to this basic language, often by means of reductions.

The original software system supporting NP-SPEC was described in [2] and was written in the ECLiPSe Constraint Programming System, based on Prolog. A second software system, SPEC2SAT1, was proposed in [3], which rewriting NP-SPEC into propositional formulas for testing satisfiability. The system has also been tested quite extensively in [5], also for several problems taken from CSPLIB, with promising results.

Interestingly, to our knowledge so far no attempt has been made to translate NP-SPEC into Answer Set Programming (ASP), which is very similar in spirit to Datalog with circumscription, and thus a good candidate as a transformation target. Moreover, several efficient ASP software systems are available, which should guarantee good performance. A crucial advantage of ASP versus propositional satisfiability is the fact that NP-SPEC problem descriptions are in general not propositional, and therefore a reduction from NP-SPEC to SAT has to include an implicit instantiation (or grounding) step. Also ASP allows for variables, and ASP systems indeed provide optimized grounding procedures, which include many advanced techniques from database theory (such as indexing, join-ordering, etc). This takes the burden of instantiating in a smart way from the NP-SPEC translation when using ASP systems.

* This work was supported by M.I.U.R. within the PRIN project LoDeN.
1 http://www.dis.uniroma1.it/cadoli/research/projects/NP-SPEC/code/SPEC2SAT/
In this paper we provide a translation from NP-SPEC into various variants of ASP. We discuss properties and limitations of the translation and also provide a prototype implementation, for which we provide a preliminary experimental analysis, which shows that our approach is advantageous, in particular that it pays off if grounding tasks are delegated to existing systems. The rest of the paper is structured as follows: in section 2 we review the language NP-SPEC and give a very brief account of ASP. In section 3 we provide the main ingredients for translations from NP-SPEC to ASP, and discuss properties and limitations. In section 4 we report on preliminary experimental results. Finally, in section 5 we draw our conclusions.

2 Preliminaries: NP-SPEC and ASP

We first provide a brief definition of NP-SPEC programs. For details, we refer to [2]. We also note that a few minor details in the input language of SPEC2SAT (in which the publicly available examples are written) are different to what is described in [2]. We will usually stick to the syntax of SPEC2SAT.

An NP-SPEC program consists of two main sections\(^2\): one section called DATABASE and one called SPECIFICATION, each of which is preceded by the respective keyword.

2.1 DATABASE

The database section defines extensional predicates or relations and (interpreted) constants. Extensional predicates are defined by writing

\[ p = \{ t_1, \ldots, t_n \}; \]

where \( p \) is a predicate symbol and each \( t_i \) is a tuple with matching arity. For unary predicates, each tuple is simply an integer or a constant symbol; for arity greater than 1, it is a comma-separated sequence of integers or constant symbols enclosed in round brackets. Unary extensions that are ranges of integers can also be abbreviated to \( n..m \), where \( n \) and \( m \) are integers or interpreted constants. Constant definitions are written as

\[ c = i; \]

where \( i \) is an integer.

Example 1. The following defines the predicate \( \text{edge} \) representing a graph with six nodes and nine edges, and a constant \( n \) representing the number of nodes.

\[
\text{DATABASE} \\
n = 6; \\
\text{edge} = \{(1, 2), (3, 1), (2, 3), (6, 2), (5, 6), (4, 5), (3, 5), (1, 4), (4, 1)\};
\]

\(^2\) SPEC2SAT also has a third, apparently undocumented section called SEARCH, which seems to define only output features and which we will not describe here.
2.2 SPECIFICATION

The SPECIFICATION section consists of two parts: a search space declaration and a stratified Datalog program. The search space declaration serves as a domain definition for “guessed” predicates and must be one or more of the metafacts $\text{Subset}(d, p)$, $\text{Permutation}(d, p)$, $\text{Partition}(d, p, n)$, and $\text{IntFunc}(d, p, n..m)$, which we will describe below.

$\text{Subset}(d, p)$. This is the basic construct to which all following search space declaration constructs are reduced in the semantic definition in [2]. Here, $d$ is a domain definition, which is either an extensional predicate, a range $n..m$, or a Cartesian product $(>)$, union $(+)$, intersection $(\ast)$, or difference $(-)$ of two domains. Symbol $p$ is a predicate identifier and the intended meaning is that the extension of $p$ can be any subset of the domain definition’s extension, thus giving rise to nondeterminism or a “guess”.

Example 2. Together with the code of Example 1, the following specification will represent all subgraphs (including the original graph) as extensions of predicate subgraph.

\[
\begin{align*}
\text{SPECIFICATION} \\
\text{Subset}(\text{edge}, \text{subgraph}).
\end{align*}
\]

$\text{Permutation}(d, p)$. Concerning this construct, $d$ is again a domain definition, and $p$ will have an extension in which each tuple of $d$ is present and an additional argument associates a unique integer between 1 and the cardinality of the extension of $d$ (say, $c$) to each tuple, thereby defining a permutation. The extensions of $p$ thus define a bijective functions from tuples of the extension of $d$ to $\{1..c\}$.

Example 3. Together with the code of Example 1, the following specification will represent all enumerations of edges.

\[
\begin{align*}
\text{SPECIFICATION} \\
\text{Permutation}(\text{edge}, \text{edgeorder}).
\end{align*}
\]

One extension of $\text{edgeorder}$ that reflects the ordering of the edges as written in Example 1 is

\[
\text{edgeorder}(1,2,1), \text{edgeorder}(3,1,2), \text{edgeorder}(2,3,3), \\
\text{edgeorder}(6,2,4), \text{edgeorder}(5,6,5), \text{edgeorder}(4,5,6), \\
\text{edgeorder}(3,5,7), \text{edgeorder}(1,4,8), \text{edgeorder}(4,1,9).
\]

$\text{Partition}(d, p, n)$. Also in this case $p$ will have one argument more than $d$. In this case, extensions of $p$ will define functions from tuples of the extension of $d$ to $\{1..n\}$, thereby defining $n$ (possibly empty) partitions.

Example 4. Together with the code of Example 1, the following specification will represent all possible pairs of graphs that partition the input graph.

\[
\begin{align*}
\text{SPECIFICATION} \\
\text{Partition}(\text{edge}, \text{partition}, 2).
\end{align*}
\]
One extension of partition that has the first four edges in the first partition (i.e., partition 0) and the last five edges in the second partition (i.e., partition 1) would be

\[
\text{partition}(1, 2, 0), \text{partition}(3, 1, 0), \text{partition}(2, 3, 0), \\
\text{partition}(6, 2, 0), \text{partition}(5, 6, 1), \text{partition}(4, 5, 1), \\
\text{partition}(3, 5, 1), \text{partition}(1, 4, 1), \text{partition}(4, 1, 1).
\]

Then verifies the Hamiltonian Cycle condition by means of integrity constraints, one exploiting the linear order of the permutation identifiers, and another one to close the cycle from the last permutation identifier to the first one.

Example 5. The following specification is equivalent to the one in Example 4:

\[
\text{SPECIFICATION}
\]

\[
\text{IntFunc}(\text{edge}, \text{partition}, 0..1).
\]

Stratified Datalog Program. The stratified Datalog program is written using < as the rule implication symbol. It may contain built-in predicates (==, <, >, >=, <=, ! =), arithmetic expressions, and stratified aggregates (\text{COUNT}, \text{SUM}, \text{MIN}, \text{MAX}). It may also contain integrity constraints, in which case rule heads contain the special symbol \text{fail}. Rule implication is denoted by <, the aggregates are written as for example \text{SUM}(p(*, ..Y), Z : n..m) where: * specifies the argument to be aggregated over; variables that are not shared with other rule literals are local (as a special case the anonymous variable ..) and represent the arguments that are not fixed; variables that are shared with other rule literals are considered fixed in the aggregation; and variable Z will contain the valuation of the aggregate, which will must be in the range n..m. Comments may be written in C++ style (using // or /* */).

Example 6. As an example, consider the well-known Hamiltonian Cycle problem. The \text{NP-SPEC} distribution contains an example program for an example graph:

\[
\text{DATABASE}
\]

\[
\text{n} = 6; \text{//no. of nodes}
\text{edge} = \{(1, 2), (3, 1), (2, 3), (6, 2), (5, 6), (4, 5), (3, 5), (1, 4), (4, 1)\};
\]

\[
\text{SPECIFICATION}
\]

\[
\text{Permutation}([1..n], \text{path}).
\text{fail} < -- \text{path}(X, P), \text{path}(Y, P + 1), \text{NOT edge}(X, Y).
\text{fail} < -- \text{path}(X, n), \text{path}(Y, 1), \text{NOT edge}(X, Y).
\]

The \text{DATABASE} section contains an encoding of the example graph by means of the binary predicate edge and defines a constant \text{n} for representing the number of nodes of that graph. Implicitly it is assumed that the nodes are labeled by integers from 1 to \text{n}. The \text{SPECIFICATION} section then first guesses a permutation of the nodes and then verifies the Hamiltonian Cycle condition by means of integrity constraints, one exploiting the linear order of the permutation identifiers, and another one to close the cycle from the last permutation identifier to the first one.
The semantics of NP-SPEC programs is provided by means of Datalog with Circumscription, in which some predicates are minimized. That means that among all models only those which are minimal with respect to the minimized predicates are accepted. Moreover, among these only those which make the special symbol \texttt{fail} false are considered and referred to as answers. All metafacts are reduced to the basic metafact \texttt{Subset} that effectively states that the predicate defined by the metafact is not minimized. For further details of the semantics, we refer to [2].

Concerning ASP, we only give a very brief overview, details may be found in works such as [1, 13, 8]. An ASP program consists of rules
\[ L_1 \lor \cdots \lor L_k : \neg \text{Body} \]
where the \( L_i \) are literals containing variables and constants\(^3\) (possibly containing strong negation) and \( \text{Body} \), which is a conjunction of literals, that may also contain built-ins, aggregates and default negation. Rules without heads act like integrity constraints. The semantics is based on the Gelfond-Lifschitz reduct [11] and also guarantees minimality of the answer sets.

Practical ASP systems differ in several details, for instance several do not support disjunction in rule heads, built-in predicates and arithmetic expressions may differ and also aggregates are sometimes written in slightly different ways. In this paper, we will use the syntax of gringo (http://potassco.sourceforge.net/) and DLV (http://www.dlvsystem.com). Both systems assume that the input programs are safe, that is, each variable in a rule must also occur in a positive body atom. While gringo can also parse disjunctive programs, clasp, the solver it is often used with, can only deal with nondisjunctive programs.

\textbf{Example 7.} As an example, consider the Hamiltonian Cycle problem and instance from above. An ASP encoding similar to the NP-SPEC program seen earlier would be:

\begin{verbatim}
#const n = 6
d(1..n).
edge(1, 2), edge(3, 1), edge(2, 3), edge(6, 2), edge(5, 6),
edge(4, 5), edge(3, 5), edge(1, 4), edge(4, 1).
d(X).
path(X, 1) \lor path(X, 2) \lor path(X, 3) \lor path(X, 4) \lor path(X, 5) \lor path(X, 6) : \neg d(X).

: - path(X, A), path(Y, A), X \neq Y.
: - path(X, P), path(Y, Z), not edge(X, Y), Z = P + 1.
: - path(X, n), path(Y, 1), not edge(X, Y).
\end{verbatim}

This program is usable for gringo with clasp, using the \texttt{--shift} option (transforming the disjunctive rule into several nondisjunctive ones), and DLV. We can observe that the extensional definition is rewritten into a number of facts and that the constant definition also just changes syntax. As for the permutation statement, here we first use a predicate \texttt{d} representing the domain definition, and then a disjunctive rule and an integrity constraint. The disjunctive rule states that each tuple in the domain definition must be assigned one of the numbers 1 to 6, and the integrity constraint enforces the bijection.

\(^3\) Many modern ASP systems also allow for function symbols, but they are not needed here.
that is, no different tuples of the domain definition must be assigned the same number. The final two integrity constraints are direct translations from the NP-SPEC program. The only difference is the arithmetic expression that has been moved outside the fact in order to conform to DLV’s syntax (gringo would also have accepted the immediate translation from the NP-SPEC program).

3 Translation from NP-SPEC to ASP

We now report how the various constructs of NP-SPEC programs can be translated into ASP. We start with the DATABASE section constructs. An extensional declaration of the form \( p = \{ t_1, \ldots, t_n \} \) will be translated to facts \( p(t_1) \cdots p(t_n) \), and one of the form \( p = \{ n \cdots m \} \) will be translated to facts \( p(n) \cdots p(m) \). Constant declarations such as \( c = i \) instead, will be managed in-memory.

Now for the main task, translating the SPECIFICATION constructs. We first look at metapredicates. The simplest one is \( \text{Subset}(d, p) \). The exact translation of this metafact (and all others as well) depends on how \( d \) is specified. First assume that it is an extensional predicate \( d \) of arity \( n \), then we can directly use it and produce

\[
p(X_1, \ldots, X_n) \lor -p(X_1, \ldots, X_n) : -d(X_1, \ldots, X_n).
\]

Otherwise, we will create an extensional predicate \( d \) of appropriate arity (assuming that it is a fresh symbol) as follows:

- for the Cartesian product \( p > q \), the following set of facts is created: \( \{ d(x_1, \ldots, x_{i+1}) | p(x_1, \ldots, x_i) \land q(x_{i+1}, \ldots, x_{i+1}) \} \), where \( i \) and \( j \) are the arities of \( p \) and \( q \), respectively;
- for the union \( p + q \), the following set of facts is created: \( \{ d(x_1, \ldots, x_i) | p(x_1, \ldots, x_i) \lor q(x_1, \ldots, x_i) \} \), where \( i \) is the arity of both \( p \) and \( q \);
- for the intersection \( p \ast q \), the following set of facts is created: \( \{ d(x_1, \ldots, x_i) | p(x_1, \ldots, x_i) \land q(x_1, \ldots, x_i) \} \), where \( i \) is the arity of both \( p \) and \( q \); and
- for the difference \( p - q \), the following set of facts is created: \( \{ d(x_1, \ldots, x_i) | p(x_1, \ldots, x_i) \land -q(x_1, \ldots, x_i) \} \), where \( i \) is the arity of both \( p \) and \( q \), and \( -q(x_1, \ldots, x_i) \) is true if and only if the fact \( q(x_1, \ldots, x_i) \) is not part of the translation.

For nested domain definitions, we just repeat this process recursively using fresh symbols in each recursive step. In the following we will assume that domain definitions have been treated in this way and that the top-level predicate of the translation is \( d \). If available (for instance when using gringo or lparse), we can also use choice rules for translating \( \text{Subset}(d, p) \):

\[
\{ p(X_1, \ldots, X_n) : d(X_1, \ldots, X_n) \}.
\]

For the metafact \( \text{Permutation}(d, p) \), we will create

\[
p(X_1, \ldots, X_n, 1) \lor \ldots \lor p(X_1, \ldots, X_n, c) : -d(X_1, \ldots, X_n).
\]

\[
: -p(X_1, \ldots, X_n, A), p(Y_1, \ldots, Y_n, A), X_1! = Y_1.
\]

\[
: -p(X_1, \ldots, X_n, A), p(Y_1, \ldots, Y_n, A), X_n! = Y_n.
\]
where \( n \) is the arity of \( d \) and \( c \) is the cardinality of \( d \). The first rule specifies intuitively that for each tuple in \( d \) one of \( p(x_1, \ldots, x_n, 1) \cdots p(x_1, \ldots, x_n, c) \) should hold, and by minimality exactly one of these will hold. The integrity constraints ensure that no different numbers will be associated to the same tuple. As an alternative to the disjunctive rule, one can use a choice rule

\[
1\{p(x_1, \ldots, x_n, 1\cdots c)\} 1: \neg d(x_1, \ldots, x_n).
\]

Instead of the \( n \) integrity constraints it is possible to write just one using an aggregate, if available. In the DLV syntax, one could write

\[
\neg \#\text{count}\{x_1, \ldots, x_n : p(x_1, \ldots, x_n, A)\} > 1, p(\ldots, \ldots, A).
\]

or in gringo syntax

\[
\neg 2 \#\text{count}\{p(x_1, \ldots, x_n, A), p(\ldots, \ldots, A).
\]

The remaining metafacts are actually much simpler to translate, as the bijection criterion does not have to be checked. For \( \text{Partition}(d, p, k) \), we will simply create

\[
p(x_1, \ldots, x_n, 0) \lor \ldots \lor p(x_1, \ldots, x_n, k - 1) : \neg d(x_1, \ldots, x_n).
\]

or the respective choice rule

\[
1\{p(x_1, \ldots, x_n, 0\ldots k - 1)\} 1: \neg d(x_1, \ldots, x_n).
\]

where \( n \) is the arity of \( d \). For \( \text{IntFunc}(d, p, i..j) \), we will simply create

\[
p(x_1, \ldots, x_n, i) \lor \ldots \lor p(x_1, \ldots, x_n, j) : \neg d(x_1, \ldots, x_n).
\]

or the respective choice rule

\[
1\{p(x_1, \ldots, x_n, i\ldots j)\} 1: \neg d(x_1, \ldots, x_n).
\]

where \( n \) is the arity of \( d \).

What remains are the Datalog rules of the SPECIFICATION section. Essentially, each \( \#\text{head} < \neg \text{Body} \) is directly translated into \( \#\text{head}' : \neg \text{Body}' \), with only minor differences. If \( \#\text{head} \) is \( \text{fail} \), then \( \#\text{head}' \) is empty, otherwise it will be exactly the same. The difference between \( \text{Body} \) and \( \text{Body}' \) is due to different syntax for arithmetics, aggregates and due to safety requirements. Concerning arithmetics, gringo can accept almost the same syntax as NP-SPEC with only minor differences (\( \#\text{abs} \) instead of \( \text{abs} \), \( \#\text{pow} \) instead of ^), while DLV is much more restrictive. DLV currently does not support negative integers and it does not provide constructs corresponding to \( \text{abs} \) and ^. Moreover, arithmetic expressions may not be nested in DLV programs, but this limitation can be overcome by flattening the expressions.

Concerning aggregates, DLV and gringo support similar syntax, which is a little bit different from the one used in NP-SPEC but rather straightforward to rewrite according to the following schema: Arguments marked with asterisks are first replaced with fresh variables; these are the arguments on which the aggregation function is applied. Apart
from COUNT, exactly one asterisk may appear in each aggregate. Hence, an aggregate
\( \sum(p(\ast, \_, Y), Z : n..m) \) is written in DLV’s syntax as
\[
\#\text{sum}\{X : p(X, \_, Y)\} = Z, \ d(Z)
\]
and in gringo’s syntax as
\[
Z \#\text{sum}\[p(X, \_, Y) = X\] Z, \ d(Z)
\]
where \( X \) is a fresh variable and \( d \) is a fresh predicate defined by facts \( d(n) \cdots d(m) \).
Aggregates MIN and MAX are rewritten similarly, while an aggregate
\(
\text{COUNT}(p(\ast, \_, \ast, Y), Z : n..m)
\)
is written in DLV’s syntax as
\[
\#\text{count}\{X_1, X_2 : p(X_1, \_, X_2, Y)\} = Z, \ d(Z)
\]
and in gringo’s syntax by
\[
Z \#\text{count}\{p(X_1, \_, X_2, Y)\} Z, \ d(Z).
\]

A more difficult problem presents the safety conditions enforced by the ASP systems. NP-SPEC has a fairly lax safety criterion, while for instance DLV requires each variable to occur in a positive, non-builtin body literal, and also gringo has a similar criterion. This mismatch can be overcome by introducing appropriate domain predicates when needed.

4 Experiments

We have created a prototype implementation of the transformation described in section 3, which is available at http://archives.alviano.com/npspec2asp/. It is written in C++ using bison and flex, and called NPSPEC2ASP. The implementation at the moment does only rudimentary correctness checks of the program and is focussed on generating ASP programs for correct NP-SPEC input. Moreover, at the moment it generates only the disjunctive rules described in section 3 rather than the choice rules, but we plan to add the possibility to create variants of the ASP code in the near future. For the experiments, the transformation used for Permutation produced the integrity constraint with the counting aggregate.

We used this implementation to test the viability of our approach, in particular assessing the efficiency of the proposed rewriting in ASP with respect to the previously available transformation into SAT. In the benchmark we included several instances available on the NP-SPEC site. More specifically, we considered two sets of instances, namely the \textit{miscellanea} and \textit{csp2npspec} benchmarks. Even if these instances have been conceived for demonstrating the expressivity of the language rather than for assessing the efficiency of an evaluator, it turned out that even for these comparatively small instances there are quite marked performance differences. Below we provide some more details on the testcases in the \textit{miscellanea} benchmark.
- **Coloring** is an instance of the **Graph Coloring** problem, i.e., given a graph $G$ and a set of $k$ colors, checking whether it is possible to assign a color to each node of $G$ in such a way that no adjacent nodes of $G$ share the same color. The tested instance has 6 nodes and 3 colors.

- In the **Diophantine** problem, three positive integers $a, b, c$ are given, and an integer solution to the equation $ax^2 + by = c$ is asked for. The parameters of the tested instance are $a = 5$, $b = 3$, and $c = 710$.

- The **Factoring** problem consists of finding two non-trivial factors (i.e., greater than 1) of a given integer $n$. In the tested instance, $n = 10000$.

- In the **Hamiltonian Cycle** problem a graph $G$ is given, and a cycle traversing each node exactly once is searched. The tested graph has 6 nodes.

- An instance of the **Job Shop Scheduling** problem consists of integers $n$ (jobs), $m$ (tasks), $p$ (processors), and $D$ (global deadline). Jobs are ordered collections of tasks, and each task is performed on a processor for some time. Each processor can perform one task at a time, and the tasks belonging to the same job must be performed in order. The problem is checking whether it is possible for all jobs to meet deadline $D$. In the testcase, $n = 6$, $m = 36$, $p = 6$, and $D = 55$.

- In the **Protein Folding** problem, a sequence of $n$ elements in $\{H, P\}$ is given, and the goal is to find a connected, non-overlapping shape of the sequence on a bi-dimensional, discrete grid, so that the number of “contacts”, i.e., the number of non-sequential pairs of $H$ for which the Euclidean distance of the positions is 1, is in a given range $R$. In the testcase, $n = 6$, and $R = \{1..12\}$.

- In the **Queens** problem, an integer $n$ is given, and the goal is to place $n$ non-attacking queens on a $n \times n$ chessboard. In the tested instance, $n = 5$.

- Given an array $A$ of integers, the **Sorting** problem consists of arranging the elements of $A$ in non-descending order. In the tested instance, the array has 7 elements.

- An instance of the **Subset Sum** problem comprises a finite set $A$, a size $s(a) \in \mathbb{N}^+$ for each $a \in A$, and $B \in \mathbb{N}^+$. The goal of the problem is checking whether there is a subset $A'$ of $A$ such that the sum of the sizes of the elements in $A'$ is exactly $B$. In the tested instance, set $A$ has 5 elements and $B = 10$.

- In a **Sudoku**, the goal is to fill a given (partially filled) grid with the numbers 1 to 9, so that every column, row, and $3 \times 3$ box indicated by slightly heavier lines has the numbers 1 to 9.

- **3-SAT** is a well-known NP-complete problem: Given a propositional formula $T$ in conjunctive normal form, in which each clause has exactly three literals, is $T$ satisfiable, i.e., does there exist an assignment of variables of $T$ to $\{true, false\}$ that makes $T$ evaluate to true? The tested instance has 3 clauses.

- The **Tournament Scheduling** problem consists of assigning the matches to rounds of a round-robin tournament for a sports league. The match is subject to several constraints, such as: (i) complementary teams $t_1$ and $t_2$ have complementary schedules, i.e., for each round $r$, if $t_1$ plays home in $r$ then $t_2$ plays away in $r$, and vice versa; (ii) two top matches cannot take place at distance smaller than a given value; (iii) any team cannot match two top teams at distance smaller than a given value. (See [5] for details.) The tested instance has 6 teams.

Below we describe the testcases in the csplib2npspec benchmark.
- Given $n \in \mathbb{N}$, find a vector $s = (s_1, ..., s_n)$ such that (i) $s$ is a permutation of $Z_n = \{0, 1, \ldots, n - 1\}$; and (ii) the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, \ldots, |s_n - s_{n-1}|)$ is a permutation of $Z_n \setminus \{0\} = \{1, 2, \ldots, n - 1\}$. A vector $v$ satisfying these conditions is called an all-interval series of size $n$; the problem of finding such a series is the *All-interval Series* problem of size $n$. In the tested instance, $n = 20$.

- In the *BACP* (balanced academic curriculum problem), each course has associated a number of credits and can have other courses as prerequisites. The goal is to assign a period to every course in a way that the number of courses and the amount of credits per period are in given ranges, and the prerequisite relationships are satisfied. The tested instance comprises 7 courses and 2 periods.

- A *BIBD* is defined as an arrangement of $v$ distinct objects into $b$ blocks such that each block contains exactly $k$ distinct objects, each object occurs in exactly $r$ different blocks, and every two distinct objects occur together in exactly $\lambda$ blocks. The parameters of the tested instance are $v = 7$, $b = 7$, $k = 3$, $r = 3$, and $\lambda = 1$.

- In the *Car Sequencing* problem, a number of cars are to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. In the test case there are 10 cars, 6 variants on a basic model, and 5 options.

- A *Golomb ruler* is a set of $m$ integers $0 = a_1 < a_2 < \cdots < a_m$ such that the $m(m-1)/2$ differences $a_j - a_i$ $(1 \leq i < j \leq m)$ are distinct. In the tested instance, $m = 8$ and $a_m$ must be lesser than or equals to 10.

- *Langford*’s problem is to arrange $k$ sets of numbers 1 to $n$ so that each appearance of the number $m$ is $m$ numbers on from the last. In the tested instance, $k = 3$ and $n = 9$.

- Given integers $n$ and $b$, the objective of the *Low Autocorrelation* problem is to construct a binary sequence $S_i$ of length $n$, where each bit takes the value +1 or -1, so that $E = \sum_{k=1}^{n-1} (C_k)^2 \leq b$, where $C_k = \sum_{i=0}^{n-k-1} S_i \cdot S_{i+k}$. In the tested instance, $n = 5$ and $b = 2$.

- An order $n$ *magic square* is a $n \times n$ matrix containing the numbers 1 to $n^2$, with each row, column and main diagonal summing up to the same value. In our setting, $n = 3$.

- The *Ramsey* problem is to color the edges of a complete graph with $n$ nodes using at most $k$ colors, in such a way that there is no monochromatic triangle in the graph. In the tested instance, $n = 5$ and $k = 3$.

- The *Round-robin Tournament* problem is to schedule a tournament of $n$ teams over $n - 1$ weeks, with each week divided into $n/2$ periods, and each period divided into two slots. A tournament must satisfy the following three constraints: every team plays once a week; every team plays at most twice in the same period over the tournament; every team plays every other team. In our setting, $n = 4$.

- *Schur’s Lemma* problem is to put $n$ balls labelled $\{1, \ldots, n\}$ into 3 boxes so that for any triple of balls $(x, y, z)$ with $x + y = z$, not all are in the same box. In the tested instance, $n = 10$. 


In the *Social Golfer* problem there are *n* golfers, each of whom play golf once a week, and always in groups of *s*. The goal is to determine a schedule of play for these golfers, to last *l* weeks, such that no golfer plays in the same group as any other golfer on more than one occasion. In our setting, *n* = 8, *s* = 2, and *l* = 4.

The experiment has been executed on an Intel Core2 Duo P8600 2.4 GHz with 4 GB of central memory, running Linux Mint Debian Edition (wheezy/sid) with kernel Linux 3.2.0-2-amd64. The tools SPEC2SAT and NPSPEC2ASP have been compiled with gcc 4.6.3. The other tools involved in the experiment are satz 215.2 [14], minisat 1.14 [7], gringo 3.0.4 [10], clasp 2.0.6 [9], and DLV 2011-12-21 [12].

In our experiment, we first measured the running time required by SPEC2SAT and NPSPEC2ASP to rewrite the input specification into SAT and ASP, respectively. Then, for each SAT encoding produced by SPEC2SAT, we ran three SAT solvers, namely satz, minisat and clasp, to obtain one solution if one exists. For each of these executions we measured the time to obtain the solution or the assertion that none exists, thus the sum of the running times of SPEC2SAT and of the SAT solvers. Moreover, for each ASP encoding produced by NPSPEC2ASP, we ran two instantiators, namely gringo and DLV (with option `--instantiate`). For each of these runs we measured the time required to compute the ground ASP program, thus the sum of the running times of NPSPEC2ASP and of the instantiator. Finally, for each ground ASP program computed by gringo and DLV, we computed one solution by using clasp and DLV, respectively, and measured the overall time required by the tool-chain. We have also measured the sizes of the instantiated formulas and programs. For SPEC2SAT, we report the number of clauses in the produced formula and the number of propositional variables occurring in it. For DLV and gringo we report the number of ground rules produced and the number of ground atoms occurring in them. There is a slight difference in the statistics provided by DLV and gringo: DLV does not count ground atoms (and facts) that were already found to be true; to be more comparable, we added the number of facts for DLV.

Experimental results concerning the *miscellanea* benchmark are reported in Table 1. We first observe that the time required by NPSPEC2ASP is below the measurement accuracy, while the execution time of SPEC2SAT is higher, sometimes by several orders of magnitude. In fact, SPEC2SAT has to compute a ground SAT instance to pass to a SAT solver, while NPSPEC2ASP outputs a non-ground ASP program. A fairer comparison is obtained by adding to the time taken by NPSPEC2ASP the time required by the ASP instantiator to obtain a ground ASP program. Columns gringo and “DLV inst” report these times, which are however always less than those of SPEC2SAT. In Table 2 it can be seen that also the number of ground rules produced by the ASP systems is usually smaller than the number of clauses produced by SPEC2SAT, even if often the number of ground atoms exceeds the number of propositional variables.

Concerning the computation of one solution from each ground specification, all considered SAT and ASP solvers are fast in almost all tests. The only exceptions are satz for *proteinFolding*, which exceeds the allotted time, and DLV for *jobShopScheduling*, whose execution lasted around 88 seconds. We also note that DLV has not been tested on 2 instances containing negative integers, which are not allowed in the DLV language.

Table 3 reports experimental results concerning the *csplib2npspec* benchmark. We start by observing that instances in this benchmark are more resource demanding than
instances in the *miscellanea* benchmark. In fact, we note that *golombRuler* is too difficult for SPEC2SAT, which did not terminate on the allotted time on this instance. On the other hand, the rewriting provided by NPSPEC2ASP is processed in around 28 seconds by gringo+clasp and in around 24 seconds by DLV. Another hard instance is *allInterval*, for which only satz and DLV terminated in the allotted time. All other solvers, includ-

### Table 1. Running times on the *miscellanea* benchmark

<table>
<thead>
<tr>
<th>Instance</th>
<th>SPEC2SAT</th>
<th>NPSPEC2ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>only</td>
<td>only</td>
</tr>
<tr>
<td></td>
<td>satz</td>
<td>minisat</td>
</tr>
<tr>
<td></td>
<td>inst</td>
<td></td>
</tr>
<tr>
<td>coloring</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>diophantine</td>
<td>0.75</td>
<td>0.81</td>
</tr>
<tr>
<td>factoring</td>
<td>5.99</td>
<td>10.21</td>
</tr>
<tr>
<td>hamiltonianCycle</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>jobShopScheduling</td>
<td>43.39</td>
<td>44.89</td>
</tr>
<tr>
<td>proteinFolding</td>
<td>132.17</td>
<td>&gt;600</td>
</tr>
<tr>
<td>queens</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>sorting</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>subsetSum</td>
<td>0.11</td>
<td>0.11</td>
</tr>
<tr>
<td>sudoku</td>
<td>3.05</td>
<td>3.07</td>
</tr>
<tr>
<td>threeSat</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>tournamentScheduling</td>
<td>0.45</td>
<td>0.45</td>
</tr>
</tbody>
</table>

* The instance contains negative integers.

### Table 2. Instance sizes of the *miscellanea* benchmark

<table>
<thead>
<tr>
<th>Instance</th>
<th>SPEC2SAT</th>
<th>NPSPEC2ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clauses</td>
<td>Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coloring</td>
<td>45</td>
<td>18</td>
</tr>
<tr>
<td>diophantine</td>
<td>14,628</td>
<td>140</td>
</tr>
<tr>
<td>factoring</td>
<td>123,748</td>
<td>498</td>
</tr>
<tr>
<td>hamiltonianCycle</td>
<td>348</td>
<td>36</td>
</tr>
<tr>
<td>jobShopScheduling</td>
<td>209,495</td>
<td>1,980</td>
</tr>
<tr>
<td>proteinFolding</td>
<td>735,721</td>
<td>669</td>
</tr>
<tr>
<td>queens</td>
<td>165</td>
<td>25</td>
</tr>
<tr>
<td>sorting</td>
<td>427</td>
<td>49</td>
</tr>
<tr>
<td>subsetSum</td>
<td>1,418</td>
<td>125</td>
</tr>
<tr>
<td>sudoku</td>
<td>33,825</td>
<td>1,458</td>
</tr>
<tr>
<td>threeSat</td>
<td>30</td>
<td>39</td>
</tr>
<tr>
<td>tournamentScheduling</td>
<td>1,641</td>
<td>108</td>
</tr>
</tbody>
</table>

* The instance contains negative integers.
ing gringo+clasp, exceeded the allotted time, even if the NPSPEC2ASP rewriting and
the instantiation by gringo is produced in less time than the output of SPEC2SAT. This
instance is an outlier in our experiments. In almost all other instances the ASP solvers
compute solutions in less than 1 second, while SAT solvers typically require several
seconds, see in particular langford, magicSquare and lowAutocorrelation. The size of
the programs produced by the ASP instantiators is always smaller than the size of the
formulas produced by SPEC2SAT, sometimes by orders of magnitude, even if the num-
ber of ground atoms often exceeds the number of propositional variables. A major cause
for the difference in size appear to be aggregates in the problem specification, which are
supported natively by ASP systems, but require expensive rewritings for SAT solvers.

The experimental results show that translating NP-SPEC programs into ASP rather
than SAT seems to be preferable, due to the fact that sophisticated instantiation tech-
niques can be leveraged. Moreover, also the nondeterministic search components of
ASP systems can compete well with SAT solvers, making the use of ASP solvers very
attractive for practical purposes.

5 Conclusion

In this paper we have presented a transformation of NP-SPEC programs into ASP. The
translation is modular and not complex at all, allowing for very efficient transforma-
tions. Compared to the previously available transformation into Boolean satisfiability,
there are a number of crucial differences: While our transformation is from a formalism
with variables into another formalism with variables, Boolean satisfiability of course
does not allow for object variables. Therefore any transformation to that language has

<table>
<thead>
<tr>
<th>Instance</th>
<th>SPEC2SAT</th>
<th>NPSPEC2ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>only</td>
<td>only</td>
</tr>
<tr>
<td></td>
<td>satz</td>
<td>DLV</td>
</tr>
<tr>
<td></td>
<td>minisat</td>
<td>inst</td>
</tr>
<tr>
<td></td>
<td>clasp</td>
<td>DLV</td>
</tr>
<tr>
<td></td>
<td>gringo</td>
<td>gringo</td>
</tr>
<tr>
<td></td>
<td>+clasp</td>
<td>+clasp</td>
</tr>
<tr>
<td>allInterval</td>
<td>1.43</td>
<td>0.00</td>
</tr>
<tr>
<td>bcp</td>
<td>6.15</td>
<td>0.00</td>
</tr>
<tr>
<td>bibd</td>
<td>3.98</td>
<td>0.00</td>
</tr>
<tr>
<td>carSequencing</td>
<td>8.69</td>
<td>0.00</td>
</tr>
<tr>
<td>golombRuler</td>
<td>&gt;600</td>
<td>0.00</td>
</tr>
<tr>
<td>langford</td>
<td>11.57</td>
<td>0.00</td>
</tr>
<tr>
<td>lowAutocorrelation</td>
<td>23.17</td>
<td>0.00</td>
</tr>
<tr>
<td>magicSquare</td>
<td>10.55</td>
<td>0.00</td>
</tr>
<tr>
<td>ramseyProblem</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>roundrobinTournament</td>
<td>2.11</td>
<td>0.00</td>
</tr>
<tr>
<td>schursLemma</td>
<td>0.13</td>
<td>0.00</td>
</tr>
<tr>
<td>socialGolfer</td>
<td>7.32</td>
<td>0.00</td>
</tr>
</tbody>
</table>

* The instance contains negative integers.
to do an implicit instantiation. It is obvious that instantiation can be very costly, and thus using sophisticated instantiation methods is often crucial. However, optimization methods for instantiation are often quite involved and not easy to implement, and therefore adopting them in a transformation is detrimental. After all, the appeal of transformations are usually their simplicity and the possibility to re-use existing software after the transformation. Our transformation method does just that; by not instantiating it is possible to re-use existing instantiators inside ASP systems, many of which use quite sophisticated techniques like join ordering heuristics, dynamic indexing and many more. We have provided a prototype implementation that showcases this advantage. Even if only rather small examples were tested, already in most of those cases a considerable advantage of our method can be observed.

There is a second aspect of our work, which regards ASP. As can be seen in section 3, the translation of Permutation either gives rise to possibly many integrity constraints or one with an aggregate. In any case, all current ASP instantiators will materialize all associations between tuples of the domain definition and the permutation identifiers, even if the identifiers are not really important for solving the problem. This means that there are obvious symmetries in the instantiated program. There exist proposals for symmetry breaking in ASP (e.g. [6]), but they typically employ automorphism detection. We argue that in cases like this, a statement like Permutation, Partition, or IntFunc would make sense as a language addition for ASP solvers, which could exploit the fact that the permutation identifiers introduce a particular known symmetry pattern that does not have to be detected by any external tool.

### Table 4. Instance sizes of the csplib2npspec benchmark

<table>
<thead>
<tr>
<th>Instance</th>
<th>SPEC2SAT</th>
<th>NPSPEC2ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clauses</td>
<td>Variables</td>
</tr>
<tr>
<td>allInterval</td>
<td>21,737</td>
<td>761</td>
</tr>
<tr>
<td>bcp</td>
<td>39,531</td>
<td>1,518</td>
</tr>
<tr>
<td>bibd</td>
<td>31,843</td>
<td>4,424</td>
</tr>
<tr>
<td>carSequencing</td>
<td>39,875</td>
<td>786</td>
</tr>
<tr>
<td>golombRuler</td>
<td>N/A**</td>
<td>N/A**</td>
</tr>
<tr>
<td>langford</td>
<td>130,518</td>
<td>7299</td>
</tr>
<tr>
<td>lowAutocorrelation</td>
<td>186,407</td>
<td>5,952</td>
</tr>
<tr>
<td>magicSquare</td>
<td>38,564</td>
<td>1,975</td>
</tr>
<tr>
<td>ramseyProblem</td>
<td>80</td>
<td>30</td>
</tr>
<tr>
<td>roundrobinTournament</td>
<td>9,272</td>
<td>456</td>
</tr>
<tr>
<td>schursLemma</td>
<td>175</td>
<td>30</td>
</tr>
<tr>
<td>socialGolfer</td>
<td>21,600</td>
<td>1,424</td>
</tr>
</tbody>
</table>

* The instance contains negative integers.
** The system did not terminate in 30 minutes.
Future work consists of consolidating the prototype software and extending the experimentation. Moreover, we intend to investigate the possibility to extend our transformation to work with other languages that are similar to NP-SPEC. Finally, we also intend to explore the possibility and impact of introducing Permutation, Partition, or IntFunc into ASP languages.

References

SPARC – Sorted ASP with Consistency Restoring Rules

Evgenii Balai1, Michael Gelfond2, and Yuanlin Zhang2

1 Kyrgyz-Russian Slavic University
2 Texas Tech University, USA
ienisen2@mail.ru, michael.gelfond@ttu.edu, y.zhang@ttu.edu

Abstract. This is a preliminary report on the work aimed at making CR-Prolog – a version of ASP with consistency restoring rules – more suitable for use in teaching and large applications. First we describe a sorted version of CR-Prolog called SPARC. Second, we translate a basic version of the CR-Prolog into the language of DLV and compare the performance with the state of the art CR-Prolog solver. The results form the foundation for future more efficient and user friendly implementation of SPARC and shed some light on the relationship between two useful knowledge representation constructs: consistency restoring rules and weak constraints of DLV.

1 Introduction

The paper continues work on design and implementation of knowledge representation languages based on Answer Set Prolog (ASP) [1]. In particular we concentrate on the extension of ASP called CR-Prolog – Answer Set Prolog with consistency restoring rules (CR-rules for short) [2]. The language, which allows a comparatively simple encoding of indirect exceptions to defaults, has been successfully used for a number of applications including planning [3], probabilistic reasoning [4], and reasoning about intentions [5]. This paper is a preliminary report on our attempts to make CR-Prolog (and hence other dialects of ASP) more user friendly and more suitable for use in teaching and large applications. This work goes in two different, but connected, directions. First we expand the syntax of CR-Prolog by introducing sorts. Second, we translate a basic version of the CR-Prolog into the language of DLV with weak constraints [6] and compare the efficiency of the resulting DLV based CR-Prolog solver with the CR-Prolog solver implemented in [7]. The original hope for the second part of the work was to obtain a substantially more efficient inference engine for CR-Prolog. This was a reasonable expectation – the older engine is built on top of existing ASP solvers and hence does not fully exploit their inner structure. However this didn’t quite work out. Each engine has its strong and weak points and the matter requires further investigation. But we believe that even preliminary results are of interest since they shed some light on the relationship between two useful knowledge representation constructs: CR-rules and weak constraints. The first goal requires a lengthier explanation. Usually, a program of an Answer Set Prolog based language is understood as a pair, consisting of a signature and a collection of logic programming rules formed from symbols of this signature.
The syntax of the language does not provide any means for specifying this signature – the usual agreement is that the signature consists of symbols explicitly mentioned in the programming rules. Even though in many cases this provides a reasonable solution there are also certain well-known (e.g., [8, 9]) drawbacks:

1. Programs of the language naturally allow unsafe rules which
   – May lead to change of the program behavior under seemingly unrelated updates. A program \( \{ p(1). \quad q \leftarrow \neg p(X). \quad \neg q \leftarrow \neg q. \} \) entails \( \neg q \), but this conclusion should be withdrawn after adding seemingly unrelated fact \( r(2) \). (This happens because of the introduction of a new constant 2 which leads to a new ground instance of the second rule: \( q \leftarrow \neg p(2) \).)
   – Cause difficulties for the implementation of ASP solvers. That is why most implementations do not allow unsafe rules. The corresponding error messages however are not always easy to decipher and the elimination of errors is not always an easy task.

2. The language is untyped and therefore does not provide any protection from unfortunate typos. Misspelling John in the fact \( \text{parent(jone, mary)} \) will not be detected by a solver and may cost a programmer unnecessary time during the program testing.

There were several attempts to address these problems for ASP and some of its variants. The \#domain statements of input language of lparse [9] — a popular grounder used for a number of ASP systems — defines sorts for variables. Even though this device is convenient for simple programs and allows to avoid repetition of atoms defining sorts of variables in the bodies of program’s rules it causes substantial difficulties for medium size and large programs. It is especially difficult to put together pieces of programs written at different time or by different people. The same variable may be declared as ranging over different sorts by different \#domain statements used in different programs. So the process of merging these programs requires renaming of variables. This concern was addressed by Marcello Balduccini [10] whose system, RSig, provided an ASP programmer with means for specifying sorts of parameters of the language predicates\(^3\). RSig is a simple extension of ASP which does not require any shift in perspective and involves only minor changes in existing programs. Our new language, \( \text{SPARC} \), can be viewed as a simple modification of RSig. In particular we propose to separate definition of sorts from the rest of the program and use this separation to improve the type checking and grounding procedure.

2 The Syntax and Semantics of \( \text{SPARC} \)

In this section we define a simple variant of \( \text{SPARC} \) which contains only one predefined sort \texttt{nat} of natural numbers. Richer variants may contain other predefined sorts with precise syntax which would be described in their manuals. The discussion will be sufficiently detailed to serve as the basis for the implementation of \( \text{SPARC} \) reasoning system.

\(^3\) In addition, RSig provides simple means for structuring a program into modules which we will not consider here.
Let \( \mathcal{L} \) be a language defined by the following grammar rules:

\[
\begin{align*}
<\text{identifier}> & : - <\text{small_letter}> | <\text{identifier}><\text{letter}> | <\text{identifier}><\text{digit}> \\
<\text{variable}> & : - <\text{capital_letter}> | <\text{variable}><\text{letter}> | <\text{variable}><\text{digit}> \\
<\text{non_zero_digit}> & : - 1|...|9 \\
<\text{digit}> & : - 0 | <\text{non_zero_digit}> \\
<\text{positive_integer}> & : - <\text{non_zero_digit}> | <\text{positive_integer}><\text{digit}> \\
<\text{natural_number}> & : - 0 | <\text{positive_integer}> \\
<\text{op}> & : + | - | * | \text{mod} \\
<\text{arithmetic_term}> & : - <\text{variable}> | <\text{natural_number}> | <\text{arithmetic_term}><\text{op}><\text{arithmetic_term}> | (\text{arithmetic_term}) \\
<\text{symbolic_function}> & : - <\text{identifier}> \\
<\text{symbolic_constant}> & : - <\text{identifier}> \\
<\text{symbolic_term}> & : - <\text{variable}> | <\text{symbolic_constant}> | <\text{symbolic_function}> (<\text{term}>,...,<\text{term}>) \\
<\text{term}> & : - <\text{symbolic_term}> | <\text{arithmetic_term}> \\
<\text{arithmetic_rel}> & : - = | != | > | >= | < | <= \\
<\text{pred_symbol}> & : - <\text{identifier}> \\
<\text{atom}> & : - <\text{pred_symbol}> (<\text{term}>,...,<\text{term}>) | <\text{arithmetic_term}><\text{arithmetic_rel}><\text{arithmetic_term}> | <\text{symbolic_term}> = <\text{symbolic_term}> | <\text{symbolic_term}> != <\text{symbolic_term}> \\
\end{align*}
\]

Note that relations = and != are defined on pairs of arithmetic and pairs of non-arithmetic terms. The first is a predefined arithmetic equality, i.e. \( 2 + 3=5 \), \( 2 + 1!=1 \), etc. The second is an identity relation\(^4\). By a ground term we mean a term containing no variables and no symbols for arithmetic functions [11].

From now on we assume a language \( \mathcal{L} \) with a fixed collection of symbolic constants and predicate symbols. A \( \text{SPARC} \) program parametrized by \( \mathcal{L} \) consists of three consecutive parts:

\[
\begin{align*}
<\text{program}> & : - \\
<\text{sorts definition}> & \\
<\text{predicates declaration}> & \\
<\text{program rules}> & \\
\end{align*}
\]

The first part of the program starts with the keywords:

\textit{sorts definition}

and is followed by the sorts definition:

\(^4\) In the implementation non-arithmetic identity should be restricted to comply with the syntax of lparse and other similar grounders.
Definition 1. By sort definition in $L$ we mean a collection $\Pi_s$ of rules of the form
\[ a_0 \leftarrow a_1, ..., a_m, \text{not } a_{m+1}, ..., \text{not } a_n. \]
such that
- $a_i$ are atoms of $L$ and $a_0$ contains no arithmetic relations;
- $\Pi_s$ has a unique answer set $S$.
- For every symbolic ground term $t$ of $L$ there is a unary predicate $s$ such that $s(t) \in S$.
- Every variable occurring in the negative part of the body, i.e. in at least one of the atoms $a_{m+1}, ..., a_n$, occurs in atom $a_i$ for some $0 < i \leq m$.

Predicate $s$ such that $s(t) \in S$ is called a defined sort of $t$. The language can also contain predefined sorts, in our case $\text{nat}$. Both, defined and predefined sorts will be referred to simply as sorts. (Note that a term $t$ may have more than one sort.)

The last condition of the definition is used to avoid unexpected reaction of the program to introduction of new constants (see example in the introduction). The condition was introduced in [8] where the authors proved that every program $\Pi$ satisfying this condition has the following property, called language independence: for every sorted signatures $\Sigma_1$ and $\Sigma_2$ groundings of $\Pi$ with respect to $\Sigma_1$ and $\Sigma_2$ have the same answer sets. This of course assumes that every rule of $\Pi$ can be viewed as a rule in $\Sigma_1$ and $\Sigma_2$.

The second part of a SPARC program starts with a keyword

`predicates declaration`

and is followed by statements of the form

`pred_symbol(sort,...,sort)`

We only allow one declaration per line. Predicate symbols occurring in the declaration must differ from those occurring in sorts definition. Finally, multiple declarations for one predicate symbol with the same arity are not allowed.

The third part of a SPARC program starts with a keyword

`program rules`

and is followed by a collection $\Pi_r$ of regular and consistency restoring rules of SPARC defined as follows:

regular rule:
\[ l_0 \lor \ldots \lor l_m \leftarrow l_{m+1}, \ldots, l_k, \text{not } l_{k+1} \ldots \text{not } l_n \quad (1) \]

As usual by $S$ we mean answer set of a ground program obtained from $\Pi$ by replacing its variables with ground terms of $L$. We assume that the program has non-empty Herbrand universe.
CR-rule:

\[ l_0 \leftarrow l_1, \ldots, l_k, \text{not} l_{k+1} \ldots \text{not} l_n \]  

(2)

where \( l \)'s are literals of \( \mathcal{L} \). Literals occurring in the heads of the rules must not be formed by predicate symbols occurring in \( \Pi_s \). In this paper, \( \leftarrow \) and \( \vdash \) are used interchangeably, so are \( \vdash \) and \( + \).

As expected, program \( \Pi_r \) is viewed as a shorthand for the set of all its ground instances which respect the sorts defined by \( \Pi_s \). Here is the precise definition of this notion.

**Definition 2.** Let \( gr(r) \) be a ground instance of a rule \( r \) of \( \Pi_r \), i.e. a rule obtained from \( r \) by replacing its variables by ground terms of \( \mathcal{L} \). We’ll say that \( gr(r) \) respects sorts of \( \Pi_s \) if every occurrence of an atom \( p(t_1, \ldots, t_n) \) of \( gr(r) \) satisfies the following condition: if \( p(s_1, \ldots, s_n) \) is the predicate declaration of \( p \) then \( t_1, \ldots, t_n \) are terms of sorts \( s_1, \ldots, s_n \) respectively. By \( gr(\Pi_r) \) we mean the collection of all ground instances of rules of \( \Pi_r \) which respect sorts of \( \Pi_s \).

Note that according to our definition \( gr(r) \) may be empty. This happens, for instance, for a rule which contains atoms \( p_1(X) \) and \( p_2(X) \) where \( p_1 \) and \( p_2 \) require parameters from disjoint sorts.

Let us now define answer sets of a ground \( S\text{PARC} \) program \( \Pi \). We assume that the readers are familiar with the definition of answer sets for standard ASP programs. Readers unfamiliar with the intuition behind the notion of consistency restoring rules of CR-Prolog are referred to the Appendix.

First we will need some notation. The set of regular rules of a \( S\text{PARC} \) program \( \Pi \) will be denoted by \( R \); the set of cr-rules of \( \Pi \) will be denoted by \( CR \). By \( \alpha(r) \) we denote a regular rule obtained from a consistency restoring rule \( r \) by replacing \( \leftarrow \) by \( \vdash \); \( \alpha \) is expanded in a standard way to a set \( X \) of cr-rules, i.e. \( \alpha(X) = \{ \alpha(r) : r \in X \} \).

**Definition 3. (Abductive Support)**

A collection \( X \) of cr-rules of \( \Pi \) such that

1. \( R \cup \alpha(X) \) is consistent (i.e. has an answer set) and
2. any \( R_0 \) satisfying the above condition has cardinality which is greater than or equal to that of \( R \)

is called an abductive support of \( \Pi \).

**Definition 4. (Answer Sets of \( S\text{PARC} \) Programs)**

A set \( A \) is called an answer set of \( \Pi \) if it is an answer set of a regular program \( R \cup \alpha(X) \) for some abductive support \( X \) of \( \Pi \).

---

6 By a literal we mean an atom \( a \) or its negation \( \neg a \). Note in this paper, we use \( \neg \) and \( - \) interchangeably.
To complete the definition of syntax and semantics of a \textit{SPARC} program we need to note that though such program is defined with respect to some language \( L \) in practice this language is extracted from the program. We always assume that terms of \( L \) defined by a \textit{SPARC} program \( P \) are arithmetic terms and terms defined by the sorts definition\(^7\); predicate symbols are those occurring in sorts definition and predicate declaration. Now we are ready to give an example of a \textit{SPARC} program.

\textit{Example 1.} \textit{[SPARC programs]}
Consider a \textit{SPARC} program \( P_1 \):

\begin{verbatim}
sorts definition
s1(1).
s1(2).
s2(X+1) :-
    s1(X).
s3(f(X,Y)) :-
    s1(X),
    s1(Y),
    X != Y.
predicates declaration
p(s1)
q(s1,s3)
r(s1,s3)
program rules
p(X).
r(1,f(1,2)).
q(X,Y) :-
    p(X),
    r(X,Y).
\end{verbatim}

The sort declaration of the program defines ground terms 1, 2, 3, \( f(1,2), f(2,1) \) with the following defined sorts:

\begin{itemize}
  \item \( s_1 = \{1,2\} \)
  \item \( s_2 = \{2,3\} \)
  \item \( s_3 = \{f(1,2), f(2,1)\} \)
\end{itemize}

Of course, 1, 2, and 3 are also of the sort \texttt{nat}. The sort respecting grounding of the rules of \( \Pi \) is

\begin{verbatim}
p(1).
p(2).
r(1,f(1,2)).
q(1,f(1,2)) :-
    p(1),
    r(1,f(1,2)).
\end{verbatim}

\(^7\) A term \( t \) is defined by \( \Pi_s \) if for some sort \( s \), \( s(t) \) belongs to the answer set of \( \Pi_s \)
The answer set of the program is \{p(1), p(2), r(1, f(1, 2)), q(1, f(1, 2))\}. (We are not showing the sort atoms.)

Consider now a SPARC program $P_2$:

**sorts definition**

t(a, b).
t(c, 1).
s1(X) :- t(X, Y).
s2(Y) :- t(X, Y).
s3(a).

**predicates declaration**
p(s1, s2).

**program rules**
p(X, Y) :- s3(X), t(X, Y).

The sort respecting grounding of the program is

\[
p(a, b) :- s3(a), t(a, b).
\]

Its answer set is \{p(a, b), t(a, b)\}.

Another example can be obtained by restating the CR-Prolog program from Example 4 in the Appendix by adding sort definitions $s_1(a)$ and $s_2(d(a))$ and predicates declarations $p(s_1)$, $q(s_1)$, $c(s_1)$ and $ab(s_2)$. One can easily check that, as expected, the answer set of the resulting program is \{¬q(a), c(a), ¬p(a)\}.

3 Translation of SPARC Programs to DLV Programs

DLV [12] is one of the well developed solvers for ASP programs. We select DLV as the target language mainly because of its weak constraints [6] which can be used to represent cr-rules. A weak constraint is of the form

\[\sim l_1, \ldots, l_k, not l_{k+1} \ldots not l_n.\]

where $l_i$’s are literals. (Weak constraints of DLV allow preferences which we ignore here.) Informally, weak constraints can be violated, but as many of them should be
satisfied as possible. The *answer sets* of a program $P$ with a set $W$ of weak constraints are those of $P$ which minimize the number of violated weak constraints.

We first introduce some notations before presenting the translation algorithm.

**Definition 5. (DLV counterparts of SPARC programs)**
A DLV program $P_2$ is a counterpart of SPARC program $P_1$ if answer sets of $P_1$ and $P_2$ coincide on literals from the language of $P_1$.

**Definition 6.** Given a SPARC program $P$, we associate a unique number to each of its cr-rules. The name of a cr-rule $r$ of $\Pi$ is a term $rn(i,X_1,...,X_n)$ where $rn$ is a new function symbol, $i$ is the unique number associated with $r$, and $X_1,...,X_n$ is the list of distinct variables occurring in $r$.

For instance, if rule $p(X,Y) \leftarrow q(Z,X,Y)$ is assigned number 1 then its name is $rn(1,X,Y,Z)$.

In what follows we describe a translation of SPARC programs into their DLV counterparts.

**Algorithm 1 (SPARC program translation)**

**Input:** a SPARC program $P_1$.

**Output:** a DLV counterpart $P_2$ of $P_1$.

1. Set variable $P_2$ to $\emptyset$, and let $appl/1$ be a new predicate not occurring in $P_1$.
2. Add all rules of the sorts definition part of $P_1$ to $P_2$.
3. For any program rule $r$ of $P_1$,
   3.1. Let $s = \{s_1(t_1),...,s_n(t_n) \mid p(t_1,...,t_n) \text{ occurs in } r \text{ and } p(s_1,...,s_n) \in P_1\}$,
      and let rule $r'$ be the result of adding all elements of $s$ to the body of $r$.
   3.2. If $r'$ is a regular rule, add it to $P_2$.
   3.3. If $r'$ is a cr-rule of the form
      \[ q \leftarrow \text{body}. \]
      add to $P_2$ the rules
      \[ \text{appl}(rn(i,X_1,...,X_n)) \lor \neg \text{appl}(rn(i,X_1,...,X_n)) :\leftarrow \text{body}. \]
      \[ :\sim \text{appl}(rn(i,X_1,...,X_n)), \text{body}. \]
      \[ q :\leftarrow \text{appl}(rn(i,X_1,...,X_n)), \text{body}. \]
      where $rn(i,X_1,...,X_n)$ is the name of $r$.

The intuitive idea behind the rules added to $P_2$ in 3.3. is as follows: $\text{appl}(rn(i,X_1,...,X_n))$ holds if the cr-rule $r$ is used to obtain an answer set of the SPARC program; the first rule says that $r$ is either used or not used; the second rule, a weak constraint, guarantees that $r$ is not used if possible, and the last rule allows the use of $r$ when necessary.

The correctness of the algorithm is guaranteed by the following theorem whose complete proof can be found at http://www.cs.ttu.edu/research/krlab/pdfs/papers/sparc-proof.pdf.
Theorem 1. A DLV program $P_2$ obtained from a SPARC program $P_1$ by Algorithm 1 is a DLV counterpart of $P_1$.

The translation can be used to compute an answer set of SPARC program $P$.

Algorithm 2 (Computing an answer set of a SPARC program)

**Input:** a SPARC program $P$.

**Output:** an answer set of $P$.

1. Translate $P$ into its DLV counterpart $P'$.
2. Use DLV to find an answer set $S$ of $P'$.
3. Drop all literals with predicate symbol appl from $S$ and return the new set.

Example 2. To illustrate the translation and the algorithm, consider the following program.

```prolog
sorts definition
s(a).
predicates declaration
p(s)
q(s)
program rules
p(X) :- not q(X).
¬p(X).
q(X) :+ .
After step 2 of Algorithm 1, $P'$ becomes:
s(a).

After the execution of the loop 3 of this algorithm for the first and second program rule, $P'$ becomes
s(a).
p(X) :- not q(X),s(X).
¬p(X):- s(X).

Assuming the only cr-rule is numbered by 1, after the algorithm is applied to the third rule, $P'$ becomes
s(a).
p(X) :- not q(X),s(X).
¬p(X):- s(X).
appl(rn(1,X)) ∨ ¬appl(rn(1,X)) :- s(X).
:+ appl(rn(1,X)), s(X).
q(X) :- appl(rn(1,X)), s(X).

Given the program $P'$, DLV solver returns an answer set

$$\{s(a), appl(rn(1,a)), q(a), ¬p(a)\}$$

After dropping $\text{appl}(rn(1,a))$ from this answer set, we obtain an answer set

$$\{s(a), q(a), ¬p(a)\}$$

for the original program.
4 Experimental Results

We have implemented a *SPARC* program solver, called *crTranslator* (available from the link in [13]), based on the proposed translation approach. CRModels2 [7] is the state of the art solver for CR-prolog programs. To compare the performance of the DLV based solver to CRModels2, we use the classical benchmark of the reaction control system for the space shuttle [3] and new benchmarks such as representing and reasoning with intentions [5], and the shortest path problem.

Clock time, in seconds, is used to measure the performance of the solvers. Since the time complexity of translation is low, the recorded problem solving time does not include the translation time.

In this experiment, we use DLV build BEN/Dec 21 2011 and CRModels2 2.0.12 [14] which uses ASP solver Clasp 2.0.5 with grounder Gringo 3.0.4 [15]. The experiments are carried out on a computer with Intel Core 2 Duo CPU E4600 at 2.40 Ghz, 3GB RAM, and Cygwin 1.7.10 on Windows XP.

4.1 The First Benchmark: Programs for Representing and Reasoning with Intentions

Recently, CR-Prolog has been employed to represent and reason with intentions [5]. We compare crTranslator with CRModels2 on the following scenarios proposed in [5]: Consider a row of four rooms, \( r_1, r_2, r_3, r_4 \) connected by doorways, such that an agent may move along the row from one room to the next. We say that two people *meet* if they are located in the same room. Assume that initially our agent Bob is in \( r_1 \) and he intends to meet with John who, as Bob knows, is in \( r_3 \). This type of intention is frequently referred to as an intention to achieve the goal. The first task is to design a simple plan for Bob to achieve this goal: move from \( r_1 \) to \( r_2 \) and then to \( r_3 \). Assuming that as Bob is moving from \( r_1 \) to \( r_2 \), John moves from \( r_3 \) to \( r_2 \), the second task is to recognize the unexpected achievement of his goal and not continue moving to \( r_3 \). Programs to implement these two tasks are given as \( B_0 \) and \( B_1 \) respectively in [5].

<table>
<thead>
<tr>
<th>Tasks</th>
<th>CRModels2</th>
<th>crTranslator</th>
</tr>
</thead>
<tbody>
<tr>
<td>task 1</td>
<td>104</td>
<td>11</td>
</tr>
<tr>
<td>task 2</td>
<td>104</td>
<td>101</td>
</tr>
</tbody>
</table>

Fig. 1. CPU time for intention reasoning benchmark using CRModels2 and crTranslator

In this experiment, crTranslator has a clear advantage over CR-Models2 on task 1 and similar performance on task 2.
4.2 The Second Benchmark: Reaction Control System of Space Shuttle

USA-Smart is a CR-prolog program to find plans with improved quality for the operation of the Reaction Control System (RCS) of the Space Shuttle. Plans consist of a sequence of operations to open and close the valves controlling the flow of propellant from the tanks to the jets of the RCS.

In our experiment, we used the USA-Smart program with four instances: fmc1 to fmc4 [16]. The \textit{SPARC} variant of the USA-Smart program is written as close as possible to USA-smart. The results of the performance of crTranslator and CRModels for these programs are listed in Figure 2.

<table>
<thead>
<tr>
<th>Instances</th>
<th>CRModels2</th>
<th>crTranslator</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmc1</td>
<td>29.0</td>
<td>74.0</td>
</tr>
<tr>
<td>fmc2</td>
<td>11.6</td>
<td>34.0</td>
</tr>
<tr>
<td>fmc3</td>
<td>6.0</td>
<td>8907.0</td>
</tr>
<tr>
<td>fmc4</td>
<td>30.5</td>
<td>22790.0</td>
</tr>
</tbody>
</table>

\textbf{Fig. 2.} CPU time for reaction control system using CRModels2 and crTranslator

We note that these instances have small abductive supports (with sizes of the supports less than 9) and relatively large number of cr-rules (with more than 1200). This can partially explain why CRModels2 is faster because it finds the abductive support by exhaustive enumeration of the candidate supports starting from size 0 to all cr-rules in an increasing manner.

4.3 The Third Benchmark: Shortest Path Problem

Given a simple directed graph and a pair of distinct vertices of the graph, the shortest path problem is to find a shortest path between these two vertices. Given a graph with \(n\) vertices and \(e\) edges, its \textit{density} is defined as \(e/(n \times (n - 1))\). In our experiment, the problem instances are generated randomly based on the number of vertices and the density of the graph. The density of the graphs varies from 0.1 to 1 so that the shortest paths involve abductive supports of different sizes. To produce graphs with longer shortest path (which needs larger abductive supports), we zoom into the density between 0 to 0.1 with a step of 0.01. To reduce the time solving the problem instances, as density increases, we use smaller number of vertices. Given a graph, we define the \textit{distance} between a pair of vertices as the length of the shortest path between them. For any randomly generated graph, we select any two vertices such that their distance is the longest among those of all pairs of vertices. The problem is to find the shortest path between these two vertices.

The \textit{SPARC} programs and CR-prolog programs are written separately due to the difference between these two languages, but we make them as similar as possible.
use exactly the same cr-rules in both programs. The experimental results are listed in Figure 3.

From the results, CRModels2 is faster on a majority of cases. Again, crTranslator is faster when the size of the abductive support is large. The graphs with density between 0.02 and 0.03 have support size of 16 while the other graphs (except the one of density 0.01) have support sizes not more than 12. Further investigation is needed to have a better understanding of the performance difference between the two solvers.

<table>
<thead>
<tr>
<th>Number of vertices</th>
<th>Density</th>
<th>CRModels2</th>
<th>crTranslator</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>0.01</td>
<td>4.0</td>
<td>0.2</td>
</tr>
<tr>
<td>60</td>
<td>0.02</td>
<td>5.1</td>
<td>0.4</td>
</tr>
<tr>
<td>60</td>
<td>0.03</td>
<td>5.7</td>
<td>0.8</td>
</tr>
<tr>
<td>60</td>
<td>0.04</td>
<td>9.1</td>
<td>66.5</td>
</tr>
<tr>
<td>60</td>
<td>0.05</td>
<td>29.2</td>
<td>337.0</td>
</tr>
<tr>
<td>60</td>
<td>0.06</td>
<td>235.7</td>
<td>4451.8</td>
</tr>
<tr>
<td>40</td>
<td>0.07</td>
<td>7.4</td>
<td>19.9</td>
</tr>
<tr>
<td>40</td>
<td>0.08</td>
<td>8.4</td>
<td>154.6</td>
</tr>
<tr>
<td>40</td>
<td>0.09</td>
<td>7.0</td>
<td>32.6</td>
</tr>
<tr>
<td>30</td>
<td>0.1</td>
<td>6.0</td>
<td>16.8</td>
</tr>
<tr>
<td>30</td>
<td>0.2</td>
<td>39.9</td>
<td>9711.4</td>
</tr>
<tr>
<td>20</td>
<td>0.3</td>
<td>7.6</td>
<td>54.9</td>
</tr>
<tr>
<td>20</td>
<td>0.4</td>
<td>9.3</td>
<td>52.2</td>
</tr>
<tr>
<td>20</td>
<td>0.5</td>
<td>16.4</td>
<td>234.8</td>
</tr>
<tr>
<td>20</td>
<td>0.6</td>
<td>9.6</td>
<td>51.7</td>
</tr>
<tr>
<td>20</td>
<td>0.7</td>
<td>14.3</td>
<td>52.0</td>
</tr>
<tr>
<td>20</td>
<td>0.8</td>
<td>17.6</td>
<td>58.8</td>
</tr>
<tr>
<td>20</td>
<td>0.9</td>
<td>22.2</td>
<td>69.1</td>
</tr>
<tr>
<td>20</td>
<td>1.0</td>
<td>5.5</td>
<td>55.6</td>
</tr>
</tbody>
</table>

Fig. 3. CPU time for solving shortest path problem using CRModels2 and crTranslator

5 Conclusion and Future Work

This paper describes a sorted version of CR-Prolog called SPARC, presents a translation of consistency restoring rules of the language into weak constraints of DLV, and investigates the possibility of building efficient inference engines for SPARC based on this translation. This is a preliminary report. There is a number of steps which should be made to truly develop SPARC into a knowledge representation language of choice for teaching and applications. In particular we plan the following:

- Expand SPARC to include a number of useful language constructs beyond the original language of ASP such as aggregates and optimization constructs. In this
expansion, instead of committing to a particular syntax, we are planning to allow
users to select their favorite input language such as that of DLV or LPARSE or
GRINGO and provide the final system with support for the corresponding language.

– Provide SPARC with means to specify different preference relations between sets of
cr-rules, define and investigate answer sets minimal with respect to these prefer-
ence relations, and implement the corresponding SPARC solvers.

– Design and implement SPARC grounders to directly use the sort information pro-
vided by definitions and declaration of a program. The emphasis will be on error
checking and incrementality of the grounders.

– Investigate more efficient reasoning algorithms for SPARC. DLV uses a more ad-
vanced technique of branch and bound to process weak constraints while CRMod-
els employs a more primitive search algorithm. However, our experiments show
that the latter is not necessarily slower. Further understanding of these two ap-
proaches is expected to inspire new techniques for building more efficient solvers
for SPARC programs.

– Expand SPARC and its solvers to other extensions of ASP including ACC [17]
and P-log [4].

Acknowledgement

The work of Gelfond and Zhang was partially supported by NSF grant IIS-1018031.

Appendix: CR-Prolog

This Appendix contains a short informal introduction to CR-Prolog. The version dis-
ussed here is less general than the standard version — in particular it omits the treat-
ment of preferences which is a task orthogonal to the goals of this paper. One of the
original goals of the CR-Prolog was to provide a construct allowing a simple repre-
sentation of exceptions to defaults, sometimes referred to as indirect exceptions. Intu-
itively, these are rare exceptions that come into play only as a last resort, to restore
the consistency of the agent’s world view when all else fails. The representation of indi-
rect exceptions seems to be beyond the power of “pure” ASP [1] which prompted the
introduction of cr-rules. To illustrate the problem let us consider the following example.

Example 3. [Indirect Exception in ASP]
Consider an ASP representation of the default “elements of class $c$ normally have prop-
erty $p$”:

$$p(X) \leftarrow c(X), \text{not } ab(d(X)), \text{not } \neg p(X).$$

(where $d(X)$ is used as the name of the default) together with the rule

$$q(X) \leftarrow p(X).$$

and two observations:

$$c(a),$$

$$\neg q(a).$$
It is not difficult to check that this program is inconsistent. No rules allow the reasoner to prove that the default is not applicable to $a$ (i.e. to prove $ab(d(a))$) or that $a$ does not have property $p$. Hence the default must conclude $p(a)$. The second rule implies $q(a)$ which contradicts the observation.

There, however, seems to exist a commonsense argument which may allow a reasoner to avoid inconsistency, and to conclude that $a$ is an indirect exception to the default. The argument is based on the Contingency Axiom for default $d(X)$ which says that Any element of class $c$ can be an exception to the default $d(X)$ above, but such a possibility is very rare and, whenever possible, should be ignored. One may informally argue that since the application of the default to $a$ leads to a contradiction, the possibility of $x$ being an exception to $d(a)$ cannot be ignored and hence $a$ must satisfy this rare property.

The CR-Prolog is an extension of ASP capable of encoding and reasoning about such rare events. In addition to regular logic programming rules the language allows consistency restoring rules of the form

$$ l_0 \leftarrow l_1, \ldots, l_k, \neg l_{k+1}, \ldots, \neg l_n $$

where $l$’s are literals. Intuitively, the rule says that if the reasoner associated with the program believes the body of the rule, then it “may possibly” believe its head. However, this possibility may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program.

The following Example shows the use of CR-Prolog for representing defaults and their indirect exceptions.

**Example 4.** [Indirect Exception in CR-Prolog]
The CR-Prolog representation of default $d(X)$ may look as follows

$$ p(X) \leftarrow c(X), \neg ab(d(X)), \neg p(X). $$

$$ \neg p(X) \leftarrow \neg c(X). $$

The first rule is the standard ASP representation of the default, while the second rule expresses the Contingency Axiom for default $d(X)$. Consider now a program obtained by combining these two rules with an atom $c(a)$. Assuming that $a$ is the only constant in the signature of this program, the program’s answer set will be \{c(a), p(a)\}. Of course this is also the answer set of the regular part of our program. (Since the regular part is consistent, the Contingency Axiom is ignored.) Let us now expand this program by the rules

$$ q(X) \leftarrow p(X). $$

$$ \neg q(a). $$

The regular part of the new program is inconsistent. To save the day we need to use the Contingency Axiom for $d(a)$ to form the abductive support of the program. As a result the new program has the answer set \{\neg q(a), c(a), \neg p(a)\}. The new information does
not produce inconsistency as in the analogous case of ASP representation. Instead the program withdraws its previous conclusion and recognizes $a$ as a (strong) exception to default $d(a)$.

The possibility to encode rare events which may serve as unknown exceptions to defaults proved to be very useful for various knowledge representation tasks, including planning, diagnostics, and reasoning about the agent’s intentions.

References

16. USA-Smart: http://marcy.cjb.net/rcs-asp/rcs/
Language ASP\{f\} with Arithmetic Expressions and Consistency-Restoring Rules

Marcello Balduccini\(^1\) and Michael Gelfond\(^2\)

\(^1\) Kodak Research Laboratories
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com
\(^2\) Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
michael.gelfond@ttu.edu

Abstract In this paper we continue the work on our extension of Answer Set Programming by non-Herbrand functions and add to the language support for arithmetic expressions and various inequality relations over non-Herbrand functions, as well as consistency-restoring rules from CR-Prolog. We demonstrate the use of this latest version of the language in the representation of important kinds of knowledge.

1 Introduction

In this paper we describe an extension of Answer Set Programming (ASP) [12,16,4] called ASP\{f,cr\}. This work continues our research on the introduction of non-Herbrand functions in ASP.

In logic programming, functions are typically interpreted over the Herbrand Universe, with each functional term \(f(x)\) mapped to its own canonical syntactical representation. That is, in most logic programming languages, the value of an expression \(f(x)\) is \(f(x)\) itself, and thus, if equality is understood as identity, \(f(x) = 2\) is false. This type of functions, the corresponding languages and efficient implementation of solvers is the subject of a substantial amount of research (we refer the reader to e.g. [8,6]).

When representing certain kinds of knowledge, however, it is sometimes convenient to use functions with non-Herbrand domains (non-Herbrand functions for short), i.e. functions that are interpreted over domains other than the Herbrand Universe. For example, when describing a domain in which people enter and exit a room over time, it may be convenient to represent the number of people in the room at step \(s\) by means of a function \(\text{occupancy}(s)\) and to state the effect of a person entering the room by means of a statement such as

\[
\text{occupancy}(S + 1) = O + 1 \leftarrow \text{occupancy}(S) = O
\]

where \(S\) is a variable ranging over the possible time steps in the evolution of the domain and \(O\) ranges over natural numbers.
Of course, in most logic programming languages, non-Herbrand functions can still be represented, but the corresponding encodings are not as natural and declarative as the one above. For instance, a common approach consists in representing the functions of interest using relations, and then characterizing the functional nature of these relations by writing auxiliary axioms. In ASP, one would encode the above statement by (1) introducing a relation \( \text{occupancy}'(s, o) \), whose intuitive meaning is that \( \text{occupancy}'(s, o) \) holds iff the value of \( \text{occupancy}(s) \) is \( o \); and (2) re-writing the original statement as a rule

\[
\text{occupancy}'(S + 1, O + 1) \leftarrow \text{occupancy}'(S, O).
\] (1)

The characterization of the relation as representing a function would be completed by an axiom such as

\[
\neg \text{occupancy}'(S, O') \leftarrow \text{occupancy}'(S, O), O \neq O'.
\] (2)

which intuitively states that \( \text{occupancy}(s) \) has a unique value. The disadvantage of this representation is that the functional nature of \( \text{occupancy}'(s, o) \) is only stated in (2). When reading (1), one is given no indication that \( \text{occupancy}'(s, o) \) represents a function – and, before finding statements such as (2), one can make no assumption about the functional nature of the relations in a program when a combination of (proper) relations and non-Herbrand functions are present. Moreover, in ASP relational encodings of functions often pose performance issues. For example, the grounding of rule (2) grows with \( O(|D|^2) \) where \( D \) is the range of variables \( O \) and \( O' \). If \( |D| \) is large, which is often the case especially with numerical variables, the size of the grounding can affect very negatively the overall solver performance.

Various extensions of ASP with non-Herbrand functions exist in the literature. In [7], Quantified Equilibrium Logic is extended with support for equality. A subset of the general language, called FLP, is then identified, which can be translated into normal logic programs. Such translation makes it possible to compute the answer sets of FLP programs using ASP solvers, although the performance issues due to the size of the grounding remain. [14] proposes instead the use of second-order theories for the definition of the semantics of the language. Again, a transformation is (partially) described, which removes non-Herbrand functions and makes it possible to use ASP solvers for the computation of the answer sets of programs in the extended language. As with the previous approach, the performance issues are present. In [15,17] the semantics is based on the notion of reduct as in the original ASP semantics [12]. For the purpose of computing answer sets, a translation is defined, which maps programs of the language from [15,17] to constraint satisfaction problems, so that CSP solvers can be used for the computation of the answer sets of programs in the extended language. Finally, the language of CLINGCON [9] extends ASP with elements from constraint satisfaction. The CLINGCON solver finds the answer sets of a program by interleaving the computations of an ASP solver and of a CSP solver. All the approaches except for [7] support only total functions. While the approaches from [15,17,9] are computationally efficient, the approaches of [7,14], based on translations to ASP, are affected by performance issues due to the size of the grounding. Finally, in [1,2] we have proposed an extension of ASP with non-Herbrand functions, called ASP\{f\}, that supports partial functions and is computationally more efficient than [7].
In the present paper, we extend the definition of $\text{ASP}\{f\}$ from [1,2] further, by adding to it support for full-fledged arithmetic expressions and for consistency restoring rules from CR-Prolog [3]. We also contribute our perspective to the current debate on the usefulness of partial vs. total functions and on the role of non-Herbrand languages in general by demonstrating the use of our extended language for the representation of important types of knowledge and for the encoding of some classical scenarios, pointing out the differences with other approaches and with ASP encodings.

The rest of the paper is organized as follows. In the next section we extend $\text{ASP}\{f\}$ with full-fledged arithmetic expressions. In the following section we introduce consistency-restoring. The resulting language is called $\text{ASP}\{f, cr\}$. Next, we address high-level issues of knowledge representation in $\text{ASP}\{f, cr\}$ and we demonstrate the use of $\text{ASP}\{f, cr\}$ for the formalization of some classical problems. Finally, we draw conclusions and discuss future work.

2 $\text{ASP}\{f\}$ with Arithmetic Expressions

In this section we summarize the syntax and the semantics of $\text{ASP}\{f\}$ [1,2], and extend the language with support for arithmetic expressions over non-Herbrand functional terms. For simplicity, in the rest of this paper we drop the attribute “non-Herbrand” and simply talk about functions and (functional) terms.

The syntax of $\text{ASP}\{f\}$ is based on a signature $\Sigma = \langle C, F, R \rangle$ whose elements are, respectively, finite disjoint sets of constants, function symbols and relation symbols. Some constants and function symbols are numerical (e.g. numerical constants 1 and 5) and have the standard interpretation. A simple term is an expression $f(c_1, \ldots, c_n)$ where $f \in F$, and $c_i$’s are 0 or more constants. An arithmetic term is either a simple term where $f$ is a numerical function, or an expression constructed from such simple terms and numerical constants using arithmetic operations, such as $(f(c_1) + g(c_2))/2$ and $|f(c_1) - g(c_2)|$. Simple terms and arithmetic terms are called terms. An atom is an expression $r(c_1, \ldots, c_n)$, where $r \in R$, and $c_i$’s are constants. The set of all simple terms that can be formed from $\Sigma$ is denoted by $S$; the set of all atoms from $\Sigma$ is denoted by $A$. A term-atom, or t-atom, is an expression of the form $f \circ g$, where $f$ and $g$ are terms and $\circ \in \{=, \neq, \leq, <, >, \geq\}$. A seed t-atom is a t-atom of the form $f = v$ such that $f$ is a simple term and $v$ is a constant. All other t-atoms are called dependent.

A regular literal is an atom $a$ or its strong negation $\neg a$. A literal is either an atom $a$, its strong negation $\neg a$, or a t-atom. Any literal that is not a dependent t-atom is called seed literal.

A rule $r$ is a statement of the form:

$$h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$$

where $h$ is a seed literal and $l_i$’s are literals. Similarly to ASP, the informal reading of $r$ is that a rational agent who believes $l_1, \ldots, l_m$ and has no reason to believe $l_{m+1}, \ldots, l_n$ must believe $h$.

$^3$ In the rest of the paper, whether an element of $\Sigma$ is numerical will be clear from the context.
Given rule \( r \), head\((r)\) denotes \{\( h \)\}; body\((r)\) denotes \{\( l_1, \ldots, \neg l_n \)\}; pos\((r)\) denotes \{\( l_1, \ldots, l_m \)\}; neg\((r)\) denotes \{\( l_{m+1}, \ldots, l_n \)\}.

A constraint is a special type of regular rule with an empty head, informally meaning that the condition described by the body of the constraint must never be satisfied. A constraint is considered a shorthand of \( \bot \leftarrow l_1, \ldots, l_m, \neg l_{m+1}, \ldots, \neg l_n, \neg \bot \), where \( \bot \) is a fresh atom.

A program is a pair \( \Pi = (\Sigma, P) \), where \( \Sigma \) is a signature and \( P \) is a set of rules. Whenever possible, in this paper the signature is implicitly defined from the rules of \( \Pi \), and \( \Pi \) is identified with its set of rules. In that case, the signature is denoted by \( \Sigma(\Pi) \) and its elements by \( \mathcal{C}(\Pi) \), \( \mathcal{F}(\Pi) \) and \( \mathcal{R}(\Pi) \). A rule \( r \) is positive if \( \text{neg}(r) = \emptyset \).

A program \( \Pi \) is positive if every \( r \in \Pi \) is positive. A program \( \Pi \) is also t-atom free if no t-atoms occur in the rules of \( \Pi \).

As in ASP, variables can be used for a more compact notation. The grounding of a rule \( r \) is the set of all the rules (its ground instances) obtained by replacing every variable of \( r \) with an element of \( \mathcal{C} \) and by performing any arithmetic operation over numerical constants. For example, one of the groundings of \( p(X + Y) \leftarrow r(X), q(Y) \) is \( p(5) \leftarrow r(3), q(2) \). The grounding of a program \( \Pi \) is the set of the groundings of the rules of \( \Pi \). A syntactic element of the language is ground if it contains neither variables nor arithmetic operations over numerical constants and non-ground otherwise. For example, \( p(5) \) is ground while \( p(X + Y) \) and \( p(3 + 2) \) are non-ground.

The semantics of a non-ground program is defined to coincide with the semantics of its grounding. The semantics of ground \( \text{ASP}\{f\} \) programs is defined below. It is worth noting that the semantics of \( \text{ASP}\{f\} \) is obtained from that of ASP in [12] by simply extending entailment to t-atoms.

In the rest of this section, we consider only ground terms, literals, rules and programs and thus omit the word “ground.” A set \( S \) of seed literals is consistent if (1) for every atom \( a \in \mathcal{A}, \{a, \neg a\} \not\subseteq S \), (2) for every term \( t \in \mathcal{S} \) and \( v_1, v_2 \in \mathcal{C} \) such that \( v_1 \neq v_2 \), \( \{t = v_1, t = v_2\} \not\subseteq S \). Hence, \( S_1 = \{p, \neg q, f = 3\} \) and \( S_2 = \{q, f = 3, g = 2\} \) are consistent, while \( \{p, \neg p, f = 3\} \) and \( \{q, f = 3, f = 2\} \) are not.

The value of a simple term \( t \) w.r.t. a consistent set \( S \) of seed literals (denoted by \( \text{val}_S(t) \)) is \( v \) iff \( t = v \in S \). If, for every \( v \in \mathcal{C}, t = v \not\in S \), the value of \( t \) w.r.t. \( S \) is undefined. The value of an arithmetic term \( t \) w.r.t. \( S \) is obtained by applying the usual rules of arithmetic to the values of the terms in \( t \) w.r.t. \( S \), if the values of all the terms in \( t \) are defined; otherwise its value is undefined. Finally, the value of a constant \( v \in \mathcal{C} \) w.r.t. \( S \) (\( \text{val}_S(v) \)) is \( v \) itself. For example, given \( S_1 \) and \( S_2 \) as above, \( \text{val}_{S_1}(f) \) is 3 and \( \text{val}_{S_2}(g) \) is 2, whereas \( \text{val}_{S_1}(g) \) is undefined. Given \( S_1 \) and a signature with \( \mathcal{C} = \{0, 1\} \), \( \text{val}_{S_1}(1) = 1 \).

A literal \( l \) is satisfied by a consistent set \( S \) of seed literals under the following conditions: (1) if \( l \) is a seed literal, then \( l \) is satisfied by \( S \) iff \( l \in S \); (2) if \( l \) is a dependent

---

4 The replacement is with constants of suitable sort. We omit the details of this process to save space.

5 This definition does not adequately capture the value of expressions such as \( 0 * f \) in the presence of undefined terms. We plan to address this and some related issues in a later paper.
t-atom of the form \( f \circ \odot g \), then \( l \) is satisfied by \( S \) iff both \( \text{val}_S(f) \) and \( \text{val}_S(g) \) are defined, and they satisfy the equality or inequality relation \( \circ \odot \) according to the usual arithmetic interpretation. Thus, seed literals \( q \) and \( f = 3 \) are satisfied by \( S_2 \); \( f \neq g \) is also satisfied by \( S_2 \) because \( \text{val}_{S_2}(f) \) and \( \text{val}_{S_2}(g) \) are defined, and \( \text{val}_{S_2}(f) \) is different from \( \text{val}_{S_2}(g) \). Conversely, \( f = g \) is not satisfied, because \( \text{val}_{S_2}(f) \) is different from \( \text{val}_{S_2}(g) \). The t-atom \( f \neq h \) is also not satisfied by \( S_2 \), because \( \text{val}_{S_2}(h) \) is undefined. When a literal \( l \) is satisfied (resp., not satisfied) by \( S \), we write \( S \models l \) (resp., \( S \not\models l \)).

An extended literal is a literal \( l \) or an expression of the form \( \neg l \). An extended literal \( \neg l \) is satisfied by a consistent set \( S \) of seed literals \( (S \models \neg l) \) if \( S \not\models l \). Similarly, \( S \not\models \neg l \) if \( S \models l \). Considering set \( S_2 \) again, extended literal \( \neg f = h \) is satisfied by \( S_2 \), because \( f = h \) is not satisfied by \( S_2 \).

Finally, a set \( E \) of extended literals is satisfied by a consistent set \( S \) of seed literals \( (S \models E) \) if \( S \models e \) for every \( e \in E \).

Next, we define the semantics of ASP\{f\}. A set \( S \) of seed literals is closed under positive rule \( r \) if \( S \models h \), where \( \text{head}(r) = \{ h \} \), whenever \( S \models \text{pos}(r) \). Hence, set \( S_2 \) described earlier is closed under \( f = 3 \leftarrow g \neq 1 \) and (trivially) under \( f = 2 \leftarrow r \), but it is not closed under \( p \leftarrow f = 3 \), because \( S_2 \models f = 3 \) but \( S_2 \not\models p \). \( S \) is closed under \( \Pi \) if it is closed under every rule \( r \in \Pi \).

**Definition 1.** A set \( S \) of seed literals is an answer set of a positive program \( \Pi \) if it is consistent and closed under \( \Pi \), and is minimal (w.r.t. set-theoretic inclusion) among the sets of seed literals that satisfy such conditions.

Thus, the program \( \{ p \leftarrow f = 2. \ f = 2. \ q \leftarrow q. \} \) has one answer sets, \( \{ f = 2, p \} \). The set \( \{ f = 2 \} \) is not closed under the first rule of the program, and therefore is not an answer set. The set \( \{ f = 2, p, q \} \) is also not an answer set, because it is not minimal (it is a proper superset of another answer set). Notice that positive programs may have no answer set. For example, the program \( \{ f = 3. \ f = 2 \leftarrow q. \ q. \} \) has no answer set. Programs that have answer sets (resp., no answer sets) are called consistent (resp., inconsistent).

Positive programs enjoy the following property:

**Proposition 1.** Every consistent positive ASP\{f\} program \( \Pi \) has a unique answer set.

Next, we define the semantics of arbitrary ASP\{f\} programs.

**Definition 2.** The reduct of a program \( \Pi \) w.r.t. a consistent set \( S \) of seed literals is the set \( \Pi^S \) consisting of a rule \( \text{head}(r) \leftarrow \text{pos}(r) \) (the reduct of \( r \) w.r.t. \( S \)) for each rule \( r \in \Pi \) for which \( S \models \text{body}(r) \setminus \text{pos}(r) \).

**Example 1.** Consider a set of seed literals \( S_3 = \{ g = 3, f = 2, p, q \} \), and program \( \Pi_1 \):

\[
\begin{align*}
r_1 &: p \leftarrow f = 2, \text{not } g = 1, \text{not } h = 0. \\
r_2 &: q \leftarrow p, \text{not } g \neq 2. \\
r_3 &: g = 3. \\
r_4 &: f = 2.
\end{align*}
\]
and let us compute its reduct. For \( r_1 \), first we have to check if \( S_3 \models body(r_1) \setminus pos(r_1) \), that is if \( S_3 \models \neg g = 1, \neg h = 0 \). Extended literal \( \neg g = 1 \) is satisfied by \( S_3 \) only if \( S_3 \nvdash g = 1 \). Because \( g = 1 \) is a seed literal, it is satisfied by \( S_3 \) if \( g = 1 \in S_3 \). Since \( g = 1 \notin S_3 \), we conclude that \( S_3 \nvdash g = 1 \) and thus \( \neg g = 1 \) is satisfied by \( S_3 \).

In a similar way, we conclude that \( S_3 \models \neg h = 0 \). Hence, \( S_3 \models body(r_1) \setminus pos(r_1) \).

Therefore, the reduct of \( r_1 \) is \( p \leftarrow f = 2 \). For the reduct of \( r_2 \), notice that \( \neg g \neq 2 \) is not satisfied by \( S_3 \). In fact, \( S_3 \nvdash \neg g \neq 2 \) only if \( S_3 \nvdash \neg g \neq 2 \). However, it is not difficult to show that \( S_3 \models \neg g \neq 2 \): in fact, \( val_{S_3}(g) \) is defined and \( val_{S_3}(g) \neq 2 \).

Therefore, \( \neg g \neq 2 \) is not satisfied by \( S_3 \), and thus the reduct of \( \Pi_1 \) contains no rule for \( r_2 \). The reducts of \( r_3 \) and \( r_4 \) are the rules themselves. Summing up, \( \Pi_1^{S_3} \) is \( \{ r'_1 : p \leftarrow f = 2, r'_3 : g = 3, r'_4 : f = 2 \} \)

The semantics of arbitrary \( \text{ASP}\{f\} \) programs is given by the following definition:

**Definition 3.** Finally, a consistent set \( S \) of seed literals is an answer set of \( \Pi \) if \( S \) is the answer set of \( \Pi_1^S \).

**Example 2.** By applying the definitions given earlier, it is not difficult to show that an answer set of \( \Pi_1^{S_3} \) is \( \{ f = 2, g = 3, p \} = S_3 \). Hence, \( S_3 \) is an answer set of \( \Pi_1^{S_3} \).

Consider instead \( S_4 = S_3 \cup \{ h = 1 \} \). Clearly \( \Pi_1^{S_4} = \Pi_1^{S_3} \). From the uniqueness of the answer sets of positive programs, it follows immediately that \( S_4 \) is not an answer set of \( \Pi_1^{S_4} \).

Therefore, \( S_4 \) is not an answer set of \( \Pi_1 \).

### 3 ASP\{f,cr\}: Consistency-Restoring Rules in ASP\{f\}

In this section we extend \( \text{ASP}\{f\} \) by consistency-restoring rules from CR-Prolog [3]. We denote the extended language by \( \text{ASP}\{f,cr\} \). As discussed in the literature on CR-Prolog, consistency-restoring rules are convenient for the formalization of various types of knowledge and of reasoning tasks. Later in this paper we show how consistency-restoring rules are useful for the formalization of knowledge about non-Herbrand functions as well.

A consistency-restoring rule (or cr-rule) is a statement of the form:

\[
h \leftarrow l_1, \ldots, l_m, \neg l_{m+1}, \ldots, \neg l_n. \tag{4}\]

where \( h \) is a seed literal and \( l_i \)'s are literals. The intuitive reading of the statement is that a reasoner who believes \( \{ l_1, \ldots, l_m \} \) and has no reason to believe \( \{ l_{m+1}, \ldots, l_n \} \), may possibly believe \( h \). The implicit assumption is that this possibility is used as little as possible, only when the reasoner cannot otherwise form a non-contradictory set of beliefs.

By \( \text{ASP}\{f,cr\} \) program we mean a pair \( (\Sigma, \Pi) \), where \( \Sigma \) is a signature and \( \Pi \) is a set of rules and cr-rules over \( \Sigma \).

Given an \( \text{ASP}\{f,cr\} \) program \( \Pi \), we denote the set of its rules by \( \Pi^r \) and the set of its cr-rules by \( \Pi^{cr} \). By \( \alpha(r) \) we denote the rule obtained from cr-rule \( r \) by replacing symbol \( \leftarrow \) with \( \leftarrow \). Given a set of cr-rules \( R \), \( \alpha(R) \) denotes the set obtained by applying \( \alpha \) to each cr-rule in \( R \). The semantics of \( \text{ASP}\{f,cr\} \) programs is defined in two steps.
Definition 4 (Answer Sets of CR-Rule Free Programs). The answer sets of a cr-rule free ASP\{f,cr\} program are the answer sets of the corresponding \(f\) program.

Definition 5. Given an arbitrary ASP\{f,cr\} program \(\Pi\), a subset \(R\) of \(\Pi_{cr}\) is an abductive support of \(\Pi\) if \(\Pi^r \cup \alpha(R)\) is consistent and \(R\) is set-theoretically minimal among the sets satisfying this property.

Definition 6 (Answer Sets of Arbitrary Programs). For an arbitrary ASP\{f,cr\} program \(\Pi\), a set of literals \(A\) is an answer set of \(\Pi\) if it is an answer set of the program \(\Pi^r \cup \alpha(R)\) for some abductive support \(R\) of \(\Pi\).

Although out of the scope of the present paper, it is also possible to extend ASP\{f,cr\} to allow for the specification of CR-Prolog-style preferences over cr-rules.

4 Knowledge Representation with ASP\{f,cr\}

In this section we demonstrate the use of ASP\{f,cr\} for the formalization of certain types of knowledge. Whenever appropriate, we also compare with ASP and with other extensions of ASP by non-Herbrand functions.

Consider a scenario in which data from a town registry about births and deaths is used to determine who should receive a certain tax bill. The registry lists the year of birth and the year of death of a person. If a person is alive, no year of death is in the registry. The tax bill should only be sent to living people who are between 18 and 65 years old. To ensure robustness, we want to be able to deal with (infrequently) missing information. Hence, whenever there is uncertainty (represented by an atom \(\text{uncertain}(p)\)) about whether a person should receive the tax bill or not, a manual check will be performed. The first requirement can be encoded in ASP\{f,cr\} by the rule:

\[
\text{bill}(P) \leftarrow \\
\text{person}(P), \\
\text{age}(P) \geq 18, \text{age}(P) \leq 65, \\
\text{not uncertain}(P), \\
\text{not } \neg \text{bill}(P).
\]

Relation \(\text{person}\) defines a list of people known to the system. To shorten the rules, from now on we will implicitly assume the occurrence of an atom \(\text{person}(P)\) in every rule where \(P\) occurs. The condition \(\text{not } \neg \text{bill}(P)\) allows one to specify exceptions in the usual way. For example, the tax might be waived for low-income people:

\[
\neg \text{bill}(P) \leftarrow \text{low\_income}(P).
\]

Similarly, condition \(\text{not uncertain}(P)\) ensures that the bill is not sent if there is uncertainty about whether the person is subject to the tax.
Next, we define a person’s current age based on their year of birth. Following intuition, we define \textit{age} only for people who are alive.

\begin{align*}
\text{has\_birth\_year}(P) & \leftarrow \text{birth\_year}(P) = X. \\
\neg \text{has\_birth\_year}(P) & \leftarrow \not \text{has\_birth\_year}(P). \\
\text{has\_death\_year}(P) & \leftarrow \text{death\_year}(P) = X. \\
\neg \text{has\_death\_year}(P) & \leftarrow \not \text{has\_death\_year}(P). \\
\neg \text{alive}(P) & \leftarrow \text{has\_death\_year}(P). \\
\text{alive}(P) & \leftarrow \text{has\_birth\_year}(P), \neg \text{has\_death\_year}(P). \\
\text{age}(P) = X & \leftarrow \text{alive}(P), X = \text{current\_year} - \text{birth\_year}(P).
\end{align*}

Rules (5) and (7) determine when information about a person’s birth and death year is available. Rules (6) and (8) formalize the closed world assumption (CWA) of relations \textit{has\_birth\_year} and \textit{has\_death\_year}. Although this encoding of CWA is common practice in ASP, it plays an important role in the distinction between languages with partial functions and languages with total functions, as we discuss later. Rule (9) states that it is possible to conclude that a person is dead if a year of death is found in the registry. Rule (10) states that a person is alive if the registry contains a year of birth and does not contain information about the person’s death. Finally, rule (11) calculates a person’s age. \textit{current\_year} is a function of arity 0 whose value corresponds to the current year.

The next set of rules deals with the possibility of information missing from the registry. One important case to consider is that in which information about a person’s death is accidentally missing from the registry. In (10) a person is assumed to be alive unless evidence exists about the person’s death. This modeling choice is justified because missing information is assumed to be infrequent. Exceptional conditions can be dealt with by requesting a manual check on whether the person should receive the tax bill. Rule (12) below states that one such case is when a person’s age according to the registry is beyond that person’s maximum life span.

\begin{align*}
\text{uncertain}(P) & \leftarrow \text{alive}(P), \text{age}(P) \geq \text{max\_span}(P). \\
\text{max\_span}(P) = 92 & \leftarrow \not \text{max\_span}(P) \neq 92. \\
\text{max\_span}(P) = 100 & \leftarrow \text{long\_lived\_family}(P). \\
\text{uncertain}(P) & \leftarrow \not \text{alive}(P), \not \neg \text{alive}(P).
\end{align*}

Rule (13) provides a simple definition of a person’s maximum life span, stating that, normally, a person’s maximum life span is 92 years. Note that the rule is written in the form of a \textit{default over non-Herbrand functions}. This makes it possible for example to predict a different life span depending on a person’s medical or family history. Along
the lines of [5], the reading of (13) is “if $P$’s maximum life span may be 92, then it is 92.” Generally speaking, an expressions of the form $\text{not } f \neq g$ can be viewed to intuitively state “$f$ may be (equal to) $g$”. Rule (14) encodes one possible exception to the default, for people from families with a history of long life spans. Rule (15) covers instead the case in which the system couldn’t determine if a person is dead or alive. The formalization of this type of test has already been covered in the literature and is shown here for completeness, and using a a slightly simplified encoding. A discussion on this topic and proposals for more sophisticated formalizations can be found in [10,11].

It is currently a source of debate [14,7] whether support for partial functions should be allowed in languages with non-Herbrand functions. Although of course from the point of view of computational complexity partial and total functions in this context are equivalent, we believe that the following elaboration of the tax-bill scenario appears to show that the availability of partial functions is indeed important.

First of all, notice that, in a language that only supported total functions, the scenario discussed so far would have to be formalized by introducing a special constant. This special constant is to be used when the birth or death years are unknown. For the sake of this discussion, let us denote the special constant by $\text{<undef>}$.

Notice that, to allow for the use of $\text{<undef>}$, a design requirement would have to be imposed on the town registry so that entries that do not have a value are set to $\text{<undef>}$. It is worth mentioning that one might be tempted to avoid the use of $\text{<undef>}$ and instead reason by cases, one for each possible value of the year of birth or death. This approach however does not appear to work well in this scenario. In fact, in this scenario it is important to know whether the year of death is present in the registry at all – see e.g. rules (5) and (7). When reasoning by cases, it is not possible to reason about this circumstance from within the program, unless rather sophisticated extensions of ASP such as [10,11] are used.

Going back to our scenario, suppose that we want to incorporate in our system information from a database of deadly car accidents. Suppose the database consists of statements of the form $\text{died}(p, y)$, where $p$ is a person and $y$ is the year in which the deadly accident occurred. If there are inconsistencies between the town registry and the accident database, we would like to give precedence to the former. This can be easily formalized in $\text{ASP}\{f,cr\}$ with:

$$
\text{death\_year}(P) = Y \leftarrow \text{died}(P, Y), \text{not death\_year}(P) \neq Y.
$$

(16)

Informally, the rule states that, if $p$ is reported to have died in a car accident in year $y$, then that is assumed to be $p$’s death year, unless the town registry contains information to the contrary.

It is important to notice that our $\text{ASP}\{f,cr\}$ formalization makes it possible to incorporate the car accident database in a completely incremental fashion. No changes are needed to the rules we showed earlier. This is possible mainly because $\text{death\_year}$ is a partial function.

Let us see now how using a language with total functions would affect the incorporation of the car accident database. Let us consider a situation in which no death year is
specified for person \( p \) in the town registry, but an entry \( \text{died}(p, 1998) \) exists in the car accident database. As discussed above, in a language that only supported total functions, \( \text{death}_\text{year}(p) \) would have to be set to \(<\text{undef}>\) in the town registry. Hence, the body of rule (16) would never be satisfied.

To the best of our knowledge, working around this issue when using a language with total functions is non-trivial and the solutions are characterized by reduced elaboration tolerance. For example, one possible method consists in introducing a relation \( \text{determined}_\text{death}_\text{year}(P) \) that encodes the overall belief of the system about a person’s death year. By default, the value of \( \text{determined}_\text{death}_\text{year}(P) \) is obtained from the town registry. When that value is undefined, a death year can be derived from the car accident database by means of a rule similar to (to avoid using a specific language from the literature, we write the rule in the syntax of ASP\{f,cr\}):

\[
\begin{align*}
\text{determined}_\text{death}_\text{year}(P) &= Y 
\text{died}(P, Y), \\
\text{not } \text{determined}_\text{death}_\text{year}(P) &\neq Y.
\end{align*}
\]

Furthermore, any rule that previously involved relation \( \text{death}_\text{year} \) would have to be modified to use the new relation. This process, although seemingly harmless on the surface, tends to be error-prone and the corresponding encoding is hardly elaboration tolerant. Every time information from another database had to be incorporated, more changes to the existing program are required – for example, the reader may want to consider would happen if one had to incorporate information about births from e.g. a health insurance database.

Up to this point we have discussed cases in which the use of total functions appears to cause some issues. One may be wondering whether it is possible to represent total functions in ASP\{f,cr\}, and if there are any drawbacks.

To address this topic, let us suppose that we would like to determine the number of dependents of a person. This information could be used for example in order to ensure that certain individuals are waived from paying the tax discussed earlier. For simplicity of presentation, however, we discuss the determination of the number of dependents independently of the code shown earlier.

Let us assume that the number of a person’s dependents is found in their tax return, if one exists. If no tax return has been filed, we would like to consider separately each possible case, corresponding to a number of dependents ranging between 0 and \( \max_d \). The number of dependents can then be viewed as a total function.

In our formalization, an atom of the form \( \text{return}_\text{dep}(p, d) \) states that \( p \) has \( d \) dependents according to \( p \)'s latest tax return (or, equivalently, a function could be used instead of a relation). We will represent the number of a person \( p \)'s dependents by means of a function \( \text{dependents}(p) \).

A straightforward formalization, \( \Pi^1_t \), of such a total function is:

\[
\begin{align*}
\text{dependents}(P) &= D \leftarrow \text{return}_\text{dep}(P, D). \quad (17) \\
\text{dependents}(P) &= V \leftarrow \text{not } \text{dependents}(P) \neq V. \quad (18)
\end{align*}
\]
Rule (17) states that the number of dependents can be obtained from the tax return, if available. Rule (18) states that a person can have any number of dependents, unless there is reason to believe otherwise. Here and below we assume that variable $V$ ranges over the domain $[0, \text{max}_d]$ (this can be easily implemented by adding a condition $\text{dom}(V)$ and a suitable definition of relation $\text{dom}$).

$\Pi_1^i$ formalizes the nature of total function dependents for simple situations. However, suppose that one wanted to take into account the case in which $p$’s tax return was audited and the number of $p$’s dependents found to be different from what was stated in the tax return. Rule (17) does not properly deal with this case, because it prevents one from overriding a person’s dependents based on information from the audit. So, a different formalization of total functions is needed to deal with more sophisticated examples.

One might then be tempted to rewrite (17) as a default and to add a suitable exception, obtaining $\Pi_2^i$:

\begin{align*}
\text{dependents}(P) &= D \leftarrow \text{return}\_\text{deps}(P, D), \text{not dependents}(P) \neq D. \quad (19) \\
\text{dependents}(P) &= V \leftarrow \text{not dependents}(P) \neq V. \quad (20) \\
\text{dependents}(P) &= D \leftarrow \text{assessed}\_\text{deps}(P, D). \quad (21)
\end{align*}

Unfortunately this formalization does not yield the intended answers because of the interaction between defaults (19) and (20): consider a person $p_1$ with 3 dependents according to their tax return, $I_1 = \{\text{return}\_\text{deps}(P, 3)\}$. One might expect $I_1 \cup \Pi_2^i$ to yield the conclusion $\text{dependents}(p) = 3$, but in fact the program has multiple answer sets, enumerating all the possible numbers of dependents between 0 and $\text{max}_d$. This is an instance of a phenomenon already studied in the literature (see e.g. [13]), which can be circumvented by properly prioritizing the defaults of $\Pi_2^i$. Doing so however tends to affect the elaboration tolerance of the encoding (e.g. in case further defaults must be added) and is somewhat cumbersome and error-prone.

A more robust and elaboration tolerant approach relies on the use of cr-rules. Intuitively, in this approach, a cr-rule determines when to trigger the default behavior of considering all the possible values of a total function. Consider the following program, $\Pi_3^i$:

\begin{align*}
\text{dependents}(P) &= D \leftarrow \text{return}\_\text{deps}(P, D), \text{not dependents}(P) \neq D. \quad (22) \\
\text{dependents}(P) &= D \leftarrow \text{assessed}\_\text{deps}(P, D). \quad (23) \\
\text{has}\_\text{dep}\_\text{info}(P) &\leftarrow \text{dependents}(P) = D. \quad (24) \\
&\leftarrow \text{not has}\_\text{dep}\_\text{info}(P). \quad (25) \\
\text{dependents}(P) &= D \overset{\dagger}{\leftarrow}. \quad (26)
\end{align*}

Program $\Pi_3^i$ is obtained from $\Pi_2^i$ by replacing rule (20) by (24-26). Rules (24-25) intuitively state that the number of dependents must be known for every person. Cr-rule (26) intuitively states that it is possible to assume that a person has any number of dependents, but that this possibility should be used only if strictly necessary and in order to restore consistency.
It is not difficult to see that \( \Pi_{i3} \) yields the expected conclusions. To this extent, it is important to notice that cr-rule (26) will only be used for people for whom no other dependent information is available. In fact, let \( I_3 \) be a set of facts providing partial information about the dependents of a group of people, and \( U_3 = \{p_1, \ldots, p_u\} \) be the set of people from \( I_3 \) for whom no dependent information is available. According to Definition 5, any abductive support of \( \Pi_{i3} \cup I_3 \) must contain, for every \( p_i \in U_3 \), a ground cr-rule \( \text{dependents}(p_i) = d \leftarrow \) for some \( d \). Let now \( R_3 \) be the set of all such cr-rules, and consider a person \( p' \) for whom dependent information is provided in \( I_3 \). The corresponding set \( R'_3 = R_3 \cup \text{dependents}(p') = d' \leftarrow \) is not an abductive support of \( \Pi_{i3} \cup I_3 \) because it is not set-theoretically minimal. Hence, cr-rule (26) is only used for the people in \( U_3 \).

It is not difficult to see that this approach for the encoding of total functions in ASP\{f,cr\} is applicable in general, and that (24) can be rewritten as a general, domain-independent axiom (an example of a domain-independent axiom can be found in the next section).

5 Some Modeling and Solving Tasks in ASP\{f,cr\}

In this section we demonstrate the use of ASP\{f,cr\} for a sample of modeling and solving tasks from the literature. We also include a (partial) discussion of the features of our encodings in relation with other methods for representing non-Herbrand functions.

We refer the reader to the description and original encodings from http://www.cs.uni-potsdam.de/∼torsten/kr12tutorial.

Water Buckets on a Scale (page 216). In this scenario, one bucket is placed on each arm of a two-armed scale. Each bucket initially contains an amount of water between 0 and 100. All amounts of water in this scenario are represented by integer values. At every time step, an agent must pour an amount \( k, 1 \leq k \leq \max \) of water into one of the buckets.\(^6\) The agent’s goal is to balance the two buckets on the scale. The ASP\{f,cr\} encoding, \( \Pi^w \), of this scenario is:

\[
\text{bucket}(a), \text{bucket}(b). \\
1\{\text{pour}(B, T, K) : \text{bucket}(B) : K \geq 1 : K \leq \max_w\}1 \leftarrow \text{time}(T), T < t. \quad \text{(27)} \\
\text{poured}(B, T) = K \leftarrow \text{pour}(B, T, K). \quad \text{(28)} \\
\text{volume}(B, T + 1) = V \leftarrow V = \text{volume}(V, T) + \text{poured}(B, T). \quad \text{(29)} \\
\text{volume}(B, T + 1) = \text{volume}(B, T) \leftarrow \text{not volume}(B, T + 1) \neq \text{volume}(B, T). \quad \text{(30)} \\
\text{heavier}(B, T) \leftarrow \text{bucket}(B), \text{bucket}(C), \text{time}(T), \text{volume}(C, T) < \text{volume}(B, T). \quad \text{(31)} \\
\text{bucket}(B), \text{heavier}(B, t). \quad \text{(32)}
\]

Rule (27) states that the agent can pour any allowed amount of water in any one bucket at every time step. For compactness, the rule uses the syntax of choice rules from

\(^6\) We deviate slightly from the original scenario in that the agent is allowed to decide how much water is to be poured.
LPARSE and GRINGO. Extending the definition of ASP\{f,cr\} to support choice rules is trivial. Rule (28) states that the amount of water poured as a consequence of action \texttt{pour}(b, t, k) is k. Rule (29) encodes a dynamic law: it states that when water is poured into a bucket, the volume of water in the bucket increases by the amount of water poured. We assume that a suitable domain has been specified for variable V. Rule (30) formalizes the inertia axiom. It states that the volume of water in a bucket stays the same unless it is forced to change. Rule (31) describes the conditions under which a bucket is heavier than the other. Finally, rule (32) states that it is impossible for a bucket to be heavier than the other at the end of the execution of the plan.

It is worth observing that, as prescribed by good knowledge representation principles, in Π\textsuperscript{w} the inertia axiom is written without references to the occurrence of any action. This allows for a rather elaboration tolerant encoding. In the original CLINGCON encoding, on the other hand, the inertia axiom mentions the occurrence of actions:

\begin{align*}
amount(B, T) & \equiv 0 \leftarrow \text{not } \text{pour}(B, T), \text{bucket}(B), T < t. \\
volume(B, T + 1) & \equiv volume(B, T) + amount(B, T) \leftarrow \text{bucket}(B), T < t.
\end{align*}

This encoding is arguably less elaboration tolerant than Π\textsuperscript{w}; for example, the CLINGCON inertia axiom (33) must be modified whenever new actions are introduced in the representation, while the inertia axiom from Π\textsuperscript{w} does not have to be changed, and the whole program can be extended in a completely incremental fashion. In fact, ASP\{f,cr\} makes it possible to encode the inertia axiom in a form that is even more general than that of rule (30):

\begin{align*}
\text{num fluent}(volume(B)) & \leftarrow \text{bucket}(B). \\
\text{val}(N, T + 1) = \text{val}(N, T) & \leftarrow \text{num fluent}(N), \text{not } \text{val}(N, T + 1) \neq \text{val}(N, T).
\end{align*}

Rule (35) states that \texttt{volume}() is a “numerical fluent”. Rule (36) states that the value of any numerical fluent remains the same over time unless it is forced to change. The advantage of this generalized form of the inertia axiom is that the corresponding rules apply without changes to any numerical fluent, so that now the addition of new numerical fluents to the encoding can be fully incremental as well.

From the point of view of the size of the grounding, the CLINGCON encoding is however superior to Π\textsuperscript{w}, because in Π\textsuperscript{w} rule (29) must be grounded for every possible value of variable V, while in the CLINGCON encoding the grounding is entirely independent of the volume of water in the buckets. On the other hand, the size of the grounding of Π\textsuperscript{w} is substantially better than the best ASP encodings that we are aware of. In the ASP encodings, in fact, the grounding of the inertia axiom grows proportionally to the square of the domain of variable V. A similar phenomenon can be observed in the encodings based on the languages of [7,14], since in those approaches computation of the answer sets is performed by translating the programs to ASP.

\footnote{To complete the encoding (29) and (31) have to be modified in a straightforward way to use \texttt{val}() as well. From a technical perspective, in this encoding ground expressions \texttt{volume}(a) and \texttt{volume}(b) are viewed as constants. This is possible because no assumptions are made about the set of constants in our definition of the language. Alternatively, one could of course extend the language with Herbrand function symbols and of Herbrand terms, at the cost of a slightly more complex presentation.}
N-Queens (page 176). In this scenario an agent must place \( n \) queens on an \( n \times n \) chessboard so that no queen can attack another. In this scenario the size of the grounding and the execution time tend to grow quickly with the increase of parameter \( n \). In straightforward ASP encodings, the growth of the grounding is due to the tests ensuring that no queen can attack another. \( \Pi^3 \) shows one possible ASP encoding:

\[
\begin{align*}
&\leftarrow \text{queen}(X_1, Y_1), \text{queen}(X_1, Y_2), Y_1 < Y_2, \\
&\leftarrow \text{queen}(X_1, Y_1), \text{queen}(X_2, Y_1), X_1 < X_2, \\
&\leftarrow \text{queen}(X_1, Y_1), \text{queen}(X_2, Y_2), X_1 < X_2, X_2 - X_1 = |Y_2 - Y_1|.
\end{align*}
\]

Conditions \( Y_1 < Y_2 \) and \( X_1 < X_2 \) are introduced in order to break symmetries. The last rule is the most problematic with respect to the size of the grounding, because its grounding grows roughly with \( O(n^4) \). Several modifications of \( \Pi^3 \) are known, which decrease the size of the grounding.\(^8\) However, it is often argued that these modifications make the corresponding encodings either less declarative, or less elaboration tolerant. Certainly, most of the modifications achieve performance by a less straightforward encoding of the constraints of the problem.

It is then interesting to compare \( \Pi^3 \) with a straightforward ASP\{f,cr\} encoding. \( \Pi^2_2 \):

\[
\begin{align*}
&\leftarrow Q_1 < Q_2, \text{col}(Q_1) = \text{col}(Q_2). \\
&\leftarrow Q_1 < Q_2, \text{row}(Q_1) = \text{row}(Q_2). \\
&\leftarrow Q_1 < Q_2, \text{col}(Q_2) - \text{col}(Q_1) = |\text{row}(Q_2) - \text{row}(Q_1)|.
\end{align*}
\]

Condition \( Q_1 < Q_2 \) performs basic symmetry breaking. \( \Pi^2_2 \) uses two functions to encode the positions of the queens. What is remarkable about \( \Pi^2_2 \) is that the grounding of the last rule grows roughly with \( O(n^2) \), although we argue that it is as straightforward an encoding of the requirement as the corresponding rule from \( \Pi^3 \). As in the previous scenario, we expect a similar growth for comparable CLINGCON encodings, and a growth of \( O(n^4) \) for the grounding of the encodings written in the languages of [7,14].

### 6 Conclusions and Future Work

In this paper we have defined the syntax and semantics of an extension of ASP by non-Herbrand functions with full-fledged arithmetic expressions and consistency-restoring rules. The resulting language ASP\{f,cr\} supports partial functions and we hope we have demonstrated that it allows for the encoding of rather sophisticated kinds of knowledge, including knowledge about total functions. Compared to similar languages, ASP\{f,cr\} strikes a remarkable balance between expressive power and efficiency of computation. In the previous section, the discussion on the efficiency of computation was based only on the size of the grounding of the corresponding encodings, but in [1] experimental evidence on solver performance was obtained using a prototype of an ASP\{f\} solver (available at http://marcy.cjb.net/clingof). We expect that a version of the solver including support for the extended language defined in this paper will be available soon. Once that becomes available, we plan to substantiate the discussion from the previous section with experimental results.

\(^8\) See especially http://www.cs.uni-potsdam.de/~torsten/kr12tutorial.
References

Utilizing ASP for Generating and Visualizing Argumentation Frameworks

Günther Charwat, Johannes Peter Wallner, and Stefan Woltran

Vienna University of Technology, Institute of Information Systems 184/2, Favoritenstraße 9-11, 1040 Vienna, Austria
{gcharwat,wallner,woltran}@dbai.tuwien.ac.at

Abstract. Within the area of computational models of argumentation, the instantiation-based approach is gaining more and more attention, not at least because meaningful input for Dung’s abstract frameworks is provided in that way. In a nutshell, the aim of instantiation-based argumentation is to form, from a given knowledge base, a set of arguments and to identify the conflicts between them. The resulting network is then evaluated by means of extension-based semantics on an abstract level, i.e. on the resulting graph. While several systems are nowadays available for the latter step, the automation of the instantiation process itself has received less attention. In this work, we provide a novel approach to construct and visualize an argumentation framework from a given knowledge base. The system we propose relies on Answer-Set Programming and follows a two-step approach. A first program yields the logic-based arguments as its answer-sets; a second program is then used to specify the relations between arguments based on the answer-sets of the first program. As it turns out, this approach not only allows for a flexible and extensible tool for instantiation-based argumentation, but also provides a new method for answer-set visualization in general.

1 Introduction

Instantiation-based argumentation [7] is a central paradigm in nonmonotonic reasoning since it gives a formal handle to separate the logical and non-classical contents of reasoning in the presence of contradicting information. Hereby, one starts with a knowledge base and constructs arguments from it. Arguments typically consist of two parts, namely a support, which is grounded in the knowledge base and a claim derived from it. In [4] the process is described with an underlying propositional knowledge base using minimal sets of consistent support classically entailing the claim. In a second step, conflicts between these arguments have to be identified. The obtained arguments and the relation between them yield a so-called argumentation framework [9]. This simple, yet expressive formalism is basically a directed graph whereby the arguments are represented via vertices and the conflicts with directed edges. Argumentation frameworks are then evaluated with one of the numerous semantics for abstract argumentation available, resulting in potentially multiple acceptable sets of arguments [3].

Here we are only interested in the instantiation part, however, which received less attention wrt. realized systems. Notable exceptions are the Carneades system, which can construct arguments using heuristics [16] and the recent TOAST implementation
for the ASPIC+ framework [27]. The reason for the lack of implementations is potentially twofold: First, due to the inherent high complexity of the problem; already constructing a single argument is hard for the second level of the polynomial hierarchy [23]. Secondly, standard instantiation schemes for propositional knowledge bases result in infinite argumentation frameworks even for finite knowledge bases [1]. The first obstacle calls for highly expressive languages, making answer-set programming [6, 21, 22] (ASP, for short) a well suited candidate. For the second obstacle, we restrict ourselves here to arguments that have their claims coming from an a priori specified set of formulae.

To summarize, we aim here for a system which takes as input a knowledge base as well as a set of potential claims and produces the instantiated argumentation framework, such that the latter can be processed by other argumentation tools, e.g. ASPARTIX [11] or CEGARTIX [10]. More specifically, our contributions are as follows:

- We provide ASP programs\(^1\) to encode the construction of arguments as well as the construction of the conflicts. For the second task, the answer-sets of the first encoding are used as input. Thus we can make use of the high sophistication modern ASP systems have reached [15, 20]. Moreover, since the argument construction and conflict identification are declaratively described via ASP code, the system is easily adaptable to other notions of arguments or conflicts.

- We present a system that, on the one hand, takes care of passing the answer-sets from one program to another. On the other hand, the system uses the answer-sets of the two programs for visualization in form of a graph. In our case we obtain an argumentation framework. Finally, this result can be exported and used by other systems for abstract argumentation.

As a by-product, we observed that this method is by no means restricted to the argumentation domain. Ultimately, it allows for a user-driven graph representation of the collection of answer-sets of a given input program, thus acting as a tool for ASP visualization in general. The most interesting feature of the tool is that the concrete specification for two answer-sets being in relation is given by an ASP program itself. In recent years, ASP has benefited from the rising number of development and visualization tools, e.g. ASPViz [8], ASPIDE [14], Kara [19] and IDPDraw [29]. These tools so far have focused on presentations of single answer-sets of the given program. However, in certain applications it is not only the single answer-sets which are of interest, but the relation between them.

While visualization is a rather new research branch in ASP, it has gained more attention in the argumentation community, where dedicated visualization tools have been proposed already in the late 90s (e.g. [2, 5, 11, 18, 24–26, 28], including Debategraph\(^2\) and Rationale\(^3\)). Many of these support the argument construction by a user via different means, such as automated reasoning, input masks and database querying. Compared to these systems, our approach combines the computational power of high-sophisticated ASP systems with visualization aspects. Moreover, thanks to the declarative nature of

---

\(^1\)http://dbai.tuwien.ac.at/proj/argumentation/vispartix/

\(^2\)http://debategraph.org

\(^3\)http://rationale.austhink.com/
the ASP encodings specifying the instantiation step, we believe that the strength of our approach lies in its flexibility and its expandability to new argumentation formalisms.

This paper is structured as follows: We briefly introduce argumentation and ASP in Section 2. Then, in Section 3, we present ASP encodings for constructing an argumentation framework. In Section 4 we outline a novel visualization tool which is used for representing relations between answer-sets. A final discussion is given in Section 5.

2 Preliminaries

We provide the necessary background from argumentation theory and ASP for this work. In particular we will explain the argumentation process based on argumentation frameworks [9] as well as briefly recall the concepts for disjunctive logic programs.

2.1 Argumentation

In this section we introduce formal argumentation. We start with the underlying process [7], which we will utilize in our context. The general process consists of three steps. First, given a knowledge base, arguments and their relationships are instantiated. After this instantiation the arguments are treated as abstract entities, without considering their concrete content. Secondly conflicts are resolved using appropriate semantics on the abstract instantiation and finally conclusions are drawn.

In this work the knowledge base $K$ is a (potentially inconsistent) set of propositional logic formulae. We construct the formulae with the usual connectives $\neg, \lor, \land, \rightarrow$, the negation, disjunction, conjunction and implication, respectively. Furthermore entailment and logical equivalence of formulae is denoted by $|$ and $\equiv$, respectively. We write formulae with lowercase Greek letters $\alpha, \beta, \gamma, \ldots$.

Example 1. Consider the following simple and inconsistent example knowledge base:

$K = \{a, a \rightarrow b, \neg b\}$

The instantiation step now constructs arguments and relations among them based on the information available in $K$ according to [4]. The abstract representation we utilize for this purpose is the widely studied argumentation framework [9]. An argumentation framework (AF) is a directed graph $F = (\text{Args}, \text{Att})$, with the vertices (\text{Args}) being abstract arguments and the directed edges (\text{Att}) denote attacks between them to represent conflicts.

The instantiation of an AF now consists of two parts, namely the argument construction and the attack relation construction. An argument $A = (S, C)$ consists of a support for the argument and a claim. The support is a subset of the knowledge base $K$ and the claim is a single logical formula. The support must be a consistent and subset minimal set of formulae, which entails the claim.

Here the arguments are pairs of support and claim to provide a formal basis for argument construction. When plugged in the argumentation framework we abstract from this “inner” structure and collapse every pair of support and claim into one abstract argument. This is the abstraction procedure of the overall process.
We note that argument construction here differs from the usual argument definition in the literature. In particular the claim can be taken only from a pre-defined set $C$. Using a pre-defined set of claims, we can restrict ourselves to reasonable claims, e.g. not involving tautologies. In this way we prohibit the construction of infinitely many arguments that could otherwise result from infinitely many syntactically different formulae which are semantically equivalent. This restriction comes with a disadvantage however, as the set of pre-defined claims must be chosen with care, since inconsistent conclusions might be drawn otherwise. Indeed, [17] identify conditions for rational and consistent end results, which require the existence of specific arguments, which must be included in $C$. On the other hand, this restriction is in line with the concept of cores of argumentation frameworks [1], which try to preserve desired properties while using only a subset of all possible arguments.

Example 2. Continuing Example 1, let the set of claims be $C = K \cup \{ \neg a, b, a \land \neg b \}$. Then we can construct the following arguments:

$$
\begin{align*}
a_1 &= (\{a\}, a) \\
a_2 &= (\{a \rightarrow b\}, a \rightarrow b) \\
a_4 &= (\{\neg b, a \rightarrow b\}, \neg a) \\
a_5 &= (\{\neg b\}, \neg b) \\
a_6 &= (\{a, \neg b\}, a \land \neg b)
\end{align*}
$$

For the construction of the attack relation several options were studied in literature. The basic idea for attacks between arguments underlying all of these options is that some sort of inconsistency occurs between them. We take the attack definitions from [17] and illustrate two types, defeat and directed defeat. An argument $A = (S, C)$ attacks an argument $A' = (S' = \{\phi_1', ..., \phi_m'\}, C')$ using defeat if $C \models \neg (\phi_1' \land ... \land \phi_m')$. The former directly defeats the latter if $C \models \neg \phi_i'$ for one $i$, $1 \leq i \leq m$.

Example 3. Continuing Example 2, the AF in Figure 1 illustrates the result using the direct defeat on the arguments built from $K$ and the claims $C$. Note that e.g. $a_5$ and $a_6$ are not mutually attacking each other, since the claim of $a_5$ does not entail a negated support formula of $a_3$.

This completes the first step of the argumentation process, namely the AF construction out of the knowledge base. For the conflict resolution a plethora of argumentation framework semantics exist. A basic property for semantics is the conflict-free property, which states that a set $M$ of arguments in an AF $F$ is conflict free if there are no attacks between them in $F$. A set of arguments $M$ is stable in an AF
Example 4. If we take the argumentation framework from Example 3, then the stable semantics selects \( \{\{a_1, a_5, a_6\}, \{a_1, a_2, a_3\}, \{a_2, a_4, a_5\}\} \) as acceptable subsets of arguments.

The last step of the argumentation process deals with drawing conclusions from the sets of acceptable arguments. One can look at the content of the abstract arguments which were accepted, e.g. one can derive the deductive closure of this content.

In general every step of this process is intractable. Hence we need sophisticated systems for tackling these steps, which makes ASP a suitable choice for embedding the process in. A more detailed computational complexity analysis can be found in [23].

### 2.2 Answer-Set Programming

In this section we recall the basics of disjunctive logic programs under the answer-sets semantics [6, 22].

We fix a countable set \( \mathcal{U} \) of (domain) elements, also called constants. 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 \( r \) is of the form

\[
a_1 \lor \ldots \lor a_n \leftarrow b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m
\]

with \( n \geq 0, m \geq k \geq 0, n + m > 0 \), where \( a_1, \ldots, a_n, b_1, \ldots, b_m \) are atoms, and “not” stands for default negation. The head of \( r \) is the set \( \text{H}(r) = \{a_1, \ldots, a_n\} \) and the body of \( r \) is \( \text{B}(r) = \{b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m\} \). Furthermore, \( B^+(r) = \{b_1, \ldots, b_k\} \) and \( B^-(r) = \{b_{k+1}, \ldots, b_m\} \). A rule \( r \) is 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 empty body. A program is a finite set of disjunctive rules.

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

### 3 Instantiation-based Argumentation

In this section we provide our ASP encodings for the construction of arguments from a knowledge-base \( K \) and a set \( C \) of claims. As input, each formula in \( K \) and \( C \) is given by the unary predicate \( kb(\cdot) \) and \( cl(\cdot) \), respectively.
Example 5. The input, as given in Example 1 and 2, is specified by:

\[
\{ \text{kb}(a), \text{kb}(\text{imp}(a, b)), \text{kb}(\neg(b)), \\
\text{cl}(a), \text{cl}(\text{imp}(a, b)), \text{cl}(\neg(b)), \text{cl}(\neg(a)), \text{cl}(b), \text{cl}((a, \neg(b))) \} 
\]

First, we introduce the ASP encodings for checking whether a certain variable assignment is a model for a given formula (or not). Model checking plays a crucial role for our instantiation-based approach. Then, we present encodings for the computation of arguments. Finally, we provide ASP code for some types of attack relations. Note that an argumentation framework is obtained by two separate ASP program calls where the first one takes as input \( K \) and \( C \) and returns a separate answer-set for each resulting argument. The second program receives as input a “flattened” version of all arguments and computes the attacks between arguments based on different attack type encodings.

3.1 Model Checking

Propositional formulae provide the basis for the construction of arguments and their attack relations. In fact, we can express most of the defining properties of arguments (such as entailment of the support to the claim) and attacks by means of propositional formulae. In this section we provide an ASP encoding that allows us to check whether a formula \( \alpha \) is true under a given interpretation \( I \), i.e. \( I \) is a model for \( \alpha \). First, the formula is split into sub-formulae until we obtain the contained atoms or constants. Due to brevity, the following encodings only exemplify this for the connectives \( \land, \neg \) and \( \rightarrow \). Note that \( \lor, \not\leftrightarrow \) and \( \leftrightarrow \) are supported as well.

\[
\pi_{\text{subformula}} = \{ \text{subformula}(F) \leftarrow \text{subformula}(\text{and}(F, _)); \  \ 	ext{(1)} \\
\text{subformula}(F) \leftarrow \text{subformula}(\text{and}(, F)); \  \ 	ext{(2)} \\
\text{subformula}(F) \leftarrow \text{subformula}(\text{neg}(F)); \ \ 	ext{(3)} \\
\text{subformula}(F) \leftarrow \text{subformula}(\text{imp}(F, _)); \ \ 	ext{(4)} \\
\text{subformula}(F) \leftarrow \text{subformula}(\text{imp}(, F)) \} \ 	ext{(5)}
\]

The atoms and constants of \( \alpha \) are then obtained via the encoding \( \pi_{\text{atom}} \). Consider rule (1) which denotes that a formula is not an atom in case it is of the form \( \text{and}(\cdot, \cdot) \).

\[
\pi_{\text{atom}} = \{ \text{noatom}(F) \leftarrow \text{subformula}(F; F1; F2), F := \text{and}(F1, F2); \ \ 	ext{(1)} \\
\text{noatom}(F) \leftarrow \text{subformula}(F; F1), F := \text{neg}(F1); \ \ 	ext{(2)} \\
\text{noatom}(F) \leftarrow \text{subformula}(F; F1; F2), F := \text{imp}(F1, F2); \ \ 	ext{(3)} \\
\text{atom}(X) \leftarrow \text{subformula}(X), \text{not atom}(X). \ \ 	ext{(4)}
\]

Now we compute whether the interpretation is a model by first evaluating the atoms and constants. In case an atom gets assigned true (false) we derive that the interpretation for this sub-formula is a model (not a model). Now, the connectives are evaluated bottom-up based on the model information of the sub-formulae. In particular, this allows to check whether \( I \) is a model for our original formula \( \alpha \), or not.

The encoding \( \pi_{\text{ismodel}} \) exemplifies this approach for some of the connectives. In the subsequent sections we have to apply model checking several times within a single ASP encoding. In order to avoid side effects of different checks, we introduce an additional

\footnote{Note that the syntax of our encodings is specific to the grounder gringo [15].}
parameter, \( K \), which serves as a key for identifying the origin of the interpretation that is currently checked. Suppose, for example, that we want to check satisfiability of two different formulae. As the formulae may evaluate to true under different interpretations we have to distinguish between the truth assignments.

\[
\pi_{\text{ismodel}} = \{ \text{ismodel}(K, X) \leftarrow \text{atom}(X), \text{true}(K, X) \}; \hspace{1cm} (1) \\
\text{ismodel}(K, F) \leftarrow \text{subformula}(F; F_1), F := \text{neg}(F_1), \hspace{1cm} (2) \\
\text{nomodel}(K, F_1); \hspace{1cm} (3) \\
\text{ismodel}(K, F) \leftarrow \text{subformula}(F), F := \text{and}(F_1, F_2), \\
\text{ismodel}(K, F_1; F_2); \hspace{1cm} (4) \\
\text{nomodel}(K, F) \leftarrow \text{subformula}(F), F := \text{imp}(F_1, F_2), \\
\text{ismodel}(K, F_1; F_2). \} 
\]

Due to brevity we omit the encoding \( \pi_{\text{nomodel}} \) here. Analogous to \( \pi_{\text{ismodel}} \) it derives the predicate \( \text{nomodel}(K, F) \) whenever an atom gets assigned false or a sub-formula is false under the current interpretation. The complete program for checking whether a formula evaluates to true under a given variable assignment consists of

\[
\pi_{\text{modelcheck}} = \pi_{\text{subformula}} \cup \pi_{\text{atom}} \cup \pi_{\text{ismodel}} \cup \pi_{\text{nomodel}} 
\]

**Example 6.** Consider the formula \( a \rightarrow b \) from \( \mathcal{K} \) of Example 5, i.e. \( \text{kb}(\text{imp}(a, b)) \). In order to check whether there exists a model we can make use of \( \pi_{\text{modelcheck}} \) in the following way: Initially, we have to define an additional rule \( \text{subformula}(X) \leftarrow \text{kb}(X) \) as \( \pi_{\text{subformula}} \) only considers formulae given by the predicate \( \text{subformula}(-) \). By adding the program \( \pi_{\text{subformula}} \cup \pi_{\text{atom}} \) the following answer-set is returned:

\[
\{ \text{kb}(\text{imp}(a, b)), \text{subformula}(\text{imp}(a, b)), \text{subformula}(a), \text{subformula}(b), \text{nomodel}(\text{imp}(a, b)), \text{atom}(b), \text{atom}(a) \} 
\]

Each atom now gets assigned true or false, representing an interpretation. We encode this by the rule \( \text{true}(k, X) \lor \text{false}(k, X) \leftarrow \text{atom}(X) \). Note that the specification of a key (in this case \( k \)) is mandatory although \( \pi_{\text{modelcheck}} \) is not applied several times in this example. By adding and running \( \pi_{\text{ismodel}} \cup \pi_{\text{nomodel}} \) four answer-sets are returned. Each contains the predicates from the previously given answer-set as well as the truth assignment for the atoms \( a \) and \( b \) and either \( \text{ismodel}(k, \text{imp}(a, b)) \) or \( \text{nomodel}(k, \text{imp}(a, b)) \). The answer-set obtained by \( \text{false}(k, a) \) and \( \text{true}(k, b) \) contains (amongst others)

\[
\{ \text{false}(k, a), \text{true}(k, b), \text{ismodel}(k, b), \text{nomodel}(k, a), \text{ismodel}(k, \text{imp}(a, b)) \} 
\]

denoting that \( I(a) = \text{false}, I(b) = \text{true} \) is a model for \( a \rightarrow b \).

### 3.2 Forming Arguments

We now derive the arguments from a knowledge base \( \mathcal{K} \) and a set \( \mathcal{C} \) of claims. According to [4], we have to check whether the support entails the claim and if the support is subset minimal as well as consistent. In order to obtain arguments we first guess exactly one claim and a subset of formulae from \( \mathcal{K} \). This guess is encoded as follows:

\[
\pi_{\text{arg}} = \{ 1 \{ \text{sclaim}(X) : \text{cl}(X) \}; \hspace{1cm} (1) \\
1 \{ \text{fs}(X) : \text{kb}(X) \} \} \hspace{1cm} (2) 
\]
The selected claim is denoted by \( sclaim(\cdot) \). The predicate \( fs(\cdot) \) is derived if the respective formula from \( K \) is contained in the support \( S \) of an argument \( A = (S,C) \).

**Entailment:** In order to be a valid argument, the support must entail the claim, i.e. \( S \models C \) must hold. As \( S \models C, \models S \rightarrow C \) must hold as well. Hence, \( \neg(S \rightarrow C) \equiv \neg(\neg S \lor C) \equiv S \land \neg C \) must be unsatisfiable. Unsatisfiability of the formula \( S \land \neg C \) can be checked by making use of the saturation technique [12]: We first assign \( \text{true}( entail, x ) \) or \( \text{false}( entail, x ) \) to each atom \( x \) in the formula using a disjunctive rule. This allows both \( \text{true}( entail, x ) \) and \( \text{false}( entail, x ) \) to be contained in the resulting answer-set. Furthermore, all formulae in \( S \) and the negated claim \( C \) are conjunctively connected. Hence, in case any of those formulae evaluates to false under a variable assignment (i.e. \( \text{nomodel}( entail, \cdot ) \) is derived) we know that \( \neg( S \rightarrow C ) \) is not satisfied which implies that \( S \models C \) evaluates to true under the given interpretation. In this case we saturate, i.e. we derive \( \text{true}( entail, x ) \) and \( \text{false}( entail, x ) \) for any atom \( x \). On the other hand, if no formula in \( S \) and \( C \) derives \( \text{entails}_\text{claim} \) the constraint \( \leftarrow \text{not entails}_\text{claim} \) removes the answer-set. If this is the case, due to the definition of stable model semantics in answer-set programming, no answer-set is returned. Only in case there exists no model for \( \neg( S \rightarrow C ) \) all guesses are saturated and we obtain a single answer-set representing a support \( S \) and claim \( C \) where \( S \models C \) holds.

In the following the program \( \pi_{\text{entailment}} \) is given. Note that \( entail \) is simply used as a key for identifying the variable assignment and model check.

\[
\pi_{\text{entailment}} = \{ \left\{ \begin{align}
\text{true}( entail, X ) & \lor \text{false}( entail, X ) \leftarrow \text{atom}(X); \quad (1) \\
\text{entails}_\text{claim} & \leftarrow \text{nomodel}( entail, \neg\text{atom}(X)), sclaim(X); \quad (2) \\
\text{entails}_\text{claim} & \leftarrow \text{nomodel}( entail, X ), fs(X); \quad (3) \\
\leftarrow \text{not entails}_\text{claim}; \quad (4) \\
\text{true}( entail, X ) & \leftarrow \text{entails}_\text{claim}, \text{atom}(X); \quad (5) \\
\text{false}( entail, X ) & \leftarrow \text{entails}_\text{claim}, \text{atom}(X). \quad (6)
\end{align} \right. \}
\]

**Subset minimality:** The support \( S \) of an argument must be a subset minimal set of formulae, i.e. there must not exist an \( S' \subset S \) s.t. \( S' \models C \). Here, we apply the concept of a loop (see e.g. [13]). For a candidate support \( S \) we consider all \( S' \subset S \) where there exists exactly one formula \( \alpha \in S \) but \( \alpha \notin S' \). In case any such \( S' \) exists where \( S' \models C \) we know that \( S \) is not a support for \( C \). Due to monotonicity of classical logic this is sufficient since if \( S' \models C \) then also for all \( S'' \subset S' \) it holds that \( S'' \models C \). First, we define a total ordering over all formulae \( fs(\cdot) \) in \( S \):

\[
\pi_c = \{ \left\{ \begin{align}
\text{lt}(X,Y) & \leftarrow \text{fs}(X), fs(Y), X < Y; \quad (1) \\
\text{nsucc}(X,Z) & \leftarrow \text{lt}(X, Y), \text{lt}(Y, Z); \quad (2) \\
\text{succ}(X,Y) & \leftarrow \text{lt}(X, Y), \not\text{nsucc}(X,Y); \quad (3) \\
\text{nin}(Y) & \leftarrow \text{lt}(X, Y); \quad (4) \\
\text{inf}(X) & \leftarrow \text{fs}(X), \not\text{nin}(X); \quad (5) \\
\text{nsup}(X) & \leftarrow \text{lt}(X, Y); \quad (6) \\
\text{sup}(X) & \leftarrow \text{fs}(X), \not\text{nsup}(X). \quad (7)
\end{align} \right. \}
\]

For any \( S' \) we now assign \( \text{true}( m(K), x ) \) or \( \text{false}( m(K), x ) \) to all atoms \( x \). \( m(K) \) is used as key for identifying the truth assignment. \( K \) is the formula \( \alpha \) where \( \alpha \notin S' \).
The idea is now to “iterate” over the ordering, beginning at the infimum \( \inf(\cdot) \). Based on the ordering, we now consider every formula from the support: In case the formula is satisfied or corresponds to the removed formula \( \alpha \) (i.e. the key \( K \)) we derive \( \text{model}_\text{upto}(m(\alpha), \cdot) \). If we can derive \( \text{hasmodel}(m(\alpha)) \) we know that the support \( S' = S \setminus \alpha \) is satisfiable and can therefore not be a valid support for our claim. On the other hand, if any \( S' \) is a valid support we can not derive \( \text{hasmodel}(m(\alpha)) \) and the answer-set is removed by the constraint \( \leftarrow \text{not hasmodel}(m(\alpha)), \text{fs}(\alpha) \).

\[
\pi_{\text{minimize}} = \{ \text{true}(m(K), X) \leftarrow \text{not false}(m(K), X), \text{atom}(X), \text{fs}(K); \}
\text{false}(m(K), X) \leftarrow \text{not true}(m(K), X), \text{atom}(X), \text{fs}(K); \}
\text{model}_\text{upto}(m(K), X) \leftarrow \inf(X), \text{ismodel}(m(K), X), \text{fs}(X), X \neq K; \}
\text{model}_\text{upto}(m(K), K) \leftarrow \inf(K), \text{fs}(K); \}
\text{model}_\text{upto}(m(K), X) \leftarrow \text{succ}(Z, X), \text{ismodel}(m(K), X), \text{fs}(X), \text{model}_\text{upto}(m(K), Z), X \neq K; \}
\text{model}_\text{upto}(m(K), K) \leftarrow \text{succ}(Z, K), \text{model}_\text{upto}(m(K), Z), \text{fs}(K); \}
\text{hasmodel}(m(K)) \leftarrow \text{sup}(K), \text{model}_\text{upto}(m(K), X), \text{ismodel}(m(K), \neg(Z)), \text{claim}(Z); \}
\leftarrow \text{not hasmodel}(m(K)), \text{fs}(K). \}
\]

**Consistency:** The support \( S \) must be a consistent set of formulae. In other words, there exists a model for the conjunction of all formulae in \( S \). The program \( \pi_{\text{consistent}} \) simply consists of a guess which assigns truth values to all atoms and a constraint that removes any unsatisfiable support.

\[
\pi_{\text{consistent}} = \{ 1 \{ \text{true}(\text{consistent}, X), \text{false}(\text{consistent}, X) \} 1 \leftarrow \text{atom}(X). \}
\leftarrow \text{not model}(\text{consistent}, X), \text{fs}(X). \}
\]

The following program then gives all arguments that can be computed from a knowledge base \( K \) and a set of claims \( C \):

\[
\pi_{\text{arguments}} = \pi_{\text{modelcheck}} \cup \pi_{\text{arg}} \cup \pi_{\text{entailment}} \cup \pi_{\text{consist}} \cup \pi_{\text{minimize}} \cup \pi_{\text{consistent}}
\]

Each answer-set obtained by \( \pi_{\text{arguments}} \) contains the predicate \( \text{claim}(\cdot) \) and a set of predicates \( \text{fs}(\cdot) \), representing claim and support.

**Example 7.** Consider the input as given in Example 5. The program \( \pi_{\text{arguments}} \) returns the following answer-sets (we restrict ourselves to the relevant predicates):

\[
a_1 : \{ \text{fs}(a), \text{claim}(a) \} \\
a_2^* : \{ \text{fs}(\text{imp}(a, b)), \text{claim}(\text{imp}(a, b)) \} \\
a_3 : \{ \text{fs}(a), \text{fs}(\text{imp}(a, b)), \text{claim}(b) \} \\
a_4 : \{ \text{fs}(\neg(b)), \text{fs}(\text{imp}(a, b)), \text{claim}(\neg(a)) \} \\
a_5 : \{ \text{fs}(\neg(b)), \text{claim}(\neg(b)) \} \\
a_6 : \{ \text{fs}(a), \text{fs}(\neg(b)), \text{claim}(\text{and}(a, \neg(b))) \}
\]
Note that due to the definition of program \( \pi_{\text{minimize}} \) and \( \pi_{\text{consistent}} \) several resulting answer-sets may represent the same derived argument. This is the case for \( a_3^* \) where actually three models are derived by the program \( \pi_{\text{consistent}} \). They only differ in the respective truth assignments \( \text{true}(\text{consistent}, \cdot) \) and \( \text{false}(\text{consistent}, \cdot) \). We eliminate duplicates in an additional post-processing step in order to remove redundant information.

### 3.3 Identifying Conflicts between Arguments

We now want to compute attacks between arguments. Therefore we first specify encodings that are used by every attack type (such as defeat and direct defeat). We then present encodings for the computation of these attack types.

In order to reason over all arguments we first have to “flatten” the answer-sets obtained by \( \pi_{\text{arguments}} \). We specify this by the predicates \( \text{as}(A, fs, \cdot) \) and \( \text{as}(A, \text{claim}, \cdot) \). \( A \) is a numeric key identifying the argument.

**Example 8.** We illustrate this by the answer-sets \( a_1, a_2 \) and \( a_3 \) from Example 7. This input is given by the following facts:

\[
\{ \text{as}(1, fs, a), \text{as}(1, \text{claim}, a), \text{as}(2, fs, \text{imp}(a, b)), \text{as}(2, \text{claim}, \text{imp}(a, b)), \text{as}(3, fs, a), \text{as}(3, fs, \text{imp}(a, b)), \text{as}(3, \text{claim}, b) \}
\]

In order to identify conflicts between arguments we first guess two arguments. \( \text{selected1}(\cdot) \) and \( \text{selected2}(\cdot) \) contain the keys of the selected arguments.

\[
\pi_{\text{att}} = \begin{cases} 
1 \{ \text{selected1}(A) : \text{as}(A, \cdot, \cdot) \} & (1) \\
1 \{ \text{selected2}(A) : \text{as}(A, \cdot, \cdot) \} & (2)
\end{cases}
\]

Furthermore, we construct one single support formula for each argument \( A \) by conjunction of all formulae in \( \text{as}(A, fs, \cdot) \). As in the previous section we first define an ordering over all formulae that are contained in the support. The only difference is that we add the argument’s key \( A \) to the predicates \( \text{inf}(A, \cdot, \cdot), \text{sup}(A, \cdot, \cdot) \) and \( \text{succ}(A, \cdot, \cdot) \). Due to brevity, the corresponding program \( \pi_{\text{key}} \) is omitted. We can then construct the support formula by iterating over the ordering and connecting the formulae by conjunction. Note that the last parameter of \( \text{fs}\text{con}(A, \cdot, \cdot) \) is simply used as an identifier for the current position in the iteration. When the supremum is reached we derive \( \text{support}(A, \cdot, \cdot) \) for \( A \) containing the support formula.

\[
\pi_{\text{support}} = \begin{cases} 
\text{fs}\text{con}(A, X, X) \leftarrow \text{inf}(A, X), \text{sup}(A, X); & (1) \\
\text{fs}\text{con}(A, X, Y) \leftarrow \text{inf}(A, X), \text{succ}(A, X, Y); & (2) \\
\text{fs}\text{con}(A, O, N) \leftarrow \text{succ}(A, C, N), \text{fs}\text{con}(A, O, C); & (3) \\
\text{support}(A, X) \leftarrow \text{fs}\text{con}(A, X, C), \text{sup}(A, C). & (4)
\end{cases}
\]

For the computation of attacks we again apply the saturation technique. The program \( \pi_{\text{att_sat}} \) is used to saturate all attack computations. First, we derive all attack type keys \( f \) from the truth assignments \( \text{true}(t, \cdot, \cdot) \) and \( \text{false}(t, \cdot, \cdot) \) of the applied attack type programs. Note that the corresponding assignments are defined separately in each attack program. In case \( \text{attack} \) is derived for all truth assignments in some attack program
we saturate. Finally, the binary predicate \( \text{attack}(\cdot, \cdot) \) is generated. It is used for the representation of the attack relation between the two arguments.

\[
\pi_{\text{attacks}} = \{ \text{attacktype}(T) \leftarrow \text{true}(T,), \quad (1) \\
\quad \text{attacktype}(T) \leftarrow \text{false}(T,), \quad (2) \\
\quad \text{true}(T, X) \leftarrow \text{attack}, \text{atom}(X), \text{attacktype}(T); \quad (3) \\
\quad \text{false}(T, X) \leftarrow \text{attack}, \text{atom}(X), \text{attacktype}(T); \quad (4) \\
\quad \leftarrow \text{not attack;} \quad (5) \\
\quad \text{attack}(X, Y) \leftarrow \text{selected1}(X), \text{selected2}(Y). \} \quad (6)
\]

We now consider the attack types \emph{defeat} and \emph{direct defeat}. The basic idea is to define a propositional formula that represents the attack condition. We then assign \( \text{true}(t, x) \) or \( \text{false}(t, x) \) to any atom \( x \) using a disjunctive rule. In case every such interpretation is a model for our attack formula we know that the formula is valid. Otherwise, if any interpretation is not a model (i.e. \text{attack} is not derived) the resulting answer-set is strictly smaller than those where the interpretation is a model. Such answer-sets are removed by the constraint \( \leftarrow \text{not attack} \). Note that this also works in case we consider several attack types: If we derive \text{attack} for all interpretations of a single attack type we saturate all interpretations of \emph{all} attack types. As the predicate \text{attack} as well as all assignments \( \text{true}(t, x) \) and \( \text{false}(t, x) \) are then contained in all answer-sets for the two selected arguments we derive every attack relation.

The following program \( \pi_{\text{defeat}} \) exemplifies the above described approach. For two arguments \( A = (S, C) \) and \( A' = (S', \{\varphi'_1, \ldots, \varphi'_m\}, C') \) we have that \( A \) defeats \( A' \) if \( C \models \neg(\varphi'_1 \land \ldots \land \varphi'_m) \). Here we make use of the support derived by \( \pi_{\text{key}} \cup \pi_{\text{support}} \). As the support is defined as a conjunction of formulae we can directly make use of claim \( C \) and the negated support \( \neg S' \).

\[
\pi_{\text{defeat}} = \{ \text{checkDefeat}(\text{imp}(C, \neg(S''))) \leftarrow \text{selected1}(X), \text{selected2}(Y), \quad (1) \\
\text{as}(X, \text{claim}, C), \text{support}(Y, S''); \quad (2) \\
\text{subformula}(X) \leftarrow \text{checkDefeat}(X); \quad (3) \\
\text{true}(\text{defeat}, X) \vee \text{false}(\text{defeat}, X) \leftarrow \text{atom}(X); \quad (4) \\
\text{attack} \leftarrow \text{ismodel}(\text{defeat}, X), \text{checkDefeat}(X). \} \quad (5)
\]

The second program we consider here is \( \pi_{\text{ddefeat}} \). \( A \) directly defeats \( A' \) if \( C \models \neg\varphi'_i \) for a \( \varphi'_i \in S' \). Hence, we have to consider each formula in \( S' \) separately. Therefore we use a combination of attack type and \( \varphi'_i \) to identify the truth assignment.

\[
\pi_{\text{ddefeat}} = \{ \text{checkDirectdefeat}(\Phi, \text{imp}(C, \neg(\Phi))) \leftarrow \text{selected1}(X), \text{selected2}(Y), \quad (1) \\
\text{as}(X, \text{claim}, C), \text{as}(Y, \text{fs}, \Phi); \quad (2) \\
\text{subformula}(X) \leftarrow \text{checkDirectdefeat}(\cdot, X); \quad (3) \\
\text{true}(\text{ddefeat}(T), X) \vee \text{false}(\text{ddefeat}(T), X) \leftarrow \text{atom}(X), \text{checkDirectdefeat}(T, \cdot); \quad (4) \\
\text{attack} \leftarrow \text{ismodel}(\text{ddefeat}(T), X), \text{checkDirectdefeat}(T, X). \} \quad (5)
\]

The program \( \pi_{\text{attacks}} \) in combination with any attack type programs (such as \( \pi_{\text{defeat}}, \pi_{\text{ddefeat}}, \ldots \)) computes the respective attack relations.

\[
\pi_{\text{attacks}} = \pi_{\text{modelcheck}} \cup \pi_{\text{att}} \cup \pi_{\text{support}} \cup \pi_{\text{key}} \cup \pi_{\text{att sat}}
\]
Example 9. Continuing our running example, we now consider the flattened input as exemplified in Example 8 and the program $\pi_{\text{attacks}} \cup \pi_{\text{defeat}}$. We obtain 9 answer-sets that contain the attack information between arguments:

\{attack(4, 1).\}, \{attack(6, 2).\}, \{attack(4, 3).\}, \{attack(6, 4).\}, \{attack(3, 5).\},
\{attack(3, 4).\}, \{attack(6, 3).\}, \{attack(4, 6).\}, \{attack(3, 6).\}

The first answer-set, for example, represents that argument $a_4 = (\neg b, a \rightarrow b), \neg a$ attacks (directly defeats) $a_1 = \{a\}, a$.

3.4 Overall Approach at a Glance

To sum it up, the overall process of our instantiation-based approach for generating argumentation frameworks consists of the following steps:

1. A knowledge-base $\mathcal{K}$ and a set $\mathcal{C}$ of claims are used as input.
2. The encoding $\pi_{\text{arguments}} = \pi_{\text{modelcheck}} \cup \pi_{\text{arg}} \cup \pi_{\text{entailment}} \cup \pi_\text{=} \cup \pi_{\text{minimize}} \cup \pi_{\text{consistent}}$ defines how arguments are derived from $\mathcal{K}$ and $\mathcal{C}$. $\pi_{\text{modelcheck}}$ is generally used for evaluating formulae under truth assignments. Within $\pi_{\text{arg}}$, for each argument $A = (S, C)$ a claim $C \in \mathcal{C}$ and a support $S \subseteq \mathcal{K}$ is guessed. The encodings $\pi_{\text{entailment}}, \pi_\text{=} \cup \pi_{\text{minimize}}$ and $\pi_{\text{consistent}}$ guarantee that the support entails the claim, the support is subset minimal and that $S$ is a consistent set of formulae.
3. The resulting arguments are “flattened” and used as input for $\pi_{\text{attacks}}$.
4. The encoding $\pi_{\text{attacks}} = \pi_{\text{modelcheck}} \cup \pi_{\text{att}} \cup \pi_{\text{support}} \cup \pi_{\text{<key}} \cup \pi_{\text{attsat}}$ is shared by all attack types. $\pi_{\text{modelcheck}}$ is again needed for model checking. $\pi_{\text{att}}$ guesses two arguments at once. $\pi_{\text{support}} \cup \pi_{\text{<key}}$ constructs a single support formula for those arguments by connecting the contained formulae by conjunction. $\pi_{\text{attsat}}$ saturates all attack computations.
5. Any attack encodings, such as $\pi_{\text{defeat}}$ and $\pi_{\text{ddefeat}}$ can be used in combination with $\pi_{\text{attacks}}$ in order to compute the respective attack relations.

4 Visualization of Argumentation Frameworks

In order to visualize argumentation frameworks we make use of the purpose-built tool ARVis\(^5\). ARVis is intended for the visualization of answer-sets and their relations by means of a directed graph. Each node in the graph represents an answer-set and a directed edge between two arguments represents a relation.

We now describe the process of generating and visualizing argumentation frameworks by using the encodings $\pi_{\text{arguments}}$ and $\pi_{\text{attacks}}$. ARVis provides a wizard that handles the respective steps:

1. Obtain arguments: The program $\pi_{\text{arguments}}$ and a problem instance must be specified within ARVis. ARVis computes the arguments by invoking an ASP solver.
2. Flatten arguments: The arguments obtained in the previous step are “flattened”, i.e. a single set of facts is generated in order to be able to reason over all obtained arguments when computing the attacks.

\[^5\]http://dbai.tuwien.ac.at/proj/argumentation/vispartix/#ARVis
3. Obtain attacks: The program $π_{attacks}$ and the attack type programs are now specified. The relations between the arguments are computed.

4. Select attack predicate: In general, ARVis accepts any binary predicate that represents answer-set relations. We define $attack/2$ here.

5. Argumentation framework: We obtain a graph visualization consisting of arguments (vertices) and attacks (edges).

6. Export: The obtained argumentation framework can be exported for further processing.

Fig. 2. ARVis: Resulting argumentation framework

The argumentation framework resulting from our running example, as represented by ARVis, is given in Figure 2. The attacks correspond to direct defeats between arguments. Each argument in the graph is represented by its id. By selecting an argument its claim and support are shown in the text field on the right side. All encodings, ARVis and detailed configuration information is available at

http://dbai.tuwien.ac.at/proj/argumentation/vispartix

ARVis is a general-purpose tool that may also be used in many other areas of research: Consider, for example, the Traveling Salesperson Problem (TSP) where the first program computes cities and the second program outputs routes where every city is visited exactly once. In fact, ARVis can be used for any problem where one is interested in the relation between answer-sets. It is only necessary to specify two answer-set programs: One for generating answer-sets and a second one for computing relations between those answer-sets.

Our approach is different from other available ASP visualization tools: ASPViz [8] takes two answer-set programs as input, one for the problem encoding and one for the visualization. The latter is used for visualization for each answer-set of the former
separately. It is realized in Java and works with pre-defined predicates to extract the visualization of answer-sets. IDPDraw [29] works in a similar fashion, which augments the presentation by providing also time points to show the result in different evolutionary states. Kara [19] from the SeaLion development environment for ASP also provides visualization of answer-sets using special predicates. ASPIDE [14] gives the user the opportunity to visualize the dependency graph of the input program and thus allows for another type of representation.

5 Conclusion

In this paper, we have presented a novel ASP-based tool for constructing argumentation frameworks from a given knowledge base. We have provided here the concrete ASP encodings used to obtain such frameworks when logic-based arguments cf. [4] are employed. However, similar encodings for further approaches of argumentation are possible and subject of future work, as well as a performance evaluation of the presented approach to check its scalability with large knowledge bases.

When designing our tool, ARVis, we tried to keep it as flexible as possible such that the concrete construction of the framework can be specified in the logic programs. As it has turned out, ARVis is thus not only a tool for generating and visualizing argumentation frameworks but also for graphically representing relations between answer-sets in a user-specified manner. Ongoing work thus focuses on application areas where it is the relation between the answer-sets (rather than the single answer-sets) that can support the designer of logic programs or where this relation is the relevant output of an ASP encoding.

Acknowledgments

We would like to thank Thomas Ambroz and Andreas Jusits for implementing and adapting ARVis according to our requirements as well as Torsten Schaub for providing hints how to process arbitrary formulae via ASP.

This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT08-028, and by the Vienna University of Technology program “Innovative Projects”.

References

Abstract. In this paper we present an Action Language-Answer Set Programming based approach to solving planning and scheduling problems in hybrid domains - domains that exhibit both discrete and continuous behavior. We use action language H to represent the domain and then translate the resulting theory into an A-Prolog program. In this way, we reduce the problem of finding solutions to planning and scheduling problems to computing answer sets of A-Prolog programs. We cite a planning and scheduling example from the literature and show how to model it in H. We show how to translate the resulting H theory into an equivalent A-Prolog program. We compute the answer sets of the resulting program using a hybrid solver called EZCSP which loosely integrates a constraint solver with an answer set solver. The solver allows us reason about constraints over reals and compute solutions to complex planning and scheduling problems. Results have shown that our approach can be applied to any planning and scheduling problem in hybrid domains.

1 Introduction

In this paper we are interested in modeling intelligent agents capable of planning, acting and reasoning in a dynamic environment. We are primarily interested in hybrid domains - domains that exhibit both discrete and continuous behavior. Many real world situations such as a robot in a manufacturing plant, a decision control system of a space shuttle etc. deal with both discrete change as well as continuous change. For example, a decision control system in a space shuttle is capable of opening and closing valves of fuel tanks to supply fuel to some jets. The actions open and close change positions of valves. The valves will remain in a certain position as long as no open or close actions occurs. The change of position of valves is, therefore, discrete. However, the fuel level in the tank can change continuously with time when the valve is open or there is an incoming supply of fuel to the tank. This type of continuous change coupled with discrete change makes this domain hybrid. We are not only interested in modeling such domains but also to solve planning and scheduling problems in such domains. A simple planning and scheduling problem in the above example would be to fire a jet within 10 seconds. This would require opening and closing the appropriate valves and delivering fuel to the jet in time to achieve the goal.

To model intelligent agents we need to provide the agent with knowledge about its environment and its own capabilities and goals. There are several approaches to representing and reasoning about such knowledge. In this paper, we use action language
H - a high level language for representing and reasoning about actions and their effects. We chose this language mainly because it is capable of representing and reasoning about hybrid domains. A description written in H describes a transition diagram whose states correspond to possible physical states of the system and whose arcs are labeled by actions. A transition, \( \langle \sigma, a, \sigma' \rangle \), of a diagram denotes that action \( a \) is possible in state \( \sigma \) and that after the execution of \( a \) the system may move to state \( \sigma' \). The diagram consists of all possible trajectories of the system.

Language H [4] was developed by extending the signature of action language \( \mathcal{AL} \) [2] with a collection of numbers for representing time, a collection of functions of time, and \textit{fluents} \(^1\) with non-boolean values including fluents defined by functions of time (\textit{process fluents}). It also allows \textit{triggers} which specifies conditions under which actions are triggered. The semantics of \( \mathcal{AL} \) is based on the McCain-Turner equation [9]. A slightly modified version of this equation defines the semantics of H. Thus, both languages are based on the same underlying intuition. However, in \( \mathcal{AL} \) a state of the transition diagram is a snapshot of the world, whereas in H, a state is a snapshot of the world over a time interval. Process fluents are defined over this time interval. Prior to H, logic-based formalisms such as Situation Calculus [13] and Event Calculus [14] were extended to reason about process fluents. Both approaches demonstrate via examples how their approach can be used for reasoning about process fluents. However, in both approaches, it is difficult to express causal relations between fluents.

To solve planning and scheduling problems in hybrid domains, we use an action language-logic programming approach. We begin by representing the domain in H. To this extent, we come up with a theory of H (also called \textit{action description}) to describe actions and their effects. We implement this theory by translating it into an \textit{Answer Set Prolog} (A-Prolog) program, a class of logic programs under \textit{answer set} semantics [7, 8], and computing answer sets (models) of the resulting program. In this way, we reduce the problem of finding solutions to the planning and scheduling problem to computing answer sets of the A-Prolog program. In [5] we show that there is a one-to-one correspondence between models of A-Prolog programs and models of our specification.

The paper is organized as follows. In section 2, we revisit the syntax and semantics of H as described in [4]. In section 3, we cite a planning and scheduling example from [12] and come up with a theory of H to model it. In section 4, we present the main contribution of this paper which is the translation of the resulting H theory into an A-Prolog program. The example from [12] was run using ZENO, a least commitment planner that supports continuous change. After ZENO, researchers have come up with several efficient solvers. One significant contribution is the Planning Domain Description Language (PDDL) which was exclusively developed for planning purposes. A variant of this language, PDDL+ [6] is capable of modeling continuous change through the use of processes and events. Like H, the semantics of this language is defined using a labeled transition diagram. The language has been shown to represent a number of complex time dependent effects. However, one major limitation of the language is that it does not support \textit{derived predicates} - predicates defined in terms of other predicates for eg. \textit{above}(x, y) from the blocks world domain. There are no such limitations in H.

\(^1\) functions whose values depend on a state and may change as a result of actions
2 Preliminaries

In this section we revisit the syntax and semantics of H as described in [4].

2.1 Syntax

By sort we mean a non-empty countable collection of strings in some fixed alphabet. A sorted signature \( \Sigma \) is a collection of sorts and function symbols. A process signature is a sorted signature with special sorts time, action, and process. Sort time is normally identified with one of the standard numerical sorts with the exception that it contains an ordinal \( \omega \) such that for any \( x \in \text{time} \setminus \{ \omega \}, \omega > x \). No operations are defined over \( \omega \). If time is discrete, elements of \( \text{time} \setminus \{ \omega \} \) may be viewed as non-negative integers, otherwise they can be interpreted as either rational numbers, constructive real numbers, etc.

Sort process contains strings of the form \( \lambda T. f(T) \) where \( T \) is a variable ranging over time and \( f(T) \) is a mathematical expression (possibly) containing \( T \). A string \( \lambda T. f(T) \) represents a function defined over time. The \( \lambda \) is said to bind \( T \) in \( f(T) \). If the expression, \( f(T) \), does not contain any variables for time then \( \lambda T. f(T) \) is said to denote a constant function. For e.g. \( \lambda T. 0 \) denotes the constant function 0. For simplicity we assume that all functions from process have the same range denoted by the sort range(process). An example of a function from sort process is \( \lambda T. h - (g/2) \times (T - t)^2 \) which defines the height in meters, at time \( T \), of a freely falling object dropped from a height \( h \), \( T - t \) seconds before. The symbol \( g \) denotes the Earth’s average gravitational acceleration which is equal to 9.8 meters/sec^2.

Sort action is divided into subsorts agent and exogenous. Members of agent denote (unit) actions performed by an agent and members of exogenous denote non-agent actions. Both agent and exogenous actions will be referred to as actions. A compound action is a set of unit actions performed at the same time.

The collection of function symbols includes names for fluents and standard numerical functions. Each fluent name is associated with an arity - a number indicating the number of arguments. Intuitively, fluents are properties that may change as a result of actions. For example, the height of a brick held at a certain position above the ground could change when it is dropped. Every process signature contains reserved fluents start and end of sort time.

A term of sort \( s \) is defined as follows:

1. A string \( y \in s \) is a term of sort \( s \);
2. If \( t_1, \ldots, t_n \) are terms of sorts \( s_1, \ldots, s_n \) respectively and \( f : s_1 \times \ldots \times s_n \rightarrow s \) is a function symbol then \( f(t_1, \ldots, t_n) \) is a term of sort \( s \).

Notice that if \( f(\bar{x}) \) is a term of sort process and \( t \) is a term of sort time then \( f(\bar{x})(t) \) is a term of sort range(process). For example, to represent the height of brick \( b \) we can introduce a fluent \( \text{height}(b) \) of sort process. Then by \( \text{height}(b)(10) \) we denote the height of \( b \) at time 10. Similarly, \( \lambda T. 200 - (g/2) \times (T - t)^2(10) \) denotes the value of the function at \( T = 10 \) which is equal to 77.5.

An atom of \( \Sigma \) is a statement of the form \( t = y \) where \( t \) is a term of some sort \( s \) and \( y \in s \). Examples of atoms are end = 10, 2 + 3 = 5 etc. If \( t \) is a term built from fluent symbols
then such an atom is called a fluent atom. Examples of fluent atoms are 
\[ \text{height}(b) = \lambda T. 100 - \left( \frac{g}{2} \right) \cdot (T - 5)^2, \text{height}(b)(5) = 100, \text{end} = 10, \text{etc}. \]

A literal of \( \Sigma \) is an atom \( t = y \) or its negation \( \neg(t = y) \). Negation of \( = \) will be often written as \( \neq \). If \( t \) is a term of Boolean sort then \( t = \text{true} \) (\( t \neq \text{false} \)) is often written as \( t \) and \( t = \text{false} \) (\( t \neq \text{true} \)) is often written as \( \neg t \). For example, the atom \( 4 < 5 = \text{true} \) will be written as \( 4 < 5 \).

Language, \( H \), is parameterized by a process signature \( \Sigma \) with standard interpretations of numerical functions and relations (such as \( +, <, \leq, \neq \), etc).

**Definition 1.** An action description of \( H(\Sigma) \) is a collection of statements of the form:

\[
\begin{align*}
& l_0 \text{ if } l_1, \ldots, l_n. \quad (1) \\
& e \text{ causes } l_0 \text{ if } l_1, \ldots, l_n. \quad (2) \\
& \text{impossible } e_1, \ldots, e_m \text{ if } l_1, \ldots, l_n. \quad (3) \\
& l_1, \ldots, l_n \text{ triggers } e. \quad (4)
\end{align*}
\]

where \( e \)'s are elements of action, \( l_0 \)'s are fluent atoms and \( l_1, \ldots, l_n \) are literals of the signature of \( H \). \( l_0 \) is referred to as the head of a statement and \( l_1, \ldots, l_n \) are referred to as the body of a statement. A statement of the form (1) is called a state constraint. It guarantees that any state satisfying \( l_1, \ldots, l_n \) also satisfies \( l_0 \). A statement of the form (2) is called a dynamic causal law and it states that if action \( e \) were executed in a state satisfying literals \( l_1, \ldots, l_n \) then any successor state would satisfy \( l_0 \). A statement of the form (3) is called an executability condition and it states that actions \( e_1, \ldots, e_m \) cannot be executed in a state satisfying \( l_1, \ldots, l_n \). If \( n = 0 \) then \( if \) is dropped from statements (1), (2) and (3). A statement of the form (4) is called a trigger and it states that action \( e \) is triggered in any state satisfying \( l_1, \ldots, l_n \).

By ground instantiations of a variable of sort \( s \), we mean the elements of \( s \). From the description of the syntax, the only variable that appears in the statements of \( H \) is the variable \( T \) ranging over time. However, variables ranging over other sorts are allowed in the statements as long as those statements are viewed as a shorthand for the collection of statements obtained by replacing each occurrence of a variables other than \( T \) by its corresponding ground instantiations.

### 2.2 Semantics

The semantics of language \( H \) is based on a slightly modified McCain-Turner equation [9]. An action description, \( AD \), of \( H(\Sigma) \) describes a transition diagram, \( TD(AD) \), whose nodes correspond to possible physical states of a system and whose arcs are labeled by actions. A transition \( \langle s, a, s' \rangle \) of the diagram denotes that action \( a \) is possible in \( s \) and as a result of execution of \( a \) the system may move to state \( s' \). It is important to note that an action description of \( H \) can be either deterministic (i.e. for any state-action pair there is at most one successor state [2]) or non-deterministic (i.e. there is a state-action pair with more than one successor state). In this section we will give a formal definition for a state and a transition of \( TD(AD) \). We begin with interpreting symbols of \( \Sigma \).

**Definition 2.** Given an action description \( AD \) of \( H(\Sigma) \), an interpretation \( I \) of \( \Sigma \) is a mapping defined as follows.
– for every non-process sort, \( s \), and every string \( y \in s \), \( I \) maps \( y \) into itself i.e. \( y^I = y \).
– standard interpretation is used for the sort \( \text{process} \) and other standard numerical functions and relations.
– \( I \) maps every fluent into a properly typed function.

Often an interpretation \( I \) of \( \Sigma \) is identified with a collection, \( s(I) \), of atoms of the form \( t = y \) such that \( t^I = y \) where \( t \) and \( y \) are terms of some sort. In other words, \( s(I) = \{ t = y \mid t^I = y \} \).

Before we give the definition of a state of \( TD(AD) \) let us consider the following definitions.

A set, \( s \), of atoms is said to be \textit{consistent} if for every atom \( t = y_1 \in s \), \( \exists y_2 \) such that \( t = y_2 \in s \) and \( y_1 \neq y_2 \).

Let us define what it means for a literal to be true w.r.t a set of atoms of \( \Sigma \).

\textbf{Definition 3.} Given a consistent set, \( L \), of atoms of \( \Sigma \)

– An atom \( t = y \) is \textit{true in} \( L \) (symbolically \( L \models t = y \)) iff \( t = y \in L \).
– A literal \( t \neq y \) is \textit{true in} \( L \) (\( L \models t \neq y \)) iff \( L \models t = y_0 \) and \( y \neq y_0 \).

We will now define what it means for a set of atoms to be closed under state constraints of \( AD \).

\textbf{Definition 4.} A set \( L \) of atoms is closed under the state constraint

\[ \text{l}_0 \text{ if } \text{l}_1, \ldots, \text{l}_n \]

of \( AD \) if, whenever \( L \models \text{l}_i \) for every \( i, 1 \leq i \leq n \), \( L \models \text{l}_0 \).

A set \( L \) of atoms is closed under state constraints of \( AD \) if \( L \) is closed under every state constraint of \( AD \).

Next, we define what it means for a set of atoms to satisfy a trigger of \( AD \).

\textbf{Definition 5.} A set \( L \) of atoms of \( H \) satisfies a trigger

\[ \text{l}_1, \ldots, \text{l}_n \text{ triggers e} \]

of \( AD \) iff \( L \models \text{l}_i \) for every \( i \) such that \( 1 \leq i \leq n \).

Intuitively, if a set of atoms satisfies a trigger it means that the corresponding action will take place at some time point. The next definition characterizes sets of atoms which “end” before any triggered action takes place.

\textbf{Definition 6.} A set \( L \) of atoms of \( H \) is closed under triggers of \( AD \) iff \( \not\exists L' \) such that \( L' \) satisfies at least one trigger of \( AD \) and \( L \setminus L' = \{ \text{end} = t_2 \} \) and \( L' \setminus L = \{ \text{end} = t_1 \} \) and \( t_1 < t_2 \).

Intuitively, a state of \( TD(AD) \) can be viewed as a collection of functions of time defined over an interval. The endpoints of the interval are implicitly defined by the reserved fluents \( \text{start} \) and \( \text{end} \). The domain of each function is the set \( \{ t \mid \text{start} \leq t \leq \text{end} \land t < \omega \} \). We say that a state is defined over an interval of the form \([\text{start}, \text{end}]\) iff \( \text{end} \neq \omega \). There is at least one arc labeled by an action leading out of such a state. We say that a state is defined over an interval of the form \([\text{start}, \text{end}]\) iff \( \text{end} = \omega \). There is no arc leading out of such a state. States that begin at time 0 are called \textit{initial states}. They define the initial conditions of a domain. Here is the formal definition of a state.
Definition 7. Given an interpretation $I$ of $\Sigma$, $s(I)$ is a state of $TD(AD)$ if each of the following holds.

- $s(I)$ is a collection of atoms of the form $t = y$ such that $t' = y$ where $t$ and $y$ are terms of the same sort.
- $s(I)$ is closed under the state constraints of $AD$.
- If $s(I) \models end = t_2$ then $t_1 \leq t_2 \land t_1 < \omega$.
- $s(I)$ is closed under the triggers of $AD$.
- If $s(I) \models p = \lambda T.f(T)$ where $p$ is a fluent of sort process then $\lambda T.f(T)$ is defined over the domain $\{t \mid start^l \leq t \leq end^l \land t < \omega\}$.
- If $p$ is a fluent of sort process and $t$ is a term of sort time then $s(I) \models p(t) = x$ iff $s(I) \models p = \lambda T.f(T)$ and $\lambda T.f(T)(t') = x$.

By definition of interpretation every symbol is mapped uniquely. Therefore, states of $TD(AD)$ are complete and consistent. Whenever convenient the parameter $I$ will be dropped from $s(I)$.

Next, we will define what is means for an action to be possible in a state.

Definition 8. Action $a$ is possible in state $s$ if for every non-empty subset $a_0$ of $a$, there is no executability condition

$$\text{impossible } a_0 \text{ if } l_1, \ldots, l_n.$$

of $AD$ such that $s \models l_i$ for every $i$, $1 \leq i \leq n$.

Given a state $s$ and action $e$ let us define what are the direct effects of executing $e$ in $s$.

Definition 9. Let $e$ be an elementary action that is possible in state $s$. By $E_s(e)$ we denote the set of all direct effects of $e$ w.r.t. $s$.

$$E_s(e) = \{l_0 \mid e \text{ causes } l_0 \text{ if } l_1, \ldots, l_n \in AD \land s \models l_i \text{ for every } i, 1 \leq i \leq n\}$$

If $a$ is a compound action then $E_s(a) = \bigcup_{e \in a} E_s(e)$.

The following definition allows us to identify sets of atoms with adjacent intervals.

Definition 10. Let $x, y$, and $z$ be elements of sort time such that $x \leq y \leq z \land y < \omega$ and $s$ and $s'$ be sets of atoms of $H$. We say that $s'$ follows $s$ iff $s \models \{\text{start} = x, \text{end} = y\}$ and $s' \models \{\text{start} = y, \text{end} = z\}$.

Given two sets of atoms $s$ and $s'$, the function $T_s(s')$ is defined as follows.

$$T_s(s') = \begin{cases} \{\text{start} = t_1, \text{end} = t_2\} & \text{if } s' \text{ follows } s \land \{\text{start} = t_1, \text{end} = t_2\} \subseteq s' \\ \emptyset & \text{otherwise.} \end{cases}$$

In other words, the function returns the interval of $s'$ if $s'$ follows $s$; otherwise it returns an empty set.

The consequences of a set of atoms w.r.t a set of state constraints is defined as follows.

Definition 11. Given a set $S$ of atoms and a set $Z$ of state constraints of $AD$ the set, $Cn_Z(S)$, of consequences of $S$ under $Z$ is the smallest set of atoms (w.r.t set theoretic inclusion) containing $S$ and closed under $Z$. 
Definition 12. Action $a$ is complete w.r.t a set of atoms $s$ if for every trigger $r \in AD$ of the form
\[ l_1, \ldots, l_n \text{ triggers } e \]
e \in a \text{ iff } s \text{ satisfies } r.

We know that a state contains arbitrary atoms of $\Sigma$. However, for the next definition we focus only on fluent atoms including those formed from $\text{start}$ and $\text{end}$. All other atoms such as $2 + 3 = 5$ which is always true will be ignored.

Definition 13. A transition diagram $TD(AD)$ is a tuple $\langle \phi, \psi \rangle$ where
- $\phi$ is the set of states.
- $\psi$ is the set of all transitions $\langle s, a, s' \rangle$ such that each of the following holds.
  - $a$ is complete w.r.t $s$
  - $a$ is possible in $s$
  - $s'$ is closed under the triggers of $AD$
  - $s' = Cn_Z\left(E_s(a) \cup (s \cap s') \cup T_s(s')\right)$

where $Z$ is the set of state constraints of $AD$.

The set, $E_s(a)$, consists of direct effects of $a$ while the set, $s \cap s'$, consists of facts preserved by inertia. $T_s(s')$ projects the start and end of $s'$. The application of $Cn_Z$ to the union of these sets adds the indirect effects. From the definition, it is evident that concurrent actions such as dropping two balls from different heights at the same time can be handled in H.

3 A planning and scheduling example

In this section we will visit a planning and scheduling example from [12] and demonstrate how to model it in H. Consider a toy world in which a single plane moves passengers between cities. Slow flying travels at 400 miles per hour and consumes 1 gallon of fuel every 3 miles, on average. Fast flying travels at 600 miles per hour and consumes 1 gallon of fuel every 2 miles. Passengers can be boarded in 30 minutes and deplaned in 20 minutes. Refueling gradually increases the fuel level to a maximum of 750 gallons, taking one hour from an empty tank. Boarding, deplaning, and refueling must all occur while the plane is on the ground. The distance between city-a and city-b is 600 miles, the distance between city-a and city-c is 1000 miles, the distance between city-b and city-c is 800 miles and the distance between city-c and city-d is 1000 miles. Suppose that Dan and Ernie are at city-c, but the empty plane and Scott are at city-a. If the plane only has 500 gallons of fuel, how can we ensure that Scott and Ernie get to city-d in less than five and a half hours? [12]

A solution to this problem requires reasoning about effects of (concurrent) actions in the presence of continuous time. There are properties of the domain that change continuously with time such as the fuel level of the plane, the distance covered by the plane etc. With proper planning and scheduling, the plane will make its tight schedule without running out of fuel. Hence, the problem is a planning and scheduling problem in the presence of continuous change. We begin with the representation of the domain in H.
3.1 Representation in H

We begin with the description of the signature. It consists of several objects - persons scott, ernie, and dan and locations a,b,c, and d. We will use another location called enroute to denote that the plane is in the air. We will use (possibly indexed) variables P and L to denote persons and locations respectively. We are given the constants distance(a,b,600), distance(a,c,1000) and so on. We are also given the fuel consumption rates for different speeds. For example, when flying at 400 mph the mileage is 3 miles per gallon. We can encode this information in the form of constants fc(400) and fc(600) which denote 3 miles per gallon and 2 miles per gallon respectively. For modeling purposes we will use minutes as our unit of time.

We know that there are several durative actions in the domain, for example, boarding which takes 30 minutes. We will use the approach used by Reiter [13] to model durative actions. We introduce the action start_boarding(P,L) which denotes that person P is boarding at location L and end_boarding(P,L) which denotes that person P has finished boarding at location L. So instead of one action we have two actions start_boarding(P,L) and end_boarding(P,L) which make the fluent boarding(P,L) true and false respectively. In our model, the end_boarding(P,L) action will be triggered 30 minutes from the time start_boarding(P,L) is executed. We will use the same approach for all durative actions. We introduce the action start_flying(L1,L2,S) to denote that the plane has started flying from location L1 to location L2 with speed S. The variable S ranges over {400,600}. The remaining actions in the domain are start_deplaning(P,L), end_deplaning(P,L), start_refueling, end_refueling, and end_flying(L1,L2).

The signature consists of the boolean fluents boarding(P,L), deplaning(P,L), on_board(P) and refueling. We also have non-boolean fluents location(P) and location(plane) which range over {a,b,c,d,enroute}. We have process fluents time_left_board(P,L) and time_left_deplane(P,L) which are clock functions to count down the time left for person P to board and deplane at location L respectively. We have process fluent distance_left(L1,L2) to denote the distance that is yet to be covered to reach L2 from L1. Finally, the process fluent fuel_level denotes the fuel level in the plane’s tank. We proceed with defining the effects of each action. The effects of start_boarding(P,L) are defined using dynamic causal laws

\[
\text{start\_boarding}(P,L) \text{ causes } \text{boarding}(P,L) .
\]

\[
\text{start\_boarding}(P,L) \text{ causes } \text{time\_left\_board}(P,L) = \lambda T. \max(0, 30 - (T - T_0)) \text{ if end = T_0 }.
\]

The following executability condition says that person P cannot start boarding if he or she is already boarding.

impossible \text{start\_boarding}(P,L) \text{ if } \text{boarding}(P,L).

The following are causal laws involving end_boarding(P,L).

\[
\text{end\_boarding}(P,L) \text{ causes } \neg\text{boarding}(P,L).
\]

\[
\text{end\_boarding}(P,L) \text{ causes } \text{on\_board}(P).
\]

impossible \text{end\_boarding}(P,L) \text{ if } \neg\text{boarding}(P,L).
The following are causal laws involving actions \textit{start\_deplaning}(P,L) and \textit{end\_deplaning}(P,L).

\textit{start\_deplaning}(P,L) causes \textit{deplaning}(P,L)
\textit{start\_deplaning}(P,L) causes 
\[ time\_left\_deplane(P,L) = \lambda T \max(0, 20 - (T - T_0)) \]
if \text{end} = T_0.

impossible \textit{start\_deplaning}(P,L) if \textit{deplaning}(P,L).
\textit{end\_deplaning}(P,L) causes \neg \textit{deplaning}(P,L).
\textit{end\_deplaning}(P,L) causes \neg \textit{on\_board}(P).
impossible \textit{end\_deplaning}(P,L) if \neg \textit{deplaning}(P,L).

The following are direct effects of refueling. Instead of assuming that it takes 1 hour to fill up from an empty tank we will assume that the rate of refueling is 20 gallons per minute. When we start refueling, we add fuel to the existing level and the level increases at the rate of 20 gallons per minute.

\textit{start\_refueling} causes \textit{fuel\_level} = \lambda T \max(750, X + 20 \times (T - T_0))
if \text{end} = T_0, \text{fuel\_level}(\text{end}) = X.

\textit{start\_refueling} causes \textit{refueling}.
impossible \textit{start\_refueling} if \textit{refueling}.
impossible \textit{start\_refueling} if \textit{location}(plane) = \textit{enroute}.
\textit{end\_refueling} causes \neg \textit{refueling}.
impossible \textit{end\_refueling} if \textit{location}(plane) = \textit{enroute}.
impossible \textit{end\_refueling} if \neg \textit{refueling}.

Next, we define the direct effects of \textit{start\_flying}(L_1,L_2,S).

\textit{start\_flying}(L_1,L_2,S) causes \textit{location}(plane) = \textit{enroute}.
\textit{start\_flying}(L_1,L_2,S) causes \textit{distance\_left}(L_1,L_2) = \lambda T \max(0, X - S \times (T - T_0)/60)
if \text{distance}(L_1,L_2,X), \text{end} = T_0.
\textit{start\_flying}(L_1,L_2,S) causes \textit{fuel\_level} = \lambda T \max(0, X - S \times (T - T_0)/(60 \times f\_c(S)))
if \text{fuel\_level}(\text{end}) = X, \text{end} = T_0.
impossible \textit{start\_flying}(L_1,L_2,S) if \textit{location}(plane) = \textit{enroute}.
impossible \textit{start\_flying}(L_1,L_2,S) if \textit{location}(plane) \neq L_1.
impossible \textit{start\_flying}(L_1,L_2,S) if \textit{boarding}(P,L_1), \textit{location}(P) = L_1.

The following are causal laws involving action \textit{end\_flying}(L_1,L_2).

\textit{end\_flying}(L_1,L_2) causes \textit{location}(plane) = L_2.
impossible \textit{end\_flying}(L_1,L_2) if \textit{location}(plane) \neq \textit{enroute}.

The following causal laws specify conditions under which each “ending” action is triggered.
\textit{time\_left\_board}(P,L,\text{end}) = 0 triggers \textit{end\_boarding}(P,L).
\textit{time\_left\_deplane}(P,L,\text{end}) = 0 triggers \textit{end\_deplaning}(P,L).
\textit{fuel\_level}(\text{end}) = 750 triggers \textit{end\_refueling}.
\textit{distance\_left}(L_1,L_2,\text{end}) = 0 triggers \textit{end\_flying}(L_1,L_2).
The following state constraint encodes common sense knowledge. It states that a person on board a plane is at the same location as the plane.

\[ \text{location}(P) = L \text{ if } \text{location}(\text{plane}) = L, \text{on_board}(P). \]

The following executability condition states that flying to a location without enough fuel is not allowed. A rational reasoner who wants to reach his or her destination safely will not violate this constraint.

\[ \text{impossible start_flying}(L_1, L_2, S) \text{ if } \text{fuel_level}(end) = X, \]
\[ \text{distance}(L_1, L_2, Y), \]
\[ X < Y / f_c(S). \]

This concludes the representation of the domain in H.

### 4 Translation into logic program

We solve the planning and scheduling problem by translating the above theory into an Answer Set Prolog program and computing answer sets of the resulting program. Thus, we reduce the problem of finding solutions to the planning and scheduling problem to computing answer sets of the resulting program. The translation that was provided in [5] is a theoretical translation from statements of H into rules of A-Prolog. However, in this paper we are going to provide a translation that adheres to the syntax of a specific ASP solver. Since we are dealing with continuous functions we chose a solver that, in addition to computing answer sets, is capable of reasoning about constraints over reals. Here is a brief summary of solvers that are available at our disposal.

In the past few years researchers [1, 10] have focused on integrating Answer Set Programming(ASP) and Constraint logic programming(CLP). They came up with new systems that achieve significant improvement in performance over existing ASP solvers. The following such systems are available: \textsc{ACsolver}[10] (and it’s successor \textit{Luna}[11]): \textsc{EZCSP} \(^2\); and \textit{Clingcon}\(^3\). Each solver couples a constraint solver with an answer set solver. However, the coupling varies. In \textit{Luna} (\textsc{ACsolver}) and \textit{Clingcon} the coupling is tight whereas in \textsc{EZCSP} the coupling is loose. We considered using \textit{Luna} for our purposes but we found out that there are some implementation issues. Besides, the underlying answer set solver is less efficient than the one used by \textsc{EZCSP}. We also considered using \textit{Clingcon}, however, the underlying constraint solver only deals with finite domains which is a limitation when dealing with continuous functions. We chose \textsc{EZCSP} as it allows us to reason about constraints over reals.

In \textsc{EZCSP}, the A-Prolog programs are written in such a way that their answer sets encode a constraint satisfaction problem. The system calls the answer set solver, uses the resulting answer sets to pose a constraint satisfaction problem in the input language of the constraint solver, then calls the constraint solver and combines the solutions returned by both solvers. The constraint solver can use the results from the answer set

\(^2\) \url{http://marcy.cjb.net/ezcsp/index.html}

\(^3\) \url{http://www.cs.uni-potsdam.de/clingcon/}
solver to solve new constraints but the answer set solver cannot use results from the constraint solver to make new inferences. The current version uses gringo+clasp\(^4\) by default as the ASP solver and SICSTUS Prolog as the constraint solver. Both solvers are very fast.

The translation that we present here will be in the input language of EZCSP. Before we begin the translation there are some modeling decisions to be made. These decisions are necessary to overcome some of the implementation issues. One of the main issues is the implementation of functions. The input language of EZCSP will allow us to represent constants and variables but it is not possible to represent functions of time. For example, we can translate statements such as \(\text{velocity} = 5\) directly into the input language of EZCSP but it is not possible to translate a statement such as \(\text{height} = \lambda T . T^2\). The reason is that there is no representation for functions in this language. The alternative is to provide a value for \(T\) and obtain the value of the function for that instance. So it is possible to translate the statement \(\text{height}(5) = \lambda T . T^2(5) = 25\). The constraint solver may obtain the value for \(T\) from other constraints and then use that value to evaluate \(\text{height}(5)\).

Sorts, actions, boolean fluents and fluents ranging over finite domains will be translated as atoms in the input language of EZCSP. All other fluents ranging over infinite or large domains will be represented as constraint variables. Every constraint involving a constraint variable will appear in an atom called \(\text{required}\). For example, we write \(\text{required}(\text{height} > 5)\) to denote the variable \(\text{height}\) must be greater than 5. Process fluents will be represented by constraint variables. For each process fluent we can use two constraint variables - one to denote the initial value and one to denote the final value in a given state.

Let us look at some statements from our example and see how they are translated into EZCSP rules. Consider the following statement from our example.

\[
\text{start refueling} \text{ causes } f_{\text{initial}} = X \text{ if } f_{\text{final}} = X.
\]
\[
\text{start refueling} \text{ causes } f_{\text{time}} = T \text{ if } \text{end} = T_0, f_{\text{level}}(\text{end}) = X.
\]

Since we cannot encode functions in EZCSP we are going to expand the signature to include a boolean fluent called \(\text{refueling}\), a real-valued fluent called \(f_{\text{time}}\) to denote the time at which fueling began and two new real-valued fluents - \(f_{\text{initial}}\) and \(f_{\text{final}}\) to denote the initial and final fuel level in a state. We then replace the above causal law with the following causal laws.

\[
\text{start refueling} \text{ causes } f_{\text{initial}} = X \text{ if } f_{\text{final}} = X.
\]
\[
\text{start refueling} \text{ causes } f_{\text{time}} = T \text{ if } \text{end} = T_0, f_{\text{level}}(\text{end}) = X.
\]
\[
\text{start refueling} \text{ causes } f_{\text{final}} = \max(750, X + 20 \ast (T - T_0)) \text{ if } f_{\text{initial}} = X, f_{\text{time}} = T_0, \text{end} = T, \text{refueling}.
\]

The first two dynamic causal laws say that executing \(\text{start refueling}\) initializes \(f_{\text{initial}}\) to the current fuel level and \(f_{\text{time}}\) to the end time of the current state. The next dynamic law says that \(\text{start refueling}\) begins the \(\text{refueling}\) process. Next, we have a state

\(^4\) http://potassco.sourceforge.net/
constraint which defines $f_{final}$ in terms of $f_{initial}$, $f_{time}$ and end of a state in which refueling is true. We consider fluents $f_{initial}$, $f_{time}$ and refueling to be inertial. These fluents will retain their values until an action causes them to change. For example, $f_{initial}$ will be reset by actions end_refueling, start_flying($L_1, L_2, S$) and end_flying($L_1, L_2$) because each action has a bearing on the fuel level. So we will add dynamic causal laws encoding the effects of these actions on $f_{initial}$. The relationship between the process fluent fuel_level in the original action description and $f_{final}$ in the modified action description is that for any given state, $f_{final} = fuel_level(end)$.

To translate the above statements into EZCSP we have to first define the various constraint variables. All the real-valued fluents will be translated as constraint variables. Since fluents are dependent on state, we have to consider state as one of its parameters during translation. It is important to note that with the assignment of time intervals to states, each state in a transition diagram is unique. Our approach is to assign (non-negative) integers to each state in the order it appears in the trajectory of a transition diagram. This is done in the language of EZCSP as follows.

$$\#const n = 10.$$  
$$step(0..n). \#domain step(I;11).$$

Here $n$ denotes the length of the trajectory and the $\#domain$ declaration says that variables $I$ and $I1$ range over $[0, n]$. Next, we will define the various constraint variables parameterized by $step$.

$$cspvar(f_{time}(I), 0, 400). \quad cspvar(f_{initial}(I), 0, 750).$$  
$$cspvar(f_{final}(I), 0, 750). \quad cspvar(end(I), 0, 400).$$

The numbers in each declaration specify the range of that variable. Next, the boolean fluent refueling will be translated into the atom $v(X, refueling, I)$ where $X$ ranges over \{true, false\}. It says that $X$ is the value of refueling in step $I$. The action start_refueling is translated as occurs(start_refueling, $I$) which says that start_refueling occurred in step $I$. Finally, here are the EZCSP rules obtained by translating the above causal laws.

$$required(f_{initial}(I) == f_{final}(I)) : - occurs(start_refueling, I), I1 = I + 1.$$  
$$required(f_{time}(I) == end(I)) : - occurs(start_refueling, I), I1 = I + 1.$$  
$$v(true, refueling, I + 1) : - occurs(start_refueling, I).$$  
$$required(f_{final}(I) == max(750, f_{initial}(I) + 20*(end(I) - f_{time}(I)))) : - v(true, refueling, I).$$

The rest of the dynamic causal laws are translated using a similar approach. As mentioned before, in EZCSP the answer set solver does not receive any input from the constraint solver to make new inferences. This poses a problem especially when translating triggers. A trigger mentions the conditions under which an action will be executed. It is possible that these conditions are constraints over time. Even if these constrains are evaluated, since there is no feedback to the answer set solver, it is not possible to know when an action is triggered. The author of EZCSP, Marcello Balduccini, has suggested a solution to overcome this problem. Suppose that we have the following trigger.

$$fuel_level(end) = 750\text{ triggers } end_{refueling}.$$
Since we have replaced process fluent \textit{fuel. level} with other real valued fluents, this trigger is really

\[ f_{initial} = X, f_{time} = T, refueling, end = (750 - X)/20 + T \] triggers \textit{end. refueling}.

Given the initial fuel level \( X \) and the refuel rate of 20 gallons per minute, the expression \((750 - X)/20\) gives the number of minutes it takes to fill up the tank. The trigger states that if a state ends at a time when the tank is full then \textit{end. refueling} is triggered. Of course, \textit{refueling} must be also be true in that state. A direct translation of this trigger is

\[ \text{occurs}(end\_refueling,I) : \neg \text{required}(end(I) == (750 - f_{initial}(I))/20 + f_{time}(I)), v(true, refueling, I). \]

This rule will be fired if the body is satisfied. The atom \( v(true, refueling, I) \) is a direct consequence of \textit{start. refueling}. However, the constraint atom in the body of the rule is not obtained from other rules of the program. Balduccini suggested that it can be generated by adding the rules

\[ \{p(I), q(I)\}1 : \neg v(true, refueling, I). \]
\[ \text{required}(end(I) == (750 - f_{initial}(I))/20 + f_{time}(I)) : \neg p(I). \]
\[ \text{required}(end(I) < (750 - f_{initial}(I))/20 + f_{time}(I)) : \neg q(I). \]

The choice rule will allow us to generate one of the two constraints. If no action takes place before the tank fills up then \textit{end. refueling} is triggered. We use the same approach to translate other triggers in the domain. Currently, there is no formal proof stating that this solution will always work. However, the solution is based on answer set programming methodology and EZCSP uses a state-of-the-art answer set solver. We rely on the solver to give us the correct solutions.

In our example, the goal is to have Scott and Ernie in city \( d \) in less than five and a half hours (330 minutes). We encode this goal in EZCSP as follows

\[ \text{goal}(I) : \neg v(d, location(scott), I), v(d, location(ernie), I), \]
\[ \text{required}(start(I) < 330). \]

where \textit{start}(I) is a constraint variable denoting the start time of a step. The constraint says that the goal state must be achieved in less than five and a half hours. It is generated by adding the rules

\[ \{g(I), ng(I)\}1. \]
\[ \text{required}(start(I) < 330) : \neg g(I). \]
\[ \text{required}(start(I) >= 330) : \neg ng(I). \]

We add the following rules to say that failure is not an option.

\[ \text{success} : \neg \text{goal}(I). \]
\[ \neg \text{not success}. \]

Next, we will talk about the planning component of the program. As in the case of non-continuous domains, we will use Answer Set Programming techniques for generating
and testing plans. A plan is a sequence of actions. We will be generating all possible sequences of actions that will lead us to our goal. In the process of generating these plans we use the constraints in our program to test these plans and discard the ones that violate the constraints. A simple way to generate plans is to use the choice rule

\[
\{\text{occurs}(A,I) : \text{action}(A)\} : -\text{step}(I), I < n.
\]

where \(n\) is the length of the plan. Note that we are going to generate only the initiating actions for example, \text{start_re fueling} etc. This is because all the terminating actions have triggers associated with them and there is no need to generate them again. Since we are dealing with continuous time, in addition to computing the sequences of actions, we will also determine when these actions will take place. This is the reason why planning in hybrid domains also involves scheduling. According to the problem specification, we have a constraint that specifies the time within which the goal has to be achieved i.e. five and a half hours. We can specify this constraint in EZCSP as follows.

\[
\text{required}(\text{end}(I) < 330) : -\text{occurs}(A,I), \text{action}(A).
\]

The rules states that all action occurrences must take place before five and a half hours. This constraint gives a very broad range of scheduling possibilities. The times during which actions will take place could now range over intervals of time. For example, in order to achieve our goal it may be required that Scott has to board the plane within the first 50 minutes and that the plane has to depart city \(c\) no later than three hours and 45 minutes into the trip. Dealing with time intervals is an issue because it does not allow us to compute values of fluents at specific time points. More issues arise when fluents are defined in terms of other fluents whose values are unknown. To overcome these issues we decided to assign specific times to the initiating actions. The times for the terminating actions were determined by our triggers. Here is a solution given by EZCSP to our planning and scheduling problem.

\[
\begin{align*}
\text{occurs}(&\text{start_boarding}(\text{scott}, a), 0) & \text{end}(0) = 5.0 \\
\text{occurs}(&\text{end_boarding}(\text{scott}, a), 1) & \text{end}(1) = 35.0 \\
\text{occurs}(&\text{start_flying}(a,c,400), 2) & \text{end}(2) = 40.0 \\
\text{occurs}(&\text{end_flying}(a,c), 3) & \text{end}(3) = 190.0 \\
\text{occurs}(&\text{start_refueling}, 4) & \text{end}(4) = 195.0 \\
\text{occurs}(&\text{start_boarding}(\text{ernie}, c), 5) & \text{end}(5) = 197.0 \\
\text{occurs}(&\text{end_refueling}, 6) & \text{end}(6) = 224.16 \\
\text{occurs}(&\text{end_boarding}(\text{ernie}, c), 7) & \text{end}(7) = 227.0 \\
\text{occurs}(&\text{start_flying}(c,d,600), 8) & \text{end}(8) = 229.0 \\
\text{occurs}(&\text{end_flying}(c,d), 9) & \text{end}(9) = 329.0
\end{align*}
\]

It is necessary to refuel in city \(c\) because there is not enough fuel to travel to city \(d\). A shorter plan can be obtained by allowing concurrent actions. For example, refueling and boarding can start at the same time. The running times and the performance of the solver will be discussed in a longer version of the paper.
5 Conclusions and Future Work

In this paper, we presented an Action Language-Answer Set Programming based approach to solving planning and scheduling problems in hybrid domains. We used action language H to model a planning and scheduling example and translated the resulting theory into an A-Prolog program. We used a hybrid solver called EZCSP to compute the answer sets of the resulting program. Our approach overcomes the limitations of existing formalisms such as PDDL+. We believe that our approach can be applied to any planning and scheduling problem in hybrid domains.

In the future, we would like to model several benchmark examples and compare the performance of EZCSP with existing planners. Some of the planners used for industry-sized problems are domain-specific [3]. It will be useful to investigate why some of these planners work really well. There are areas for improvement including the efficiency of solvers, the expressiveness of the input language and so on.

References

Eliminating Unfounded Set Checking for HEX-Programs*

Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller

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

Abstract. HEX-programs are an extension of the Answer Set Programming (ASP) paradigm incorporating external means of computation into the declarative programming language through so-called external atoms. Their semantics is defined in terms of minimal models of the Faber-Leone-Pfeifer (FLP) reduct. Developing native solvers for HEX-programs based on an appropriate notion of unfounded sets has been subject to recent research for reasons of efficiency. Although this has lead to an improvement over naïve minimality checking using the FLP reduct, testing for foundedness remains a computationally expensive task. In this work we improve on HEX-program evaluation in this respect by identifying a syntactic class of programs, that can be efficiently recognized and allows to entirely skip the foundedness check. Moreover, we develop criteria for decomposing a program into components, such that the search for unfounded sets can be restricted. Observing that our results apply to many HEX-program applications provides analytic evidence for the significance and effectiveness of our approach, which is complemented by a brief discussion of preliminary experimental validation.

Keywords: Answer Set Programming, Nonmonotonic Reasoning, Unfounded Sets, FLP Semantics

1 Introduction

In the last years, Answer Set Programming (ASP) has emerged as an increasingly popular approach to declarative problem solving for a range of applications [2], thanks to expressive and efficient systems like SMODELS [20], DLV [19], cmodels [17], and CLASP [15]. However, recent developments in computing, in which context awareness, distribution and heterogeneous information sources gain importance, raised the need for access to external sources in programs, be it in the context of the Web to access web services, databases, or ontological information in different formats, in the context of agents to acquire sensor input, etc.

To cater for this need, HEX-programs [11] extend ASP with so called external atoms, through which the user can couple any external data source with a logic program. Roughly, such atoms pass information from the program, given by predicate extensions,
into an external source which returns output values of an (abstract) function that it computes. This extension has been utilized for a range of applications, including querying data and ontologies on the Web, multi-context reasoning, and reasoning about actions and planning, to mention a few (cf. [5]). Notably, recursive data exchange between the rules and the external sources is supported, which makes the formalism powerful.

The semantics of a \( \text{HEX} \)-program \( \Pi \) is defined in terms of answer sets based on the FLP reduct [14]: an interpretation \( A \) is an answer set of \( \Pi \), if and only if it is a \( \subseteq \)-minimal model of the FLP-reduct \( f\Pi^A \) of \( \Pi \) wrt. \( A \), which is the set of all rules whose body is satisfied by \( A \). For ordinary logic programs, this semantics coincides with the one where the canonical GL-reduct [16] is in place of \( f\Pi^A \), and it is more appealing for extensions with nonmonotonic aggregates [14], and the more general external atoms in \( \text{HEX} \)-programs.

The evaluation of a \( \text{HEX} \)-program \( \Pi \) in the \( \text{DLVHEX}^1 \) solver proceeds in two steps as follows. In Step 1, external atoms are viewed as ordinary atoms (replacement atoms) and their truth values are guessed by choice rules that are added. The resulting ordinary ASP program \( \hat{\Pi} \) is then evaluated by an ordinary ASP solver and each of its answer sets \( \hat{A} \) is checked against the external sources, i.e., the guess is verified. After that, the guess for the non-replacement atoms, called \( A \), is known to be a model of \( \Pi \), and thus also of the reduct \( f\Pi^A \). Step 2 then checks whether \( A \) is a \( \subseteq \)-minimal model or, equivalently, whether \( A \) is unfounded-free [13], i.e., there exists no unfounded set (UFS) of \( \Pi \) wrt. \( A \).

Unfortunately, Step 2 is computationally expensive in general, and it is intractable even for Horn programs with nonmonotonic external atoms of polynomial complexity, as follows from results in [14]. It is thus worthwhile to be aware of cases where this test is tractable, or even better, superfluous such that Step 2 can be skipped.

Motivated by this issue, we consider in this paper programs \( \Pi \) for which the result of Step 1 is a \( \subseteq \)-minimal model of the reduct \( f\Pi^A \). We provide a sound syntactic criterion for deciding whether the minimality check is needed, and in further elaboration, we describe how a program can be decomposed into program components such that unfoundedness checks can be delegated to the components, and the necessity of Step 2 thus be assessed on a finer-grained level.

More in detail, our main contributions are the following:

- We present a syntactic decision criterion which can be used to decide whether a program possibly has unfounded sets. If the result of this check is negative, then the computationally expensive search for unfounded sets can be skipped. The criterion is based on atom dependency and, loosely speaking states that there are no cyclic dependencies of ground atoms through external atoms. This criterion can be efficiently checked for a given ground \( \text{HEX} \)-program using standard methods, and in fact applies to a range of applications, in particular, for input-stratified programs, where external sources are accessed in a workflow to produce input for the next stage of computation. However, there are relevant applications of \( \text{HEX} \)-programs where cycles through external atoms are essential, e.g., in encodings of problems on multi-context systems [1] or abstract argumentation systems [4], for which Step 2 cannot be skipped.

- In further elaboration, we consider a decomposition of a program \( \Pi \) into components based on the dependency graph that is induced by the program. We show that \( \Pi \) has

\[ \text{http://www.kr.tuwien.ac.at/research/systems/dlvhex/} \]
some unfound set with respect to the candidate answer set \( A \) if and only if (at least) one of the components \( \Pi_C \) in the decomposition has some unfound set wrt. \( A \); note that computing the decomposition is efficiently possible, and thus does not incur a large overhead. This allows us to apply the decision criterion for the necessity of Step 2 efficiently on a more fine-grained level, and the search for unfound sets can be guided to relevant parts of the program. In particular, for the HEX-encoding of a Dung-style argumentation semantics [4] which we consider, the decomposition approach yields a considerable gain, as shown in a preliminary experimental evaluation.

This paper complements recent work on unfoundness checking for HEX-programs in [7, 8], which is part of a larger effort to provide efficient evaluation of HEX-programs, based on new algorithms cf. [6]. By their wide applicability, our results are significant especially for many potential applications in practice.

2 Preliminaries

In this section, we start with some basic definitions, and then introduce syntax and semantics of HEX-programs and the notion of unfound sets we are going to use.

A (signed) literal is a positive or a negative formula \( Ta \) resp. \( Fa \), where \( a \) is a ground atom of form \( p(c_1, \ldots, c_r) \), with predicate \( p \) and constants \( c_1, \ldots, c_r \), abbreviated \( p(c) \). For a literal \( \sigma = Ta \) or \( \sigma = Fa \), let \( \bar{\sigma} \) denote its opposite, i.e., \( \overline{Ta} = Fa \) and \( \overline{Fa} = Ta \).

An assignment is a consistent set of literals \( Ta \) or \( Fa \), where \( Ta \) expresses that \( a \in A \) and \( Fa \) that \( a \notin A \). \( A \) is complete, also called an interpretation, if no assignment \( A' \supset A \) exists. We denote by \( A^T = \{ a \mid Ta \in A \} \) and \( A^F = \{ a \mid Fa \in A \} \) the set of atoms that are true, resp. false in \( A \), and by \( ext(q, A) = \{ c \mid Ta(q(c)) \in A \} \) the extension of a predicate \( q \). Furthermore, \( A|_q = \{ c \mid q(c) \in A \} \) is the set of all literals over atoms of form \( q(c) \) in \( A \). For a list \( q = q_1, \ldots, q_k \) of predicates we write \( p \in q \) iff \( q_i = p \) for some \( 1 \leq i \leq k \), and let \( A|_q = \bigcup_j A|_{q_j} \).

A nogood is a set \( \{ L_1, \ldots, L_n \} \) of literals \( L_i \), \( 1 \leq i \leq n \). An interpretation \( A \) is a solution to a nogood \( \delta \) (resp. a set \( \Delta \) of nogoods), iff \( \delta \nsubseteq A \) (resp. \( \delta \nsubseteq A \) for all \( \delta \in \Delta \)).

2.1 HEX-Programs

HEX-programs were introduced in [11] as a generalization of (disjunctive) extended logic programs under the answer set semantics [16]; for details and background see [11].

Syntax. HEX-programs extend ordinary ASP programs by external atoms, which enable a bidirectional interaction between a program and external sources of computation. External atoms have a list of input parameters (constants or predicate names) and a list of output parameters. Informally, to evaluate an external atom, the reasoner passes the constants and extensions of the predicates in the input tuple to the external source associated with the external atom. The external source computes output tuples which are matched with the output list. More formally, a ground external atom is of the form

\[ \&q(p)(c), \]

where \( p = p_1, \ldots, p_k \) are constant input parameters (predicate names or object constants), and \( c = c_1, \ldots, c_l \) are constant output terms.

Ground HEX-programs are then defined similar to ground ordinary ASP programs.
Definition 1 (Ground HEX-programs). A ground HEX-program consists of rules
\[ a_1 \lor \cdots \lor a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n, \]
where each \( a_i \) is an (ordinary) ground atom \( p(c_1, \ldots, c_\ell) \) with constants \( c_i \), \( 1 \leq i \leq \ell \), each \( b_j \) is either an ordinary ground atom or a ground external atom, and \( k + n > 0 \).

The head of a rule \( r \) is \( H(r) = \{ a_1, \ldots, a_n \} \) and the body is \( B(r) = \{ b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \} \). We call \( b \) or not \( b \) in a rule body a default literal; \( B^+(r) = \{ b_1, \ldots, b_m \} \) is the positive body, \( B^-(r) = \{ b_{m+1}, \ldots, b_n \} \) is the negative body. For a program \( \Pi \), let \( A(\Pi) \) be the set of all ordinary atoms occurring in \( \Pi \).

We also use non-ground programs. However, as suitable safety conditions allow for using a grounding procedure [12], we limit our investigation to ground programs.

Semantics and Evaluation. Intuitively, a ground external atom \( \&g[p](c) \) is true, if the external source \( \&g \) yields output tuple \( c \) when evaluated with input \( p \). Formally, the semantics of a ground external atom \( \&g[p](c) \) wrt. an interpretation \( A \) is given by the value of a \( 1+k+l \)-ary Boolean oracle function \( f_{\&g} \) that is defined for all possible values of \( A, \ p, \ c \), where \( k \) is the length of \( p \) and \( l \) is the length of \( c \). Thus, \( \&g[p](c) \) is true relative to \( A \) if and only if it holds that \( f_{\&g}(A, p, c) = 1 \). Satisfaction of ordinary rules and ASP programs [16] is then extended to HEX-rules and programs in the obvious way, and the notion of extension \( ext(\cdot, A) \) for external predicates \( \&g \) with input lists \( p \) is naturally defined by \( ext(\&g[p], A) = \{ c \mid f_{\&g}(A, p, c) = 1 \} \).

Definition 2 (FLP-Reduct [14]). For an interpretation \( A \) over a program \( \Pi \), the FLP-reduct \( f_\Pi^A \) of \( \Pi \) wrt. \( A \) is the set \( \{ r \in \Pi \mid A \models b, \text{ for all } b \in B(r) \} \) of all rules whose body is satisfied under \( A \).

An assignment \( A_1 \) is smaller or equal to another assignment \( A_2 \) wrt. a program \( \Pi \), denoted \( A_1 \leq_\Pi A_2 \) iff \( \{ T a \in A_1^T \mid a \in A(\Pi) \} \subseteq \{ T a \in A_2^T \mid a \in A(\Pi) \} \).

Definition 3 (Answer Set). An answer set of \( \Pi \) is a \( \leq_\Pi \)-minimal (complete) model \( A \) of \( f_\Pi^A \).

Since interpretations (and thus answer sets, etc.) are complete assignments, slightly abusing notation, we adopt the usual convention to uniquely identify them with the set of all positive literals they contain.

Example 1. Consider the program \( \Pi = \{ p \leftarrow &id[p](), \} \), where \( &id[p]() \) is true iff \( p \) is true. Then \( \Pi \) has the answer set \( A_1 = \emptyset \), which is indeed a \( \leq_\Pi \)-minimal model of \( f_\Pi^{A_1} = \emptyset \).

The answer sets of a HEX-program \( \Pi \) are determined by the DLVHEX solver using a transformation to ordinary ASP programs as follows. Each external atom \( \&g[p](c) \) in \( \Pi \) is replaced by an ordinary ground external replacement atom \( e_{\&g[p]}(c) \) and a rule \( e_{\&g[p]}(c) \lor \neg e_{\&g[p]}(c) \leftarrow \) is added to the program. The answer sets of the resulting guessing program \( \Pi \) are determined by an ordinary ASP solver and projected to non-replacement atoms. However, the resulting interpretations are not necessarily models

\footnotetext{2 For simplicity, we do not formally introduce strong negation but view, as customary, classical literals \( \neg a \) as new atoms together with a constraint \( \leftarrow a, \neg a \).}
of $\Pi$, as the value of $\&g[p]$ under $f_{\&g}$ can be different from the one of $e_{\&g[p]}(c)$. Each answer set of $\hat{\Pi}$ is thus merely a candidate which must be checked against the external sources. If no discrepancy is found, the model candidate is a compatible set of $\Pi$. More precisely,

**Definition 4 (Compatible Set).** A compatible set of a program $\Pi$ is an interpretation $\hat{A}$ such that

(i) $A$ is an answer set [16] of the guessing program $\hat{\Pi}$, and

(ii) $f_{\&g}(\hat{A}, p, c) = 1$ iff $T e_{\&g[p]}(c) \in A$ for all external atoms $\&g[p](c)$ in $\Pi$, i.e. the guessed values coincides with the actual output under the input from $\hat{A}$.

The compatible sets of $\Pi$ include (modulo $A(\Pi)$) all (FLP) answer sets. For each answer set $A$ there is a compatible set $\hat{A}$ such that $A$ is the restriction of $\hat{A}$ to non-replacement atoms, but not vice versa. To filter out the compatible sets which are not answer sets, the current evaluation algorithm proceeds as follows. Each compatible set $A$ is fed to the minimality check, which is realized as a search for unfounded sets. This is justified by the following Definitions 5 and 6 and Theorem 1 from [7]. (These results lift unfounded set checking as a separate search problem using an encoding as a SAT instance. That is, for a program $\Pi$ and an interpretation $A$ we construct a set of nogoods $\Gamma^A_\Pi$ such that its solutions contain representations of all unfounded sets of $\Pi$ wrt. $A$. A (relatively simple) post-check finds the unfounded sets among the solutions of $\Gamma^A_\Pi$.)

**Definition 5 (Unfounded Set [7]).** Given a program $\Pi$ and an interpretation $A$, let $X$ be any set of ordinary ground atoms appearing in $\Pi$. Then, $X$ is an unfounded set for $A$ iff, for each rule $r$ having some atoms from $X$ in the head, at least one of the following conditions holds, where $A \cup \neg X = (A \setminus \{Ta \mid a \in X\}) \cup \{Fa \mid a \in X\}$:

(i) some literal of $B(r)$ is false wrt. $A$,

(ii) some literal of $B(r)$ is false wrt. $A \cup \neg X$, or

(iii) some atom of $H(r) \setminus X$ is true wrt. $A$.

**Definition 6 (Unfounded-free Interpretations [7]).** An interpretation $A$ of a program $\Pi$ is unfounded-free iff $A^T \cap X = \emptyset$, for all unfounded sets $X$ of $\Pi$ wrt. $A$.

**Theorem 1 (Characterization of Answer Sets [7]).** A model $A$ of a program $\Pi$ is an answer set iff it is unfounded-free.

**Example 2 (cont’d).** Reconsider the program $\Pi = \{ p \leftarrow &id[p]() \}$ from above. Then the corresponding guessing program is $\hat{\Pi} = \{ p \leftarrow e_{&id[p]}(); e_{&id[p]} \lor ne_{&id[p]} \leftarrow \}$ and has the answer sets $A_1 = \emptyset$ and $A_2 = \{Tp, Te_{&id[p]}\}$. While $A_1$ does not intersect with any unfounded sets and is thus also a $\leq_{IT}$-minimal model of $f^{\Pi A_1} = \emptyset$, $A_2$ intersects with the unfounded set $U = \{p\}$ and is not an answer set.

Our HEX implementation DLVHEX realizes the search for unfounded sets as a separate search problem using an encoding as a SAT instance. That is, for a program $\Pi$ and an interpretation $A$ we construct a set of nogoods $\Gamma^A_\Pi$ such that its solutions contain representations of all unfounded sets of $\Pi$ wrt. $A$. A (relatively simple) post-check finds the unfounded sets among the solutions of $\Gamma^A_\Pi$.

### 3 Deciding the Necessity of the UFS Check

An alternative to the search for unfounded sets is an explicit construction of the reduct and a search for smaller models. However, it turned out that the minimality check based
on unfounded sets is more efficient. Nevertheless the computational costs are still high. Moreover, during evaluation of $\Pi$ for computing the compatible set $A$, the ordinary ASP solver has already made an unfounded set check, and we can safely assume that it is founded from its perspective. Hence, all remaining unfounded sets which were not discovered by the ordinary ASP solver have to involve external sources, as their behavior is not fully captured by the ASP solver.

In this section we formalize these ideas and define a decision criterion which allows us to decide whether a further UFS check is necessary for a given program. We eventually define a class of programs which does not require an additional unfounded set check. Intuitively, we show that every unfounded set that is not already detected during the construction of $A$ contains input atoms of external atoms which are involved in cycles. If no such input atom exists in the program, then the UFS check is superfluous.

Let us therefore start with a definition of atom dependency.

**Definition 7 (Atom Dependency).** For a ground program $\Pi$, and ground atoms $p(c)$ and $q(d)$, we say that

(i) $p(c)$ depends on $q(d)$, denoted $p(c) \rightarrow q(d)$, iff for some rule $r \in \Pi$ we have $p(c) \in H(r)$ and $q(d) \in B^+(r)$;

(ii) $p(c)$ depends externally on $q(d)$, denoted $p(c) \rightarrow_e q(d)$, iff for some rule $r \in \Pi$ we have $p(c) \in H(r)$ and there is a $\&q[q_1, \ldots, q_n](e) \in B^+(r) \cup B^-(r)$ with $q_i = q$ for some $1 \leq i \leq n$. 

In the following, we consider dependency graphs $G^R_\Pi$ for a ground program $\Pi$, where the set of vertices is the set of all ground atoms, and the set of edges is given by a binary relation $R$ over ground atoms. If $R$ is not explicitly mentioned, then it is assumed to consist of $\rightarrow \cup \rightarrow_e$, whose elements are also called ordinary edges and $e$-edges, respectively.

The next definition and lemma allow to restrict our attention to the “core” of an unfounded set, i.e., its most essential part. For our purpose, we can then focus on such cores, disregarding atoms in a cut which is defined as follows.

**Definition 8 (Cut).** Let $U$ be an unfounded set of $\Pi$ wrt. $A$. A set of atoms $C \subseteq U$ is called a cut, iff

(i) $b \not\rightarrow_e a$, for all $a \in C$ and $b \in U$ ($C$ has no incoming or internal $e$-edges), and

(ii) $b \not\rightarrow a$ and $a \not\rightarrow b$, for all $a \in C$ and $b \in U \setminus C$ (there are no ordinary edges between $C$ and $U \setminus C$).

**Example 3.** Consider the program $\Pi$ given as the following set of rules

\[
\begin{align*}
r & \leftarrow \&id[r]() \\
p & \leftarrow \&id[r]() \\
p & \leftarrow q \\
q & \leftarrow p
\end{align*}
\]

We have $p \rightarrow q, q \rightarrow p, r \rightarrow_e r$ and $p \rightarrow_e r$. Program $\Pi$ has the unfounded set $U = \{p, q, r\}$ wrt. $A = \{Tp, Tq, Tr\}$. Observe that $C = \{p, q\}$ is a cut, and therefore we have that $U \setminus C = \{r\}$ is an unfounded set of $\Pi$ wrt. $A$.

We first prove that cuts can be removed from unfounded sets and the resulting set is still an unfounded set.
Lemma 1 (Unfounded Set Reduction Lemma). Let \( U \) be an unfounded set of \( \Pi \) wrt. \( A \), and let \( C \) be a cut. Then, \( Y = U \setminus C \) is an unfounded set of \( \Pi \) wrt. \( A \).

Proof (Sketch). If \( Y = \emptyset \), then the result holds trivially. Otherwise, let \( r \in \Pi \) with \( H(r) \cap Y \neq \emptyset \). We show that one of the conditions in Definition 5 holds. Observe that \( H(r) \cap U \neq \emptyset \) because \( U \supseteq Y \). Since \( U \) is an unfounded set of \( \Pi \) wrt. \( A \), either

(i) \( A \not\models b \) for some \( b \in B(r) \); or
(ii) \( A \cup \neg U \not\models b \) for some \( b \in B(r) \); or
(iii) \( A \models h \) for some \( h \in H(r) \setminus U \)

If (i), then the condition also holds wrt. \( Y \).

If (ii), let \( a \in H(r) \) such that \( a \in Y \), and \( b \in B(r) \) such that \( A \cup \neg U \not\models b \). We make a case distinction: either \( b \) is an ordinary literal or an external one.

If it is an ordinary default-negated atom not \( c \), then \( A \cup \neg U \not\models b \) implies \( \neg c \in A \) and \( c \not\in U \), and therefore also \( A \cup \neg Y \not\models b \). So assume \( b \) is an ordinary atom. If \( b \not\in U \) then \( A \not\models b \) and case (i) applies, so assume \( b \in U \). Because \( a \in H(r) \) and \( b \in B(r) \), we have \( a \rightarrow b \) and therefore either \( a, b \in C \) or \( a, b \in Y \) (because there are no ordinary edges between \( C \) and \( Y \)). But by assumption \( a \in Y \), and therefore \( b \in Y \), hence \( A \cup \neg Y \not\models b \).

If \( b \) is an external literal, then there is no \( q \in U \) with \( a \rightarrow_e q \) and \( q \not\in Y \). Otherwise, this would imply \( q \in C \) and \( C \) would have an incoming e-edge, which contradicts the assumption that \( C \) is a cut. Hence, for all \( q \in U \) with \( a \rightarrow_e q \), also \( q \in Y \), and therefore the truth value of \( b \) under \( A \cup \neg U \) and \( A \cup \neg Y \) is the same. Hence \( A \cup \neg Y \not\models b \).

If (iii), then also \( A \models h \) for some \( h \in H(r) \setminus Y \) because \( Y \subseteq U \) and therefore \( H(r) \setminus Y \supseteq H(r) \setminus U \).

Next we prove, intuitively, that for each unfounded set \( U \) of \( \Pi \), either the input to some external atom is unfounded itself, or \( U \) is already detected when \( \tilde{\Pi} \) is evaluated.

Lemma 2 (EA-Input Unfoundedness). Let \( U \) be an unfounded set of \( \Pi \) wrt. \( A \). If there are no \( x, y \in U \) such that \( x \rightarrow_e y \), then \( U \) is an unfounded set of \( \tilde{\Pi} \) wrt. \( \tilde{A} \).

Proof (Sketch). If \( U = \emptyset \), then the result holds trivially. Otherwise, let \( \tilde{r} \in \tilde{\Pi} \) such that \( H(\tilde{r}) \cap U \neq \emptyset \). Let \( a \in H(\tilde{r}) \cap U \). Observe that \( \tilde{r} \) cannot be an external atom guessing rule because \( U \) contains only ordinary atoms. We show that one of the conditions in Definition 5 holds for \( \tilde{r} \) wrt. \( \tilde{A} \).

Because \( \tilde{r} \) is no external atom guessing rule, there is a corresponding rule \( r \in \Pi \) containing external atoms in place of replacement atoms. Because \( U \) is an unfounded set of \( \Pi \) and \( H(r) = H(\tilde{r}) \), either:

(i) \( A \not\models b \) for some \( b \in B(r) \); or
(ii) \( A \cup \neg U \not\models b \) for some \( b \in B(r) \); or
(iii) \( A \models h \) for some \( h \in H(r) \setminus U \)

If (i), let \( b \in B(r) \) such that \( A \not\models b \) and \( b \) the corresponding literal in \( B(\tilde{b}) \) (which is the same if \( b \) is ordinary and the corresponding replacement literal if \( b \) is external). Then also \( A \not\models \tilde{b} \) because \( A \) is compatible.

For (ii), we make a case distinction: either \( b \) is ordinary or external.

If \( b \) is ordinary, then \( b \in B(\tilde{r}) \) and \( A \cup \neg U \not\models b \) holds because \( A \) and \( \tilde{A} \) are equivalent for ordinary atoms.
If $b$ is an external atom or default-negated external atom, then no atom $p(c) \in U$ is input to it, i.e. $p$ is not a predicate input parameter of $b$; otherwise we had $a \rightarrow_e p(c)$, contradicting our assumption that $U$ has no internal e-edges. But then $A \cup \neg U$ implies $A \not= b$ because the truth value of $b$ under $A \cup \neg U$ and $A$ is the same. Therefore we can apply case (i).

If (iii), then also $\hat{A} \models h$ for some $h \in H(\hat{r}) \setminus U$ because $H(r) = H(\hat{r})$ contains only ordinary atoms and $A$ is equivalent to $\hat{A}$ for ordinary atoms. $\square$

**Example 4.** Reconsider the program $\Pi$ from Example 3. Then the unfounded set $U' = \{p, q\}$ wrt. $A' = \{Tp, Tq, Fr\}$ is already detected when $\hat{\Pi}$ consisting of $e_{\&id[\hat{r}]}() \leftarrow$

\[
\begin{align*}
 r & \leftarrow e_{\&id[\hat{r}]}() \\
p & \leftarrow e_{\&id[\hat{r}]}() \\
p & \leftarrow q \\
q & \leftarrow p
\end{align*}
\]

is evaluated by the ordinary ASP solver because $p \not\rightarrow_e q$ and $q \not\rightarrow_e p$. In contrast, the unfounded set $U'' = \{p, q, r\}$ wrt. $A'' = \{Tp, Tq, Tr\}$ is not detected by the ordinary ASP solver because $p, r \in U''$ and $p \rightarrow_e r$.

The essential property of unfounded sets of $\Pi$ wrt. $A$ that are not recognized during the evaluation of $\hat{\Pi}$, is the existence of cyclic dependencies including input atoms of some external atom. Towards a formal characterization of a class of programs without this property, i.e., that do not require additional UFS checks, we define cycles as follows.

**Definition 9 (Cycle).** A cycle under a binary relation $\circ$ is a sequence of elements $C = c_0, c_1, \ldots, c_n, c_{n+1}$ with $n \geq 0$, such that $(c_i, c_{i+1}) \in \circ$ for all $0 \leq i \leq n$ and $c_0 = c_{n+1}$. We say that a set $S$ contains a cycle under $\circ$, if there is a cycle $C = c_0, c_1, \ldots, c_n, c_{n+1}$ under $\circ$ such that $c_i \in S$ for all $0 \leq i \leq n + 1$.

The following proposition states, intuitively, that each unfounded set $U$ of $\Pi$ wrt. $A$ which contains no cycle through the input atoms to some external atom has a corresponding unfounded set $U'$ of $\Pi$ wrt. $A$. That is, the unfoundedness is already detected when $\Pi$ is evaluated.

Let $\rightarrow^d = \rightarrow \cup \leftarrow \cup \rightarrow_e$, where $\leftarrow$ is the inverse of $\rightarrow$, i.e. $\leftarrow = \{(x, y) \mid (y, x) \in \rightarrow\}$. A cycle $c_0, c_1, \ldots, c_n, c_{n+1}$ under $\rightarrow^d$ is called an e-cycle, iff it contains e-edges, i.e., iff $(c_i, c_{i+1}) \in \rightarrow_e$ for some $0 \leq i \leq n$.

**Proposition 1 (Relevance of e-cycles).** Let $U \not= \emptyset$ be an unfounded set of $\Pi$ wrt. $A$ that does not contain any e-cycle under $\rightarrow^d$. Then, there exists a nonempty unfounded set of $\Pi$ wrt. $A$.

**Proof (Sketch).** We define the reachable set $R(a)$ from some atom $a$ as $R(a) = \{b \mid (a, b) \in \rightarrow \cup \leftarrow^*\}$, i.e. the set of atoms $b \in U$ reachable from $a$ using edges from $\rightarrow \cup \leftarrow$ only but no e-edges. We first assume that $U$ contains at least one e-edge, i.e. there are $x, y \in U$ such that $x \rightarrow_e y$. Now we show that there is a $u \in U$ with outgoing e-edge (i.e. $u \rightarrow_e v$
for some \( v \in U \), but such that \( R(u) \) has no incoming e-edges (i.e. for all \( v \in R(u) \) and \( b \in U \), \( b \not\rightarrow_e v \) holds). Suppose to the contrary that for all \( a \) with outgoing e-edges, the reachable set \( R(a) \) has an incoming e-edge. We now construct an e-cycle under \( \rightarrow^d \), which contradicts our assumption. Start with an arbitrary node with an outgoing e-edge \( c_0 \in U \) and let \( p_0 \) be the (possibly empty) path (under \( \rightarrow \cup \leftarrow \) ) from \( c_0 \) to the node \( d_0 \in R(c_0) \) such that \( d_0 \) has an incoming e-edge, i.e. there is a \( c_1 \) such that \( c_1 \rightarrow_e d_0 \); note that \( c_1 \not\in R(c_0)^3 \). By assumption, also some node \( d_1 \) in \( R(c_1) \) has an incoming e-edge (from some node \( c_2 \not\in R(c_1) \)). Let \( p_1 \) be the path from \( c_1 \) to \( d_1 \), etc. By iteration we can construct the concatenation of the paths \( p_0, (d_0, c_1), p_1, (d_1, c_2), p_2, \ldots, p_i, (d_i, c_{i+1}), \ldots \), where the \( p_i \) from \( c_i \) to \( d_i \) are the paths within reachable sets, and the \((d_i, c_{i+1})\) are the e-edges between reachable sets. However, as \( U \) is finite some nodes on this path must be equal, i.e., a prefix of the constructed sequence represents an e-cycle (in reverse order).

This proves that \( u \) is a node with outgoing e-edge but such that \( R(u) \) has no incoming e-edges. We next show that \( R(u) \) is a cut. Condition (i) is immediately satisfied by definition of \( u \). Condition (ii) is shown as follows. Let \( u' \in R(u) \) and \( v' \in U \setminus R(u) \). We have to show that \( u' \not\rightarrow v' \) and \( v' \not\rightarrow u' \). Suppose, towards a contradiction, that \( u' \rightarrow v' \). Because of \( u' \in R(u) \), there is a path from \( u \) to \( u' \) under \( \rightarrow \cup \leftarrow \). But if \( u' \rightarrow v' \), then there would also be a path from \( u \) to \( v' \) under \( \rightarrow \cup \leftarrow \) and \( v' \) would be in \( R(u) \), a contradiction. Analogously, \( v' \rightarrow u' \) would also imply that there is a path from \( u \) to \( v' \) because there is a path from \( u \) to \( u' \), again a contradiction.

Therefore, \( R(u) \) is a cut of \( U \), and by Lemma 1, it follows that \( U \setminus R(u) \) is an unfounded set. Observe that \( U \setminus R(u) \) contains one e-edge less than \( U \) because \( u \) has an outgoing e-edge. Further observe that \( U \setminus R(u) \neq \emptyset \) because there is a \( w \in U \) such that \( u \rightarrow_e w \) but \( w \not\in R(u) \). By iterating this argument, the number of e-edges in the unfounded set can be reduced to zero in a nonempty core. Eventually, Lemma 2 applies, proving that the remaining set is an unfounded set of \( \hat{\Pi} \).

**Corollary 1.** If there is no e-cycle under \( \rightarrow^d \) and \( \hat{\Pi} \) has no unfounded set wrt. \( \hat{A} \), then \( A \) is unfounded-free for \( \Pi \).

**Proof (Sketch).** Suppose there is an unfounded set \( U \) of \( \Pi \) wrt. \( A \). Then it contains no e-cycle because there is no e-cycle under \( \rightarrow^d \). Then by Proposition 1 there is an unfounded set of \( \hat{\Pi} \) wrt. \( \hat{A} \), which contradicts our assumption.

This corollary can be used as follows to increase performance of an evaluation algorithm: if there is no cycle under \( \rightarrow^d \) containing e-edges, then an explicit unfounded set check is not necessary because the unfounded set check made during evaluation of \( \hat{\Pi} \) suffices. Note that this test can be done efficiently (in fact in linear time, similar to deciding stratifiability of an ordinary logic program). Moreover, in practice one can abstract from \( \rightarrow^d \) by using analogous relations on the level of predicate symbols instead of atoms. Clearly, if there is no e-cycle in the predicate dependency graph, then there can also be no e-cycle in the atom dependency graph. Hence, the predicate dependency graph can be used to decide whether the unfounded set check can be skipped.

---

3 Whenever \( x \rightarrow_e y \) for \( x, y \in U \), then there is no path from \( x \) to \( y \) under \( \rightarrow \cup \leftarrow \), because otherwise we would have an e-cycle under \( \rightarrow^d \).
Example 5. All example programs considered until here require an UFS check, but the
program $\Pi = \{ \text{out}(X) \leftarrow \text{diff}(\text{set}_1, \text{set}_2)(X) \} \cup F$ does not for any set of facts $F$, because there is no e-cycle under $\rightarrow^d$, where $\text{diff}$ computes the set difference of the extensions of $\text{set}_1$ and $\text{set}_2$.

Also $\Pi = \{ \text{str}(Z) \leftarrow \text{dom}(Z), \text{str}(X), \text{str}(Y), \text{not} \ \&\text{concat}(X, Y)(Z) \}$ (where $\&\text{concat}$ takes two constants and computes their string concatenation) does not need such a check; there is a cycle over an external atom, but no e-cycle under $\rightarrow^d$.

Moreover, the following proposition states that, intuitively, if $\hat{\Pi}$ has no unfounded
sets wrt. $A$, then any unfounded set $U$ of $\Pi$ wrt. $A$ must contain an atom which is
involved in a cycle under $\rightarrow^d$ that has an e-edge.

Definition 10 (Cyclic Input Atoms). For a program $\Pi$, an atom $a$ is a cyclic input
atom iff there is an atom $b$ such that $b \rightarrow_e a$ and there is a path from $a$ to $b$ under $\rightarrow^d$.

Let $\text{CA}(\Pi)$ denote the set of all cyclic input atoms of program $\Pi$.

Proposition 2 (Unfoundedness of Cyclic Input Atom). Let $U \neq \emptyset$ be an unfounded
set of $\Pi$ wrt. $A$ such that $U$ does not contain cyclic input atoms. Then, $\hat{\Pi}$ has a nonempty
unfounded set wrt. $A$.

Proof (Sketch). If $U$ contains no cyclic input atoms, then all cycles under $\rightarrow^d$ containing
e-edges in the atom dependency graph of $\Pi$ are broken, i.e. $U$ does not contain an
e-cycle under $\rightarrow^d$. Then by Proposition 1 there is an unfounded set of $\hat{\Pi}$ wrt. $A$. \hfill $\Box$

Proposition 2 allows for generating the additional nogood $\{ \exists a \mid a \in \text{CA}(\Pi) \}$ and
adding it to $I^A_{\hat{\Pi}}$. Again, considering predicate symbols instead of atoms is possible to
reduce the overhead introduced by the dependency graph.

4 Program Decomposition

It turns out that the usefulness of the decision criterion can be increased by decomposing
the program into components, such that the criterion can be applied component-wise. This allows for restricting the unfounded set check to components with e-cycles, whereas
e-cycle-free components can be ignored in the check.

Let $C$ be a partitioning of the ordinary atoms $A(\Pi)$ of $\Pi$ into subset-maximal
strongly connected components under $\rightarrow \cup \rightarrow_e$. We define for each partition $C \in \mathcal{C}$ the
subprogram $\Pi_C$ associated with $C$ as $\Pi_C = \{ r \in \Pi \mid H(r) \cap C \neq \emptyset \}$.

We next show that if a program has an unfounded set $U$ wrt. $A$, then $U \cap C$ is an
unfounded set wrt. $A$ for the subprogram of some strongly connected component $C$.

Proposition 3. Let $U \neq \emptyset$ be an unfounded set of $\Pi$ wrt. $A$. Then, for some $\Pi_C$ with $C \in \mathcal{C}$ it holds that $U \cap C$ is a nonempty unfounded set of $\Pi_C$ wrt. $A$.

Proof (Sketch). Let $U$ be a nonempty unfounded set of $\Pi$ wrt. $A$. Because $C$ is a
decomposition of $A(\Pi)$ into strongly connected components, the component dependency
graph

$$\langle \mathcal{C}, \{ (C_1, C_2) \mid C_1, C_2 \in \mathcal{C}, \exists a_1 \in C_1, a_2 \in C_2 : (a_1, a_2) \in \rightarrow \cup \rightarrow_e \} \rangle$$

T. Eiter et al.
is acyclic. Following the hierarchical component dependency graph from the nodes without predecessor components downwards, we can find a “first” component which has a nonempty intersection with \( U \), i.e., there exists a component \( C \in \mathcal{C} \) such that \( C \cap U \neq \emptyset \) but \( C' \cap U = \emptyset \) for all transitive predecessor components \( C' \) of \( C \).

We show that \( U \cap C \) is an unfounded set of \( \Pi_C \) wrt. \( A \). Let \( r \in \Pi_C \) be a rule such that \( H(r) \cap (U \cap C) \neq \emptyset \). We have to show that one of the conditions of Definition 5 holds for \( r \) wrt. \( A \) and \( U \cap C \).

Because \( U \) is an unfounded set of \( \Pi \) wrt. \( A \) and \( H(r) \cap (U \cap C) \neq \emptyset \) implies \( H(r) \cap U \neq \emptyset \), we know that one of the conditions holds for \( r \) wrt. \( A \) and \( U \). If this is condition (i) or (iii), then it trivially holds also wrt. \( A \) and \( U \cap C \) because these conditions depend only on the assignment \( A \), but not on the unfounded set \( U \).

If it is condition (ii), then \( A \cup \neg U \not\models b \) for some (ordinary or external) body literal \( b \in B(r) \). We show next that the truth value of all literals in \( B(r) \) is the same under \( A \cup \neg U \) and \( A \cup \neg (U \cap C) \), which proves that condition (ii) holds also wrt. \( A \) and \( U \cap C \).

If \( b = \not a \) for some atom \( a \), then \( Ta \in A \) and \( a \notin U \) and consequently \( a \notin U \cap C \), hence \( A \cup \neg (U \cap C) \not\models b \). If \( b \) is an ordinary atom, then either \( Fb \in A \), which implies immediately that \( A \cup \neg (U \cap C) \not\models b \), or \( b \in U \). But in the latter case \( b \) is either in a predecessor component \( C' \) of \( C \) or in \( C \) itself (since \( h \rightarrow b \) for all \( h \in H(r) \)). But since \( U \cap C' = \emptyset \) for all predecessor components of \( C \), we know \( b \in C \) and therefore \( b \in (U \cap C) \), which implies \( A \cup \neg (U \cap C) \not\models b \).

If \( b \) is a positive or default-negated external atom, then all input atoms \( a \) to \( b \) are in a predecessor component \( C' \) of \( C \) or in \( C \) itself (since \( h \rightarrow a \) for all \( h \in H(r) \)). We show with a similar argument as before that the truth value of each input atom \( a \) is the same under \( A \cup \neg U \) and \( A \cup \neg (U \cap C) \); if \( A \cup \neg U \models a \), then \( Ta \in A \) and \( a \notin U \), hence \( a \notin (U \cap C) \) and therefore \( A \cup \neg (U \cap C) \models a \). If \( A \cup \neg U \not\models a \), then either \( Fa \in A \), which immediately implies \( A \cup \neg (U \cap C) \not\models a \), or \( a \notin U \). But in the latter case \( a \) must be in \( C \) because \( U \cap C' = \emptyset \) for all predecessor components \( C' \) of \( C \). Therefore \( a \in (U \cap C) \) and consequently \( A \cup \neg (U \cap C) \not\models a \). Because all input atoms \( a \) have the same truth value under \( A \cup \neg U \) and \( A \cup \neg (U \cap C) \), the same holds also for the positive or default-negated external atom \( b \) itself.

This proposition states that a search for unfounded sets can be done independently for the subprograms \( \Pi_C \) for all \( C \in \mathcal{C} \). If there exists a global unfounded set, then there exists also one in at least one of the program components. However, we know by Corollary 1 that programs \( \Pi \) without e-cycles cannot contain unfounded sets, which are not already detected when \( \Pi \) is solved. If we apply this proposition to the subprograms \( \Pi_C \), we can safely ignore e-cycle-free program components.

Example 6. Reconsider the program \( \Pi \) from Example 3. Then \( \mathcal{C} \) contains the components \( C_1 = \{ p, q \} \) and \( C_2 = \{ r \} \) and we have \( \Pi_{C_1} = \{ p \leftarrow \&id[p]; p \leftarrow q; q \leftarrow p \} \) and \( \Pi_{C_2} = \{ r \leftarrow \&id[r] \} \). By Proposition 3, each unfounded set of \( \Pi \) wrt. some assignment can also be detected over one of the components. Consider e.g. \( U = \{ p, q, r \} \) wrt. \( A = \{ Tp, Tq, Tr \} \). Then \( U \cap \{ r \} = \{ r \} \) is also an unfounded set of \( \Pi_{C_2} \) wrt. \( A \).

By separate application of Corollary 1 to the components, we can conclude that there can be no unfounded sets over \( \Pi_{C_2} \) that are not already detected when \( \Pi \) is evaluated (because it has no e-cycles). Hence, the additional unfounded set check is only necessary
for $\Pi_{C_2}$. Indeed, the only unfounded set which is not detected when $\tilde{H}$ is evaluated is $\{r\}$ of $\Pi_{C_2}$ wrt. any interpretation $A \supseteq \{Tr\}$.

Finally, one can also show that splitting, i.e., the component-wise check for foundedness, does not lead to spurious unfounded sets.

**Proposition 4.** If $U$ is an unfounded set of $\Pi_{C}$ wrt. $A$ such that $U \subseteq C$, then $U$ is an unfounded set of $\Pi$ wrt. $A$.

**Proof (Sketch).** If $U = \emptyset$, then the result holds trivially. By definition of $\Pi_{C}$ we have $H(r) \cap C = \emptyset$ for all $r \in \Pi \setminus \Pi_{C}$. By precondition of the proposition we have $U \subseteq C$. But then $H(r) \cap U = \emptyset$ for all $r \in \Pi \setminus \Pi_{C}$ and $U$ is an unfounded set of $\Pi$ wrt. $A$. □

## 5 Implementation and Evaluation

For implementing our technique, we integrated CLASP into our prototype system DLVHEX; we use CLASP as an ASP solver for computing compatible sets and as a SAT solver for solving the nogood set of the UFS check. We evaluated the implementation on a Linux server with two 12-core AMD 6176 SE CPUs with 128GB RAM.

**Argumentation Benchmarks.** In this benchmark we compute ideal set extensions for randomized instances of abstract argumentation frameworks [4] of different sizes. In these instances, the cycles involve usually only small parts of the overall programs, hence the program decomposition is very effective. Table 1 shows results of our experimental evaluation on argumentation benchmark instances; for computing average times, we considered 300 seconds for instances that timed out. The encodings contain a cyclic part with cycles over external atoms, and a cyclic part with cycles that do not contain external atoms. Therefore in these instances our new approach can help in limiting the set of atoms for which unfounded sets must be checked, which explains the significant performance gain due to less time spent in the UFS check.

**Multi-Context System Benchmarks.** MCSs [1] are a formalism for interlinking knowledge based systems; in [9], *inconsistency explanations (IEs)* for an MCS were defined.

<table>
<thead>
<tr>
<th></th>
<th>first answer set</th>
<th>all answer sets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>standard approach</td>
<td>new approach</td>
</tr>
<tr>
<td>timeouts avg</td>
<td>timeouts avg</td>
<td>gain</td>
</tr>
<tr>
<td>5</td>
<td>0 1.09</td>
<td>0 1.07</td>
</tr>
<tr>
<td>6</td>
<td>0 2.40</td>
<td>0 2.30</td>
</tr>
<tr>
<td>7</td>
<td>0 5.58</td>
<td>0 5.33</td>
</tr>
<tr>
<td>8</td>
<td>0 14.26</td>
<td>0 12.74</td>
</tr>
<tr>
<td>9</td>
<td>0 39.82</td>
<td>0 33.57</td>
</tr>
<tr>
<td>10</td>
<td>2 126.54</td>
<td>0 80.00</td>
</tr>
</tbody>
</table>

Table 1: Argumentation Benchmarks: standard approach means the state-of-the-art approach without decomposition of the UFS check and without elimination of unnecessary checks, times are in seconds, timeout was 300 sec, for each system size there were 50 instances.
This benchmark computes the IEs, which correspond 1-1 to answer sets of an encoding rich in cycles through external atoms (which evaluate local knowledge base semantics). We use random instances of different topologies created with an available benchmark generator. For the MCS benchmarks we tested 68 consistent and 88 inconsistent MCSs for which we compute inconsistency explanations [9]. This encoding contains saturation over external atoms, where nearly all cycles in the HEX-program contain at least one external atom. Therefore the methods we introduce in this work can only very rarely reduce the set of atoms for which the UFS check needs to be performed.

The benchmark result for MCS instances confirms that the syntactic check we introduce in this paper is very cheap and does not impede performance, even if an instance does not admit a considerable simplification for the UFS check: over all 156 instances, we had an overall runtime of 25357 seconds with the standard approach, and a runtime of 25115 seconds with our new approach; the gain is 242 seconds which is less than one percent speedup (for enumerating all inconsistency explanations) by applying our method. This is a very small gain, and there is no difference in the number of instances that timed out.

**Default Reasoning over Description Logics Benchmarks.** Another application of HEX-programs is the DL-plugin [10], which integrates description logics ontologies with rules. This allows, for instance, default reasoning over description logic knowledge bases, which is not possible in DL knowledge bases alone. Defaults require cyclic dependencies over external atoms. However, as all such dependencies involve default negated atoms, we have no cycles according to Definition 7, which respects only positive dependencies. Hence, the decision criterion comes to the conclusion that no UFS check is required.

We used variants of the benchmarks presented in [6], which query wines from an ontology and classify them as red or white wines, where a wine is assumed to be white unless the ontology explicitly entails the contrary. In this scenario, the decision criterion eliminates all unfounded set checks. However, as there is only one compatible set per instance, there would be only one unfounded set check anyway, hence the speedup due to the decision criterion is not significant. But the effect of the decision criterion can be increased by slightly modifying the scenario such that there are multiple compatible sets. This can be done, for instance, by nondeterministic default classifications, e.g., if a wine is not Italian, then it is either French or Spanish by default. Our experiments have shown that with a small number of compatible sets, the performance enhancement due to the decision criterion is marginal, but increases with larger numbers of compatible sets. For instance, for 243 compatible sets (and thus 243 unfounded set checks) we could observe a speedup from 13.59 to 12.19 seconds.

**6 Conclusion**

The evaluation of HEX-programs requires a minimality check of model candidates which is realized as an equivalent search for unfounded sets (UFS). However, this check is computationally costly. Moreover, during construction of the model candidate, the ASP solver used as a backend has already performed a “restricted” form of unfounded set check, i.e., an UFS check over the program $\hat{\Pi}$, viewing external atoms as ordinary ones. Hence, it already excludes certain unfounded candidates. Redoing a complete UFS
search is thus a waste of resources, and the goal is to minimize the number of additional foundedness checks.

In this paper we presented a syntactic criterion which can be efficiently tested and allows to decide whether an additional UFS check is necessary for a given program. It turned out that the essential property is the existence of cyclic dependencies of atoms which involve predicate inputs to external atoms. If no such dependencies exist, then there is no need for an additional check, and the check built into the ordinary ASP solver is already sufficient. In further elaboration, we have refined the basic idea by splitting the input program into components. This allows for independent applications of the decision criterion to the different components. Thus, the UFS check is restricted to relevant parts of the program, while it can safely be ignored for other parts.

Related to our work is [3], where a similar program decomposition is used, yet for ordinary programs only. While we consider e-cycles, which are specific for HEX-programs, the interest in [3] is with head-cycles with respect to disjunctive rule heads. In fact, our implementation may be regarded as an extension of the work in [3], since the evaluation of $\hat{H}$ follows their principles of performing UFS checks in case of head-cycles. Note however, that the applied component splitting does not generalize the well-known splitting theorem [18] as we consider only positive dependencies for ordinary atoms.

An interesting issue for further research is to consider refinements of the decision criterion, or alternative criteria. One direction for refinement is to dynamically take the model candidate into account, in addition to the program structure, which intuitively may prune dependencies and thus allow to skip the UFS check even in the presence of (syntactic) e-cycles. Another extension is to exploit additional semantic information on the external atoms, e.g., such as (anti-)monotonicity etc. Moreover, a more extensive experimental analysis is subject of our future work, where case studies may give rise to alternative criteria and further optimizations.

References

Backdoors to Normality for Disjunctive Logic Programs⋆

Johannes Klaus Fichte and Stefan Szeider
fichte@kr.tuwien.ac.at, stefan@szeider.net

Vienna University of Technology

Abstract.

Over the last two decades, propositional satisfiability (SAT) has become one of the most successful and widely applied techniques for the solution of NP-complete problems. The aim of this paper is to investigate theoretically how SAT can be utilized for the efficient solution of problems that are harder than NP or co-NP. In particular, we consider the fundamental reasoning problems in propositional disjunctive answer set programming (ASP), Brave Reasoning and Cautious Reasoning, which ask whether a given atom is contained in at least one or in all answer sets, respectively. Both problems are located at the second level of the Polynomial Hierarchy and thus assumed to be harder than NP or co-NP. We cannot transform these two reasoning problems to SAT in polynomial time, unless the Polynomial Hierarchy collapses.

We show that certain structural aspects of logic programs can be utilized to break through this complexity barrier, using new techniques from Parameterized Complexity. In particular, we exhibit transformations from Brave and Cautious Reasoning to SAT that run in time $O(2^k n^2)$ where $k$ is a structural parameter of the instance and $n$ the input size. In other words, the reduction is fixed-parameter tractable for parameter $k$. As the parameter $k$ we take the size of a smallest backdoor with respect to the class of normal (i.e., disjunction-free) programs. Such a backdoor is a set of atoms that when deleted makes the program normal. In consequence, the combinatorial explosion, which is expected when transforming a problem from the second level of the Polynomial Hierarchy to the first level, can now be confined to the parameter $k$, while the running time of the reduction is polynomial in the input size $n$, where the order of the polynomial is independent of $k$. We show that such a transformation is not possible if we consider backdoors with respect to tightness instead of normality.

1 Introduction

Over the last two decades, propositional satisfiability (SAT) has become one of the most successful and widely applied techniques for the solution of NP-complete problems. Today’s SAT-solvers are extremely efficient and robust, instances with hundreds of thousands of variables and clauses can be solved routinely. In fact, due to the success of SAT, NP-complete problems have lost their scariness, as in many cases one can efficiently encode NP-complete problems to SAT and solve them by means of a SAT-solver [21].

⋆ Research supported by the ERC, Grant COMPLEX REASON 239962.
We investigate transformations to SAT for problems that are harder than NP or co-NP. In particular, we consider various search problems that arise in disjunctive answer set programming (ASP). With ASP one can describe a problem by means of rules that form a disjunctive logic program, whose solutions are answer sets. Many important problems of AI and reasoning can be represented in terms of the search for answer sets [4, 34, 35]. Two of the most fundamental ASP problems are Brave Reasoning (is a certain atom contained in at least one answer set?) and Cautious Reasoning (is a certain atom contained in all answer sets?). Both problems are located at the second level of the Polynomial Hierarchy [10] and thus assumed to be harder than NP or co-NP. It would be desirable to utilize SAT-solvers for these problems. However, we cannot transform these two reasoning problems to SAT in polynomial time, unless the Polynomial Hierarchy collapses, which is believed to be unlikely.

**New Contribution** In this work we show how to utilize certain structural aspects of the ASP instances to transform the two ASP reasoning problems to SAT. In particular, we exhibit a transformation to SAT that runs in time $O(2^k n^2)$ where $k$ is a structural parameter of the instance and $n$ is the input size of the instance. Thus the combinatorial explosion, which is expected when transforming problems from the second level of the Polynomial Hierarchy to the first level, is confined to the parameter $k$, while the running time is polynomial in the input size $n$, and the order of the polynomial is independent of $k$. Such transformations are known as “fpt-transformations” and form the base of the completeness theory of Parameterized Complexity [8, 15].

It is known that the two reasoning problems, when restricted to so-called normal programs, drop to NP and co-NP [2, 32, 33], respectively. Hence it is natural to consider as a structural parameter $k$ the distance of a given program from being normal. We measure the distance in terms of the smallest number of atoms that need to be deleted to make the program normal. Following [38] we call such a set of deleted atoms a backdoor. We show that in time $O(2^k n^2)$ we can solve both of the following two tasks for a given program $P$ of input size $n$ and an atom $a^*$:

- **Backdoor Detection:** Find a backdoor of size at most $k$ of the given program $P$, or decide that a backdoor of size $k$ does not exist.

- **Backdoor Evaluation:** Transform the program $P$ into two propositional formulas $F_{Brave}(a^*)$ and $F_{Caut}(a^*)$ such that (i) $F_{Brave}(a^*)$ is satisfiable if and only if $a^*$ is in some answer set of $P$, and (ii) $F_{Caut}(a^*)$ is unsatisfiable if and only if $a^*$ is in all answer sets of $P$.

**Tightness** is a property of programs that, similar to normality, lets the complexities of Brave and Cautious Reasoning drop to NP and co-NP, respectively [6, 12]. Consequently, one could also consider backdoors to tightness. We show, however, that the reasoning problems already reach their full complexities (i.e., completeness for the second level of the Polynomial Hierarchy) with programs of distance one from being tight. Hence an fpt-transformation to SAT for programs of distance $k > 0$ from being tight is not possible unless the Polynomial Hierarchy collapses.


Related Work  Williams, Gomes, and Selman [38] introduced the notion of
backdoors to explain favorable running times and the heavy-tailed behavior of
SAT and CSP solvers on practical instances. The parameterized complexity of
finding small backdoors was initiated by Nishimura, Ragde, and Szeider [36]. For
further results regarding the parameterized complexity of problems related to
backdoors for SAT, we refer to a recent survey paper [16]. Fichte and Szeider [14]
formulated a backdoor approach for ASP problems, and obtained complexity
results with respect to the base class of Horn programs and various base classes
based on acyclicity; some results could be generalized [13].

Translations from ASP problems to SAT have been explored by several authors;
existing research mainly focused at transforming programs for which the reasoning
problems already belong to NP or co-NP. In particular, translations have been
considered for head cycle free programs [1], tight programs [12], and normal
programs [31, 24].

Some authors have generalized the above translations to capture programs for
which the reasoning problems are outside NP and co-NP. Janhunen et al. [23]
considered programs where the number of disjunctions in the heads of rules is
bounded. They provided a translation that allows a SAT encoding of the test
whether a candidate set of atoms is indeed an answer set of the input program.
Lee and Lifschitz [29] considered programs with a bounded number of cycles in
the positive dependency graph. They suggested a translation that, similar to
ours, transforms the input program into an exponentially larger propositional
formula whose satisfying assignments correspond to answer sets of the program.
As pointed out by Lifschitz and Razborov [30], this translation produces an
exponential blowup already for normal programs (we note that, in way of contrast,
our translation is in fact quadratic for normal programs).

Over the last few years, several SAT techniques have been integrated into
practical ASP solvers. In particular, solvers for normal programs (Cmodels [20],
ASSAT [31], Clasp [17]) use certain extensions of Clark’s completion and then
utilize either black box SAT solvers or integrate conflict analysis, backjumping,
and other techniques within the ASP context. ClaspD [9] is a disjunctive ASP-
solver utilizing nogoods that are based on the logical characterizations of loop
formulas [28]. It uses, among others, the SAT techniques backjumping and CDCL.

2 Preliminaries

Answer set programs  We consider a universe $U$ of propositional atoms. A
disjunctive logic program (or simply a program) $P$ is a set of rules of the form
$x_1 \lor \ldots \lor x_l \leftarrow y_1, \ldots, y_n, \neg z_1, \ldots, \neg z_m$ where $x_1, \ldots, x_l, y_1, \ldots, y_n, z_1, \ldots, z_m$
are atoms and $l, n, m$ are non-negative integers. We write $H(r) = \{x_1, \ldots, x_l\}$
(the head of $r$), $B^+(r) = \{y_1, \ldots, y_n\}$ (the positive body of $r$), and $B^-(r) =
\{z_1, \ldots, z_m\}$ (the negative body of $r$). We denote the sets of atoms occurring in a
rule $r$ or in a program $P$ by $\text{at}(r) = H(r) \cup B^+(r) \cup B^-(r)$ and $\text{at}(P) = \bigcup_{r \in P} \text{at}(r)$,
respectively. A rule $r$ is negation-free if $B^-(r) = \emptyset$, $r$ is normal if $|H(r)| \leq 1$,
$r$ is a constraint if $|H(r)| = 0$, $r$ is constraint-free if $|H(r) > 0|$, $r$ is Horn if it
is negation-free and normal, $r$ is positive if it is Horn and constraint-free, and
r is tautological if \( B^+(r) \cap (H(r) \cup B^-(r)) \neq \emptyset \). We say that a program has a certain property if all its rules have the property. We denote the class of all normal programs by \textbf{Normal}. In the following we restrict ourselves to programs that do not contain any tautological rules. This restriction is not significant as tautological rules can be omitted from a program without changing its answer sets [3].

A set \( M \) of atoms satisfies a rule \( r \) if \( (H(r) \cup B^-(r)) \cap M \neq \emptyset \) or \( B^+(r) \setminus M \neq \emptyset \). \( M \) is a model of \( P \) if it satisfies all rules of \( P \). The \textit{GL reduct} of a program \( P \) under a set \( M \) of atoms is the program \( P^M \) obtained from \( P \) by first removing all rules \( r \) with \( B^-(r) \cap M \neq \emptyset \) and second removing all \( \neg z \) where \( z \in B^-(r) \) from all remaining rules \( r \) [19]. \( M \) is an answer set (or stable set) of a program \( P \) if \( M \) is a minimal model of \( P^M \). The Emden-Kowalski operator of a program \( P \) and a subset \( A \) of atoms of \( P \) is the set \( T_P(A) := \{ a \mid a \in H(r), B^+(r) \subseteq A, \ r \in \ P \} \). The least model \( LM(P) \) is the least fixed point of \( T_P(A) \) [11]. Note that every positive program \( P \) has a unique minimal model which equals the least model \( LM(P) \) [18].

The main reasoning problems for ASP are \textbf{Brave Reasoning} (given a program \( P \) and an atom \( a \in at(P) \), is \( a \) contained in some answer set of \( P \)?) and \textbf{Cautious Reasoning} (given a program \( P \) and an atom \( a \in at(P) \), is \( a \) contained in all answer sets of \( P \)?). \textbf{Brave Reasoning} is \( \Sigma^P_2 \)-complete, \textbf{Cautious Reasoning} is \( \Pi^P_2 \)-complete [10].

\textbf{Example 1.} Consider the program \( P = \{ a \lor c \leftarrow b; \ b \leftarrow c, \neg g; \ c \leftarrow a; \ b \lor e \leftarrow c; \ h \lor i \leftarrow g, \neg c; \ a \lor b; \ g \leftarrow \neg i; \ e \}. \) The set \( A = \{ b, c, g \} \) is an answer set of \( P \) since \( P^A = \{ a \lor c \leftarrow b; \ c \leftarrow a; \ b \lor e \leftarrow c; \ a \lor b; \ g; \ e \} \) and the minimal models of \( P^A \) are \( \{ b, c, g \} \) and \( \{ a, c, g \} \). \( \square \)

\textbf{Parameterized Complexity} We give some basic background on parameterized complexity. For more detailed information we refer to other sources [8, 15].

A parameterized problem \( L \) is a subset of \( \Sigma^* \times \mathbb{N} \) for some finite alphabet \( \Sigma \). For an instance \( (I, k) \in \Sigma^* \times \mathbb{N} \) we call \( I \) the main part and \( k \) the parameter. \( L \) is fixed-parameter tractable if there exists a computable function \( f \) and a constant \( c \) such that there exists an algorithm that decides whether \( (I, k) \in L \) in time \( O(f(k)||I||^c) \) where \( ||I|| \) denotes the size of \( I \). Such an algorithm is called an \textit{fpt-algorithm}. FPT is the class of all fixed-parameter tractable decision problems.

Let \( L \subseteq \Sigma^* \times \mathbb{N} \) and \( L' \subseteq \Sigma^* \times \mathbb{N} \) be two parameterized problems for some finite alphabets \( \Sigma \) and \( \Sigma' \). An fpt-reduction \( r \) from \( L \) to \( L' \) is a many-to-one reduction from \( \Sigma^* \times \mathbb{N} \) to \( \Sigma'^* \times \mathbb{N} \) such that for all \( I \in \Sigma^* \) we have \( (I, k) \in L \) if and only if \( r(I, k) = (I', k') \in L' \) such that \( k' \leq g(k) \) for a fixed computable function \( g : \mathbb{N} \to \mathbb{N} \) and there is a computable function \( f \) and a constant \( c \) such that \( r \) is computable in time \( O(f(k)||I||^c) \) where \( ||I|| \) denotes the size of \( I \) [15]. Thus, an fpt-reduction is, in particular, an fpt-algorithm. It is easy to see that the class FPT is closed under fpt-reductions. We would like to note that the theory of fixed-parameter intractability is based on fpt-reductions [15].

\textbf{Propositional satisfiability} A truth assignment is a mapping \( \tau : X \to \{0, 1\} \) defined for a set \( X \subseteq U \) of atoms. For \( x \in X \) we put \( \tau(\neg x) = 1 - \tau(x) \). By \( ta(X) \)
we denote the set of all truth assignments $\tau : X \to \{0,1\}$. A truth assignment $\tau$ satisfies a propositional formula $\neg F_1$ if $\tau$ does not satisfy $F_1$, $\tau$ satisfies $F_1 \land F_2$ if $\tau$ satisfies $F_1$ and $F_2$, $\tau$ satisfies $F_1 \lor F_2$ if $\tau$ satisfies $F_1$ or $F_2$, for formulas $F_1$ and $F_2$, and $\tau$ always satisfies the propositional constant true and does not satisfy false. A formula $F$ is satisfiable if there is some truth assignment $\tau$ that satisfies $F$. We usually say variable instead of atom in the context of formulas. A propositional formula $F$ is in conjunctive normal form (CNF) if it is of the form $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} h_{i,j}$ where $n, m_1, \ldots, m_n \geq 1$ and $h_{i,j} \in \{ x, \neg x \mid x \in U \}$. The problem SAT asks, given a CNF formula $F$, whether $F$ is satisfiable. We can consider SAT as a parameterized problem by simply associating with every formula the parameter 0.

## 3 Backdoors of Programs

In the following we give the main notions concerning backdoors for answer set programming, as introduced by Fichte and Szeider [14]. Let $P$ be a program, $X$ a set of atoms, and $\tau \in \text{ta}(X)$. The truth assignment reduct of $P$ under $\tau$ is the logic program $P_{\tau}$ obtained from $P$ by removing all rules $r$ for which at least one of the following holds: (i) $H(r) \cap \tau^{-1}(1) \neq \emptyset$, (ii) $H(r) \subseteq X$, (iii) $B^+(r) \cap \tau^{-1}(0) \neq \emptyset$, and (iv) $B^-(r) \cap \tau^{-1}(1) \neq \emptyset$, and then removing from the heads and bodies of the remaining rules all literals $v, \neg v$ with $v \in X$. In the following, let $C$ be a class of programs. We call $C$ to be rule induced if for each $P \in C$, $P' \subseteq P$ implies $P' \in C$. A set $X$ of atoms is a strong $C$-backdoor of a program $P$ if $P_{\tau} \in C$ for all truth assignments $\tau \in \text{ta}(X)$. For a program $P$ and a set $X$ of atoms we define $P - X$ as the program obtained from $P$ by deleting all atoms contained in $X$ and their negations from the heads and bodies of all the rules of $P$. A set $X$ of atoms is a deletion $C$-backdoor of a program $P$ if $P - X \in C$ [14].

**Example 2.** Consider the program $P$ from Example 1. The set $X = \{ b, c, h \}$ is a strong Normal-backdoor since the truth assignment reducts $P_{b=0,c=0,h=0} = P_{000} = \{ i \leftarrow g; a; g \leftarrow \neg i \}$, $P_{b=1,c=0,h=0} = P_{010} = P_{011} = P_{101} = \{ a; g \leftarrow \neg i \}$, $P_{b=0,c=1,h=0} = P_{100} = \{ a; i \leftarrow g; g \leftarrow \neg i \}$, and $P_{b=1,c=1,h=0} = P_{110} = P_{111} = \{ g \leftarrow \neg i \}$ are in the class Normal. □

In the following we refer to $C$ as the target class of the backdoor. For most target classes $C$, deletion $C$-backdoors are strong $C$-backdoors. For $C = \text{Normal}$ even the opposite direction is true.

**Proposition 1** ([14]) If $C$ is rule induced, then every deletion $C$-backdoor is a strong $C$-backdoor.

**Lemma 1** Let $P$ be a program. A set $X$ is a strong Normal-backdoor of a program $P$ if and only if it is a deletion Normal-backdoor of $P$.

Because of space constraints we omit the proof of this lemma.
Each target class \( C \) gives rise to the following problems.

**Strong \( C \)-Backdoor-Asp-Check**

*Instance:* A (disjunctive logic) program \( P \), a strong \( C \)-backdoor \( X \) of \( P \), and a set \( M \subseteq \text{at}(P) \).

*Parameter:* The size of the backdoor \( k = |X| \).

*Question:* Is \( M \) an answer set of \( P \)?

**Strong \( C \)-Backdoor-Brave-Reasoning**

*Instance:* A (disjunctive logic) program \( P \), a strong \( C \)-backdoor \( X \) of \( P \), and an atom \( a^* \in \text{at}(P) \).

*Parameter:* The size of the backdoor \( k = |X| \).

*Question:* Does \( a^* \) belong to some answer set of \( P \)?

**Strong \( C \)-Backdoor-Skeptical-Reasoning**

*Instance:* A (disjunctive logic) program \( P \), a strong \( C \)-backdoor \( X \) of \( P \), and an atom \( a^* \in \text{at}(P) \).

*Parameter:* The size of the backdoor \( k = |X| \).

*Question:* Does \( a^* \) belong to all answer sets of \( P \)?

Problems for deletion \( C \)-backdoors can be defined similarly.

## 4 Using Backdoors

In this section, we show results regarding the use of backdoors with respect to the target class **Normal**.

**Theorem 1** The problem **Strong Normal-Backdoor-Asp-Check** is fixed-parameter tractable. More specifically, given a program \( P \) of input size \( n \), a strong **Normal**-backdoor \( N \) of \( P \) of size \( k \), and a set \( M \subseteq \text{at}(P) \) of atoms, we can check in time \( O(2^k n) \) whether \( M \) is an answer set of \( P \).

The most important part for establishing Theorem 1 is to check whether a model is a minimal model. In general, this is a co-NP-complete task, but in the context of Theorem 1 we can achieve fixed-parameter tractability based on the following construction and lemma.

Let \( P \) be a given program without tautological rules, \( X \) a strong **Normal**-backdoor of \( P \) of size \( k \), and let \( M \subseteq \text{at}(P) \). For a set \( X_1 \subseteq M \cap X \) we construct a program \( P_{X_1 \subseteq X} \) as follows:

1. remove all rules \( r \) for which \( H(r) \cap X_1 \neq \emptyset \) and
2. replace for all remaining rules \( r \) the head \( H(r) \) with \( H(r) \setminus X \) and the positive body \( B^+(r) \) with \( B^+(r) \setminus X_1 \).

Since \( X \) is a strong **Normal**-backdoor of \( P \), it is also a deletion **Normal**-backdoor of \( P \) by Lemma 1. Hence \( P \setminus X \) is normal. Let \( r \) be an arbitrarily chosen rule in \( P \). Then there is a corresponding rule \( r' \in P \setminus X \) and a corresponding rule \( r'' \in P_{X_1 \subseteq X} \). Since we remove in both constructions exactly the same literals from the head of every rule, \( H(r') = H(r'') \) holds. Consequently \( P_{X_1 \subseteq X} \) is normal, and \( P_{X_1 \subseteq X}^M \) is Horn (here \( P_{X_1 \subseteq X}^M \) denotes the GL reduct of \( P_{X_1 \subseteq X} \) under \( M \)).
For any program \( P' \) let Constr\((P') \) denote the set of constrains of \( P' \) and Pos\((P') = P' \setminus \text{Constr}(P') \). If \( P' \) is Horn, Pos\((P') \) has a least model \( L \) and \( P' \) has a model if and only if \( L \) is a model of Constr\((P') \) [7].

Given \( M \subseteq \text{at}(P) \), the algorithm MinCheck below performs the following steps for each set \( X_1 \subseteq M \cap X \):

1. Compute the Horn program \( P^M_{X_1 \subseteq X} \).
2. Compute the least model \( L \) of Pos\((P^M_{X_1 \subseteq X}) \).
3. Return True if at least one of the following conditions holds:
   - \( L \) is not a model of Constr\((P^M_{X_1 \subseteq X}) \).
   - \( L \) is not a subset of \( X \),
   - \( L \cup X_1 \) is not a proper subset of \( M \),
   - \( L \cup X_1 \) is not a model of \( P^M \).
4. Otherwise return False.

We say that the algorithm succeeds if the algorithm returns True for each set \( X_1 \subseteq M \cap X \).

**Lemma 2** A model \( M \subseteq \text{at}(P) \) of \( P^M \) is a minimal model of \( P^M \) if and only if the algorithm MinCheck succeeds.

Because of space constraints we omit the lengthy but straightforward proof.

We are now in the position to establish Theorem 1.

**Proof (of Theorem 1).** First we check whether \( M \) is a model of \( P^M \). If \( M \) is not a model of \( P^M \) then it is not an answer set of \( P \), and we can neglect it. Hence assume that \( M \) is a model of \( P^M \). Now we run the algorithm MinCheck. By Lemma 2 the algorithm decides whether \( M \) is an answer set of \( P \).

In order to complete the proof, it remains to bound the running time. The check whether \( M \) is a model of \( P^M \) can clearly be carried out in linear time. For each set \( X_1 \subseteq M \cap X \) the algorithm MinCheck runs in linear time. This follows directly from the fact that we can compute the least model of a Horn program in linear time [7]. As there are at most \( 2^k \) sets \( X_1 \) to consider, the total running time is \( O(2^k n) \) where \( n \) denotes the input size of \( P \) and \( k = |X| \). Thus, in particular, the decision is fixed-parameter tractable for parameter \( k \).

**Example 3.** Consider the program \( P \) from Example 1. Let \( M = \{b, c, g\} \subseteq \text{at}(P) \). Since \( M \) satisfies all rules in \( P \), the set \( M \) is a model of \( P \). Hence the GL-reduct \( P^M = \{a \lor c \leftarrow b; c \leftarrow a; b \lor c \leftarrow e; a \lor b; g; c\} \). We apply the algorithm MinCheck for each subset of \( \{b, c\} \). For \( X_1 = \emptyset \) we obtain \( P^M_{X_1 \subseteq X} = \{a \leftarrow b; c \leftarrow a; e \leftarrow c; a \lor b; g; c\} \). The set \( L = \{a\} \) is the least model of Pos\((P^M_{X_1 \subseteq X}) \). Since Condition (a) holds (\( L \) is not a model of Constr\((P^M_{X_1 \subseteq X}) \)), the algorithm returns True for \( X_1 \). For \( X_2 = \{b\} \) we have \( P^M_{X_2 \subseteq X} = \{a \leftarrow a; g; c\} \) and the least model \( L = \{a, g\} \) of Pos\((P^M_{X_2 \subseteq X}) \). Since Condition (a) holds (\( L \) is not a model of \( P^M_{X_2 \subseteq X} \)), MinCheck returns True for \( X_2 \). For \( X_3 = \{c\} \) we gain \( P^M_{X_3 \subseteq X} = \{a; g\} \) and the least model \( L = \{a, g\} \) of Pos\((P^M_{X_3 \subseteq X}) \). Since Condition (c) holds (\( L \cup X_3 \) is not a proper subset of \( M \)), the algorithm returns True for \( X_3 \). For \( X_4 = \{b, c\} \) we obtain \( P^M_{X_4 \subseteq X} = \{g\} \). The set \( L = \{g\} \) is the
least model of \( \text{Pos}(P_{X_4 \subseteq X}) \). Since Condition (c) holds (\( L \cup X_4 \) is not a proper subset of \( M \)), the algorithm returns \text{True} for \( X_4 \). The Algorithm MinCheck returns \text{True} for all subsets \( X_i \) of \( M \cap X \). Consequently, \( M \) is a minimal model of \( P^M \) and thus an answer set of \( P \).

Next we state and prove that there are fpt-reductions from Strong Normal-Backdoor-Brave-Reasoning and Strong Normal-Backdoor-Skeptical-Reasoning to \text{Sat} which is the main result of this paper.

**Theorem 2** Given a program \( P \) of input size \( n \), a strong Normal-backdoor \( X \) of \( P \) of size \( k \), and an atom \( a^* \in \text{at}(P) \), we can produce in time \( O(2^k n^2) \) propositional CNF formulas \( F_{\text{Brave}}(a^*) \) and \( F_{\text{Caut}}(a^*) \) such that (i) \( F_{\text{Brave}}(a^*) \) is satisfiable if and only if \( a^* \) is in some answer set of \( P \), and (ii) \( F_{\text{Caut}}(a^*) \) is unsatisfiable if and only if \( a^* \) is in all answer sets of \( P \).

**Proof.** We would like to use a similar approach as in the proof of Theorem 1. However, we cannot consider all possible models \( M \) one by one, as there could be too many of them. Instead, we will show that it is possible to implement MinCheck nondeterministically in such a way that we do not need to know \( M \) in advance. Possible sets \( M \) will be represented by the truth values of certain variables, and since the truth values do not need to be known in advance, this will allow us to consider all possible sets \( M \) without enumerating them.

Next we describe the construction of the formulas \( F_{\text{Brave}}(a^*) \) and \( F_{\text{Caut}}(a^*) \) in detail. In fact, we will first construct propositional formulas \( F_{\text{Brave}}'(a^*) \) and \( F_{\text{Caut}}'(a^*) \) that are not in CNF. The required CNF formulas \( F_{\text{Brave}}(a^*) \) and \( F_{\text{Caut}}(a^*) \) can then be obtained from \( F_{\text{Brave}}'(a^*) \) and \( F_{\text{Caut}}'(a^*) \) by means of the well-known transformation due to Tseitin [37], see also [27]. This transformation produces for a given propositional formula \( F' \) in linear time a CNF formula \( F \) such that both formulas are equivalent with respect to their satisfiability, and the length of \( F \) is linear in the length of \( F' \).

Among the variables of our formulas will be a set \( V := \{ v[a] \mid a \in \text{at}(P) \} \) containing a variable for each atom of \( P \). The truth values of the variables in \( V \) represent a subset \( M \subseteq \text{at}(P) \), such that \( v[a] \) is true if and only if \( a \in M \).

We define

\[
F_{\text{Brave}}'(a^*) := F^\text{mod} \land F^\text{min} \land v[a^*] \quad \text{and} \quad F_{\text{Caut}}'(a^*) := F^\text{mod} \land F^\text{min} \land \lnot v[a^*],
\]

where \( F^\text{mod} \) and \( F^\text{min} \) are formulas, defined below, that check whether the truth values of the variables in \( V \) represent a model \( M \) of \( P^M \), and whether \( M \) is a minimal model of \( P^M \), respectively.

The definition of \( F^\text{mod} \) is easy:

\[
F^\text{mod} := \bigwedge_{r \in P} \left( \bigwedge_{b \in B^-(r)} \lnot v[b] \rightarrow \left( \bigvee_{b \in B^+(r)} \lnot v[b] \lor \bigvee_{b \in H(r)} v[b] \right) \right).
\]

The definition of \( F^\text{min} \) is more involved. First we define:

\[
F^\text{min} := \bigwedge_{1 \leq i \leq 2^k} F^\text{min}_i,
\]
where $F_{i}^{\text{min}}$, defined below, encodes the Algorithm MinCheck for the set $X_{i}$ where $X_{1}, \ldots, X_{2k}$ is an enumeration of all the subsets of $X$.

The formula $F_{i}^{\text{min}}$ will contain, in addition to the variables in $V$, $p$ distinct variables for each atom of $P$, $p := \min(|P|, |\text{at}(P)|)$. In particular, the set of variables of $F_{i}^{\text{min}}$ is the disjoint union of $V$ and $U_{i}$ where $U_{i} := \{u_{i}^{j}[a] \mid a \in \text{at}(P), 1 \leq j \leq p\}$. We use $U_{i}$ for the subset of $U_{i}$ containing all the variables $u_{i}^{j}[a]$. We assume that for $i \neq i'$ the sets $U_{i}$ and $U_{i'}$ are disjoint. For each $a \in \text{at}(P)$ we also use the propositional constants $X(a)$ and $X_{1}(a)$ that are true if and only if $a \in X$ and $a \in X_{1}$, respectively.

We define the formula $F_{i}^{\text{min}}$ by means of the following auxiliary formulas.

The first auxiliary formula checks whether the truth values of the variables in $V$ represent a set $M$ that contains $X_{i}$:

$$F_{i}^{(a)} := \bigwedge_{a \in X} X_{i}(a) \rightarrow v[a].$$

The next auxiliary formula encodes the computation of the least model ("lm") $L$ of $\text{Pos}(P_{X_{i} \subseteq X}^{M})$ where $M$ and $L$ are represented by the truth values of the variables in $V$ and $U_{i}$, respectively.

$$F_{i}^{\text{lm}} := \bigwedge_{a \in \text{at}(P), 0 \leq j \leq p} F_{i}^{(a,i)},$$

where

$$F_{i}^{(a,0)} := u_{i}^0 \leftrightarrow \text{false},$$

$$F_{i}^{(a,j)} := u_{i}^{j}[a] \leftrightarrow \left[u_{i}^{j-1}[a] \bigvee \bigwedge_{r \in P_{X_{i} \subseteq X}, a \in H(r)} \left(\bigwedge_{b \in B^{r}} u_{i}^{j-1}[b] \bigwedge_{b \in B^{-r}} \neg v[b]\right)\right]$$

(for $1 \leq j \leq p - 1$).

The idea behind the construction of $F_{i}^{\text{lm}}$ is to simulate the linear-time algorithm of [7]. Initially, all variables are set to false. This is represented by variables $u_{i}^{0}[a]$. Now we flip a variable from false to true if and only if there is a Horn rule where all the variables in the rule body are true. Once a fixed-point is reached, we have the least model. The flipping is represented in our formula by setting a variable $u_{i}^{j}[a]$ to true if and only if either $u_{i}^{j-1}[a]$ is true, or there is a rule $r \in \text{Pos}(P_{X_{i} \subseteq X}^{M})$ such that $H(r) = \{a\}$ and $u_{i}^{j}[b]$ is true for all $b \in B^{+}(r)$. The truth values of the variables $u_{i}^{p}$ now represent the least model of $\text{Pos}(P_{X_{i} \subseteq X}^{M})$.

The next four auxiliary formulas check whether the respective condition (a)–(d) of algorithm MinCheck does not hold for $L$. $F_{i}^{(a)}$ expresses that there is a rule in $\text{Constr}(P_{X_{i} \subseteq X}^{M})$ that is not satisfied by $L$:

$$F_{i}^{(a)} := \forall_{r \in P_{X_{i} \subseteq X}, H(r) \subseteq X} \left(\bigwedge_{b \in B^{-}(r)} \neg v[b] \bigwedge_{b \in B^{+}(r)} u_{i}^{p}[b]\right).$$

$F_{i}^{(b)}$ expresses that $L$ contains an atom that is not in $M \setminus X$:

$$F_{i}^{(b)} := \forall_{a \in \text{at}(P) \setminus X} \left(\neg v[a] \bigwedge u_{i}^{p}[a]\right).$$

$F_{i}^{(c)}$ expresses that $L \cup X_{i}$ equals $M$ or $L \cup X_{i}$ contains an atom that is not in $M$:

$$F_{i}^{(c)} := \left(\bigwedge_{a \in \text{at}(P)} v[a] \leftrightarrow (u_{i}^{p}[a] \bigvee X_{i}(a))\right) \bigvee \left(\forall_{a \in \text{at}(P)} (u_{i}^{p}[a] \bigvee X_{i}(a)) \bigwedge \neg v[a]\right).$$
$F(d)$ expresses that $PM$ contains a rule that is not satisfied by $L \cup X_i$:

$$F(d)_i := \bigvee_{r \in P} \left[ \bigwedge_{a \in B^-(r)} \neg v[a] \wedge \bigwedge_{a \in H(r)} (\neg u_i^p[a] \wedge \neg X_i(a)) \wedge \bigwedge_{b \in B^+(r)} (u_i^p[b] \lor X_i(b)) \right].$$

Now we can put the auxiliary formulas together and obtain

$$F_{\text{min}}^i := (F_{\text{Caut}}^i \wedge F_{\text{Brave}}^i) \rightarrow (F_{\text{mod}}^i \lor F_{\text{Caut}}^i \lor F_{\text{Brave}}^i).$$

It follows by Lemma 2 and by the construction of the auxiliary formulas that

(i) $F_{\text{Brave}}^i(a^*)$ is satisfiable if and only $a^*$ is in some answer set of $P$, and
(ii) $F_{\text{Caut}}^i(a^*)$ is unsatisfiable if and only if $a^*$ is in all answer sets of $P$.

Hence it remains to observe that for each $i \leq 2^k$ the auxiliary formula $F_{\text{min}}^i$ can be constructed in quadratic time, whereas the auxiliary formulas $F_{\text{Caut}}^i$ and $F_{\text{Brave}}^i \lor F_{\text{Caut}}^i \lor F_{\text{Brave}}^i$ can be constructed in linear time. Since $|X| = k$ by assumption, we need to construct $O(2^k)$ auxiliary formulas in order to obtain $F_{\text{Caut}}^i(a^*)$ and $F_{\text{Brave}}^i(a^*)$. Hence the running time as claimed in Theorem 2 follows and the theorem is established.

Example 4. Consider the program $P$ from Example 1 and the strong Normal-backdoor $X = \{b, c, h\}$ of $P$ from Example 2. We ask whether the atom $b$ is contained in at least one answer set. To decide the question, we explain that $F_{\text{Brave}}^i(b)$ is satisfiable and we answer the question positively. Since $M = \{b, c, g\}$ is model of $PM$ we can satisfy $F_{\text{mod}}^i$ with a truth assignment $\tau$ that maps 1 to each variable $v[x]$ where $x \in \{b, c, g\}$ and 0 to each variable $v[x]$ where $x \in \text{at}(P)$ \{\{b, c, g\}. For $i = 1$ let $X_1 = \emptyset$. Then we have for the constants $X_1(x) = 0$ where $x \in \{b, c\}$. Observe that $\tau$ already satisfies $F_{\text{Caut}}^i$ and that $F_{\text{Brave}}^i$ encodes the computation of the least model $L$ of $\text{Pos}(PM_{X_1 \subseteq X})$ where $L$ is represented by the truth values of the variables in $U^p = \{u_i^p[x] \mid x \in \text{at}(P)\}$. Thus $\tau$ also satisfies $F_{\text{Brave}}^i$ if $\tau$ maps $u_i^p[a]$ to 1 and $u_i^p[x]$ to 0 where $x \in \text{at}(P) \setminus \{a\}$. As $\tau$ satisfies $F_{\text{Brave}}^i$, the truth assignment $\tau$ satisfies the formula $F_{\text{min}}^i$. It is not hard to see that $F_{\text{min}}^i$ is satisfiable for other values of $i$. Hence the formula $F_{\text{Brave}}^i(b)$ is satisfiable and $b$ is contained in at least one answer set.

Completeness for paraNP and co-paraNP

The parameterized complexity class paraNP contains all parameterized decision problems $L$ such that $(I, k) \in L$ can be decided nondeterministically in time $O(f(k)||I||^c)$, for some computable function $f$ and constant $c$ [15]. By co-paraNP we denote the class of all parameterized decision problems whose complement (the same problem where yes and no answers are swapped) is in paraNP.

If a non-parameterized problem is NP-complete, then adding a parameter that makes it paraNP-complete does not make much sense, as this holds even true if the parameter is the constant 0. Therefore a paraNP-completeness result for a problem that without parameterization is in NP, is usually considered as an utterly negative result. However, if the considered problem without parameter is outside NP, and we can show that with a suitable parameter the problem becomes paraNP-complete, this is in fact a positive result. Indeed, we get such a positive result as a corollary to Theorem 2.
Corollary 1 Strong Normal-Backdoor-Brave-Reasoning is paraNP-complete, and Strong Normal-Backdoor-Skeptical-Reasoning is paraNP-complete.

Proof. If a parameterized problem \( L \) is NP-hard when we fix the parameter to a constant, then \( L \) is paraNP-hard ([15], Theorem 2.14). As Strong Normal-Backdoor-Brave-Reasoning is NP-hard for backdoor size 0, we conclude that Strong Normal-Backdoor-Brave-Reasoning is paraNP-hard. A similar argument shows that Strong Normal-Backdoor-Skeptical-Reasoning is co-paraNP-hard.

Sat, considered as a parameterized problem with constant parameter 0, is clearly paraNP-complete, this also follows from the mentioned result of Flum and Grohe [15]; hence UnSat is co-paraNP-complete. As Theorem 2 provides fpt-reductions from Strong Normal-Backdoor-Brave-Reasoning to Sat, and from Strong Normal-Backdoor-Skeptical-Reasoning to UnSat, we conclude that Strong Normal-Backdoor-Brave-Reasoning is in paraNP, and Strong Normal-Backdoor-Skeptical-Reasoning is in co-paraNP. \( \square \)

5 Finding Backdoors

In this section, we study the problem of finding backdoors which yields the following parameterized problem:

**Strong \( C \)-Backdoor-Detection**

**Instance:** A (disjunctive logic) program \( P \).

**Parameter:** An integer \( k \).

**Question:** Find a strong \( C \)-backdoor \( X \) of \( P \) of size at most \( k \), or report that such \( X \) does not exist.

We also consider the problem **Deletion \( C \)-Backdoor-Detection**, defined similarly.

Let \( P \) be a program. Let the head dependency graph \( U_H^P \) be the undirected graph \( U_H^P = (V, E) \) defined on the set \( V = at(P) \) of atoms of the given program \( P \), where two atoms \( x, y \) are joined by an edge \( xy \in E \) if and only if \( P \) contains a non-tautological rule \( r \) with \( x, y \in H(r) \). A vertex cover of a graph \( G = (V, E) \) is a set \( X \subseteq V \) such that for every edge \( uv \in E \) we have \( \{u, v\} \cap X \neq \emptyset \).

**Lemma 3** Let \( P \) be a program. A set \( X \subseteq at(P) \) is a deletion \( C \)-backdoor of \( P \) if and only if \( X \) is a vertex cover of \( U_H^P \).

Proof. Let \( X \) be a deletion \( C \)-backdoor of \( P \). Consider an edge \( uv \) of \( U_H^P \), then there is a rule \( r \in P \) with \( u, v \in H(r) \) and \( u \neq v \). Since \( X \) is a deletion \( C \)-backdoor set of \( P \), we have \( \{u, v\} \cap X \neq \emptyset \). We conclude that \( X \) is a vertex cover of \( U_H^P \).

Conversely, assume that \( X \) is a vertex cover of \( U_H^P \). Consider a rule \( r \in P - X \) for proof by contradiction. If \( |H(r)| \geq 2 \) then there are two variables \( u, v \in H(r) \) and an edge \( uv \) of \( U_H^P \) such that \( \{u, v\} \cap X = \emptyset \), contradicting the assumption that \( X \) is a vertex cover. Hence the lemma prevails. \( \square \)
Theorem 3  The problems Strong Normal-Backdoor-Detection and Deletion Normal-Backdoor-Detection are fixed-parameter tractable. In particular, given a program $P$ of input size $n$, and an integer $k$, we can find in time $O(1.2738^k + kn)$ a strong Normal-backdoor of $P$ with a size $\leq k$ or decide that no such backdoor exists.

Proof. In order to find a deletion Normal-backdoor of a given program $P$, we use Lemma 3 and find a vertex cover of size at most $k$ in the head dependency graph $U^P$. A vertex cover of size $k$, if it exists, can be found in time $O(1.2738^k + kn)$ [5]. Then the theorem holds for deletion Normal-backdoors. Lemma 1 states that the strong Normal-backdoors of $P$ are exactly the deletion Normal-backdoors of $P$ (as we assume that $P$ does not contain any tautological rules). Thus the theorem follows. □

In Theorem 2 we assume that a strong Normal-backdoor of size at most $k$ is given when solving the problems Strong Normal-Backdoor-Brave-Reasoning and Skeptical-Reasoning. As a direct consequence of Theorem 3, this assumption can be dropped, and we obtain the following corollary.

Corollary 2  The results of Theorem 2 and Corollary 1 sustain if the backdoor is not given as part of the input.

6 Backdoors to Tightness

We associate with each program $P$ its positive dependency graph $D^+_P$. It has as vertices the atoms of $P$ and a directed edge $(x, y)$ between any two atoms $x, y \in \text{at}(P)$ for which there is a rule $r \in P$ with $x \in H(r)$ and $y \in B^+(r)$. A program is called tight if $D^+_P$ is acyclic [29].

It is well known that the reasoning problems are in NP and co-NP for tight programs; in fact, a reduction to SAT based on the concept of loop formulas has been proposed by Lin and Zhao [31]. This was then generalized by Lee and Lifschitz [29] with a reduction that takes as input a disjunctive normal program $P$ together with the set $S$ of all directed cycles in the positive dependency graph of $P$, and produces a CNF formula $F$ such that answer sets of $P$ correspond to the satisfying assignments of $F$. This provides an fpt-reduction from the problems brave reasoning and cautious reasoning to SAT, when parameterized by the number of all cycles in the positive dependency graph of a given program $P$, assuming that these cycles are given as part of the input.

The number of cycles does not seem to be a very practical parameter, as this number can quickly become very large even for very simple programs. Lifschitz and Razborov [30] have shown that already for normal programs an exponential blowup may occur, since the number of cycles in a normal program can be arbitrarily large. Hence it would be interesting to generalize the result of Lee and Lifschitz [29] to a more powerful parameter. In fact, the size $k$ of a deletion Tight-backdoor would be a candidate for such a parameter, as it is easy to see that it is at most as large as the number of cycles, but can be exponentially smaller. This results from the following two observations: (i) If a program $P$ has exactly $k$ cycles in $D^+_P$, we can construct a deletion Tight-backdoor $X$ of $P$ by taking one element from each cycle into $X$. (ii) If a program $P$ has a deletion
Tight-backdoor of size 1, it can have arbitrary many cycles that run through the atom in the backdoor.

In the following, we show that this parameter $k$ is not of much use, as the reasoning problems already reach their full complexity for programs with a deletion Tight-backdoor of size 1.

**Theorem 4** The problems Strong Tight-Backdoor-Brave-Reasoning and Strong Tight-Backdoor-Skeptical-Reasoning are $\Sigma_2^P$-hard and $\Pi_2^P$-hard, respectively, even for programs that admit a strong Tight-backdoor of size 1, and the backdoor is provided with the input. The problems remain hard when we consider a deletion Tight-backdoor instead of a strong Tight-backdoor.

**Proof (Sketch).** The reduction of Eiter and Gottlob [10] produces a program $P := \{ x_i \lor v_i; y_i \lor z_j; y_j \leftarrow w; z_j \leftarrow y_j, z_j; w \leftarrow g(l_{k,1}), g(l_{k,2}), g(l_{k,3}); w \leftarrow \neg w \}$ for each $i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}, k \in \{1, \ldots, r\}$, and $g$ maps as follows $g(\neg x_i) = v_i, g(\neg y_j) = z_j,$ and otherwise $g(l) = l.$ Thus $P$ has deletion Tight-backdoors and strong Tight-backdoors of size 1. Consequently, the restriction does not yield tractability.

\( \square \)

7 Conclusion

We have shown that backdoors of small size capture structural properties of ASP instances that yield to a reduction of problem complexity. In particular, small backdoors to normality admit an fpt-translation from ASP to SAT and thus reduce the complexity of the fundamental ASP problems from the second level of the Polynomial Hierarchy to the first level. Thus, the size of a smallest Normal-backdoor is a structural parameter that admits a fixed-parameter tractable complexity reduction without making the problem itself fixed-parameter tractable.

Our result gives the use of fixed-parameter tractability a new twist and enlarges its applicability. In fact, our approach as exemplified above for ASP is very general and might be applicable to a wide range of other hard combinatorial problems that lie beyond NP or co-NP. We hope that our work stimulates further investigations into this direction.

It might be interesting to investigate the practical potential of our SAT-approach to ASP. In order to make it work in practice, several improvements can be considered. For instance, with shifting [26] one can preprocess a program and possibly reduce the size of Normal-backdoors. A further possibility is to replace the quadratic formula $F_{\text{lcm}}^i$ for the computation of least models with a smaller and more sophisticated SAT or SAT (DL) encoding [22, 25].

References

Answer Set Programming for Stream Reasoning

M. Gebser¹, T. Grote¹, R. Kaminski¹, P. Obermeier², O. Sabuncu¹, and T. Schaub¹

¹ Universität Potsdam, Germany
² DERI Galway, Ireland

Abstract. The advance of Internet and Sensor technology has brought about new challenges evoked by the emergence of continuous data streams. Beyond rapid data processing, application areas like ambient assisted living, robotics, or dynamic scheduling involve complex reasoning tasks. We address such scenarios and elaborate upon approaches to knowledge-intense stream reasoning, based on Answer Set Programming (ASP). While traditional ASP methods are devised for singular problem solving, we develop new techniques to formulate and process problems dealing with emerging as well as expiring data in a seamless way.

1 Introduction

The advance of Internet and Sensor technology has brought about new challenges evoked by the emergence of continuous data streams, like web logs, mobile locations, or online measurements. While existing data stream management systems [4] allow for high-throughput stream processing, they lack complex reasoning capacities [5]. We address this shortcoming and elaborate upon approaches to knowledge-intense stream reasoning, based on Answer Set Programming (ASP; [6]) as a prime tool for Knowledge Representation and Reasoning (KRR; [7]). The emphasis thus shifts from rapid data processing towards complex reasoning, as required in application areas like ambient assisted living, robotics, or dynamic scheduling.

In contrast to traditional ASP methods, which are devised for singular problem solving, “stream reasoning, instead, restricts processing to a certain window of concern, focusing on a subset of recent statements in the stream, while ignoring previous statements” [8]. To accommodate this in ASP, we develop new techniques to formulate and process problems dealing with emerging as well as expiring data in a seamless way. Our modeling approaches rely on the novel concept of time-decaying logic programs [1], where logic program parts are associated with life spans to steer their emergence as well as expiration upon continuous reasoning. Time-decaying logic programs are implemented as a recent extension of the reactive ASP system oclingo [9], using the ASP grounder gringo [10] for the recurrent composition of a static “offline” encoding with dynamic “online” data into queries to the ASP solver clasp [11].

While oclingo makes powerful ASP technology accessible for stream reasoning, its continuous query formulation and processing impose particular modeling challenges.

* This paper complements a short KR’12 paper [1]; an extended draft [2] is available at [3].
** Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.
First, re-grounding parts of an encoding wrt. new data shall be as economical as possible. Second, traditional modeling techniques, eg. frame axioms [12], need to be reconsidered in view of the expiration of obsolete program parts. Third, the re-use of propositional atoms and rules shall be extensive to benefit from conflict-driven learning (cf. [13]). We here tackle such issues in continuous reasoning over data from a sliding window [4]. In a nutshell, we propose to encode knowledge about any potential window contents offline, so that dynamic logic program parts can concentrate on activating readily available rules (via transient online data). Moreover, we show how default conclusions, eg. expressed by frame axioms, can be faithfully combined with transient data.

After providing the necessary background, we demonstrate modeling approaches on three toy domains. The underlying principles are, however, of general applicability and thus establish universal patterns for ASP-based reasoning over sliding windows.³

2 Background

We presuppose familiarity with (traditional) ASP input languages (cf. [14, 15]) and (extended) logic programs (cf. [16, 6]). They provide the basis of incremental logic programs [17], where additional keywords, "#base," "#cumulative," and "#volatile," allow for partitioning rules into a static, an accumulating, and a transient program part. The latter two usually refer to some constant $t$, standing for a step number. In fact, when gradually increasing the step number, starting from 1, ground instances of rules in a #cumulative block are successively generated and joined with ground rules from previous steps, whereas a #volatile block contributes instances of its rules for the current step only. Unlike accumulating and transient program parts, which are re-processed at each incremental step, the static part indicated by #base is instantiated just once, initially, so that its rules correspond to a “0th” #cumulative block.

Application areas of incremental logic programs include planning (cf. [18]) and finite model finding (cf. [19]). For instance, (a variant of) the well-known Yale shooting problem (cf. [20]) can be modeled by an incremental logic program as follows:

```plaintext
#base.
{ loaded }.
live(0).
#cumulative t.
{ shoot(t+1) }.
  ab(t) :- shoot(t), loaded.
live(t) :- live(t-1), not ab(t).
#volatile t.
:- live(t).
:- shoot(t+1).
```

The first answer set, containing loaded, live(0), live(1), shoot(2), and ab(2), is generated from the following ground rules at step 2:

```plaintext
% #base.
{ loaded }.
live(0).
```

³ For formal details on time-decaying logic programs and a discussion of related work in stream processing, we refer the interested reader to [1, 2].
Observe that the static #base part is augmented with rules from the #cumulative block for step 1 and 2, whereas #volatile rules are included for step 2 only.

The reactive ASP system oclingo extends offline incremental logic programs by functionalities to incorporate external online information from a controller, also distinguishing accumulating and transient external inputs. For example, consider a character stream over alphabet \{a, b\} along with the task of continuously checking whether a stream prefix at hand matches regular expression \((a|b)^*aa\). To provide the stream prefix \aab, the controller component of oclingo can successively pass facts as follows:

#step 1. read(a,1).
#step 2. read(a,2).
#step 3. read(b,3).

The number \(i\) in “#step \(i\).” directives informs oclingo about the minimum step number up to which an underlying offline encoding must be instantiated in order to incorporate external information (given after a #step directive) in a meaningful way. In fact, the following offline encoding builds on the assumption that values \(i\) in atoms of the form read(a, \(i\)) or read(b, \(i\)) are aligned with characters’ stream positions:

#iinit 0.
#cumulative t.
#external read(a, t+1). #external read(b, t+1).
accept(t) :- read(a, t), read(a, t-1), not read(a, b, t+1).

The (undefined) atoms appearing after the keyword “#external” are declared as inputs to #cumulative blocks; that is, they are protected from program simplifications until they become defined (by external rules from the controller). Observe that any instance of the predicate read/2 is defined externally. In particular, future inputs that can be provided at (schematic) step \(t+1\) are used to cancel an obsolete rule defining accept(t), once it does no longer refer to the last position of a stream prefix. Given that inputs are expected from step \(1 = t+1\) on, the directive “#iinit 0.” specifies \(t = 0\) as starting value to instantiate #cumulative blocks for, so that read(a, 1) and read(b, 1) are initially declared as external inputs. In view of this, the answer set obtained for stream prefix \aab\ at step 2 includes accept(2) (generated via “accept(2) :- read(a, 2), read(a, 1), not read(a, 3), not read(b, 3).”), while accept(3) does not belong to the answer set for \aab\ at step 3. Note that acceptance at different stream positions is indicated by distinct instances of accept/1; this is important to comply with modularity conditions (cf. [1, 2]) presupposed by oclingo, which essentially object to the (re)definition of (ground) head atoms at different steps.

Although the presented reactive ASP encoding correctly accepts stream prefixes matching \((a|b)^*aa\), the fact that external instances of read/2 are accumulated over time is a major handicap, incurring memory pollution upon running oclingo for a (quasi)
To circumvent this, external inputs could be made transient, as in the following alternative sequence of facts from oclingo’s controller component:

- **Step 1.** 
  - volatile. read(a,1,1).
- **Step 2.** 
  - volatile. read(a,1,2). read(a,2,2).
- **Step 3.** 
  - volatile. read(a,2,3). read(b,3,3).

Note that the first two readings, represented by read(a,1,1) and read(a,2,2) as well as read(a,1,2) and read(a,2,3), are provided by facts twice, where the respective (last) stream position is included as additional argument to avoid redefinitions.

In view of this, the previous offline encoding could be replaced by the following one:

```
#cumulative t.
#external read(a,t-1;t,t). #external read(b,t-1;t,t).
accept(t) :- read(a,t-1), read(a,t,t).
```

While the automatic expiration of transient inputs after each inquiry from the controller (along with the possible use of “#forget i.” directives) omits a blow-up in space as well as an explicit cancelation of outdated rules, it leads to the new problem that the whole window contents (two readings in this case) must be passed in each inquiry from the controller. This delegates the bookkeeping of sliding window contents to the controller, which then works around limitations of reactive ASP as introduced in [9].

Arguably, neither accumulating inputs (that are no longer inspected) over time nor replaying window contents (the width of the window many times) is acceptable in performing continuous ASP-based stream reasoning. To overcome the preexisting limitations, we introduced time-decaying logic programs [1] that allow for associating arbitrary life spans (rather than just 1) with transient program parts. Such life spans are given by integers l in directives of the form “#volatile : l.” (for online data) or “#volatile t : l.” (for offline encoding parts). With our example, stream readings can now be conveniently passed in #volatile blocks of life span 2 as follows:

- **Step 1.** 
  - volatile : 2. read(a,1).
- **Step 2.** 
  - volatile : 2. read(a,2).
- **Step 3.** 
  - volatile : 2. read(b,3).

In view of automatic expiration in two steps (e.g. at step 3 for read(a,1) provided at step 1), the following stripped-down offline encoding correctly handles stream prefixes:

```
#cumulative t.
#external read(a,t). #external read(b,t).
accept(t) :- read(a,t-1), read(a,t).
```

Note that the embedding of encoding rules in a #cumulative block builds on automatic rule simplifications relative to expired inputs. As an (equivalent) alternative, one could use “#volatile t : 2.” to discard outdated rules via internal assumption literals (cf. [17]). However, we next investigate more adept uses of blocks for offline rules.

### 3 Modeling and Reasoning

The case studies provided below aim at illustrating particular features in modeling and reasoning with time-decaying logic programs and stream data. For the sake of clarity,

---

4 Disposal of elapsed input atoms that are yet undefined, such as read(b,1), read(b,2), and read(a,3) wrt. stream prefix aab, can be accomplished via “#forget i.” directives. Ground rules mentioning such atoms are in turn “automatically” simplified by clasp (cf. [17]).
we concentrate on toy domains rather than any actual target application. We begin with
modelings of the simple task to monitor consecutive user accesses, proceed with an
overlapping scenario utilizing frame axioms, and then turn to the combinatorial problem
of online job scheduling; the corresponding encodings can be downloaded at [3].

3.1 Access Control

Our first scenario considers users attempting to access some service, for instance, by
logging in via a website. Access attempts can be denied or granted, eg. depending on
the supplied password, and a user account is (temporarily) closed in case of three access
denials in a row. A stream segment of access attempts is shown in Listing 1. It provides
data about three users, alice, bob, and claude. As specified by ”#volatile : 3.”
(for each step), the life span of access data is limited to three incremental steps (which
may be correlated to some period of real time), aiming at an (automatic) reopening of
closed user accounts after some waiting period has elapsed. We further assume that time
stamps in the third argument of facts over access/3 deviate from i in ”#step i.” by
at most 2; that is, the terms used in transient facts are coupled to the step number (in an
underlying incremental logic program). Given the segment in Listing 1, the following
table summarizes non-expired logged accesses per step, where granted accesses are
enclosed in brackets and sequences of three consecutive denials are underlined:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>[1]</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>bob</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>claude</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

For instance, observe that the three denied accesses by bob logged in the second and
fourth step are consecutive in view of the time stamps 3, 2, and 4 provided as argument
values, eg. in access(bob,denied,3) expiring at step 5.
Listing 2. Cumulative access control encoding

```prolog
#const window = 3. #const offset = 2. #const denial = 3. #iinit 1 - offset.

#base.
user(bob; alice; claude). % some users
signal(denied; granted). % some signals
account(U, closed) : user(U).
account(U, open) :- user(U), not account(U, closed).

#cumulative t.
access(U, S, t + offset) : user(U) : signal(S).
denied(U, 1, t) :- access(U, denied, t + offset).
denied(U, N + 1, t) :- access(U, denied, t + offset),
denied(U, N, t - 1), N < denial.
denied(U, denial, t) :- denied(U, denial, t - 1).
:- denied(U, denial, t), not account(U, closed).

#volatile t.
:- account(U, closed), not denied(U, denial, t).
```

Our first offline encoding is shown in Listing 2. To keep the sliding window width, matching the life span of stream transients, adjustable, the constant window is introduced in Line 1 (and set to default value 3). Similarly, the maximum deviation of time stamps from incremental step numbers and the threshold of consecutive denied accesses at which an account is closed are represented by the constants offset and denial. After introducing the three users and the possible outcomes of their access attempts via facts, the static #base part includes in Line 6 a choice rule on whether the status of a user account is closed, while it remains open otherwise. In fact, the dynamic parts of the incremental program solve the task to identify the current status of the account of a user U and represent it by including either account(U, closed) or account(U, open) in an answer set, yet without redefining account/2 over steps. This makes sure that stepwise (ground) incremental program parts are modularly composable (cf. [1, 2]), which is required for meaningful closed-world reasoning by oclingo in reactive settings.

The encoding in Listing 2 (mainly) relies on accumulating rules given below “#cumulative t.” (in Line 9), resembling incremental planning encodings (cf. [17]) based on a history of actions. In order to react to external inputs, the #external directive in Line 10 declares (undefined) atoms as inputs that can be provided by the environment, i.e. the controller component of oclingo. Note that instances of access/3 with time stamp t + offset (where offset is the maximum deviation from t) are introduced at incremental step t; in this way, ground rules are prepared for shifted access data arriving early. The rules in Line 11–13 implement the counting of consecutive denied access attempts per user, up to the threshold given by denial; if this threshold is reached (wrt. non-expired access data), the account of the respective user is temporarily closed. The “positive” conclusion from denied many denied access attempts to closing an account is encoded via the integrity constraint in Line 15, while more care is needed in concluding the opposite: the right decision whether to leave an account open can be made only after inspecting the whole window contents. To this end, the rule in Line 14 passes information about the threshold being reached on to later steps, and the query
in Line 18, refuting an account to be closed if there were no three consecutive denied access attempts, is included only for the (currently) last incremental step.

For the stream segment in Listing 1, in the fourth incremental step, we have that 
\texttt{denied(bob,3,4)} is derived in view of (transient) facts 
\texttt{access(bob,denied,3)}, 
\texttt{access(bob,denied,2)}, and \texttt{access(bob,denied,4)}. Due to the integrity constraint in Line 15 of Listing 2, this enforces \texttt{account(bob,closed)} to hold. Since \texttt{access(bob,denied,3)} expires at step 5, we can then no longer derive \texttt{denied(bob,3,4)}, and \texttt{denied(bob,3,5)} does not hold either; the query in Line 18 thus enforces \texttt{account(bob,closed)} to be false in the fifth incremental step. Similarly, we have that \texttt{account(claude,closed)} or \texttt{account(alice,closed)} hold at step 5, 6, and 8, respectively, but are enforced to be false at any other step. As one may have noticed, given the values of constants in Listing 2, the consideration of atoms over \texttt{access/3} starts at \( t + \text{offset} = 3 \) for \( t = 1 \). To still initialize the first (two) windows wrt. the “past,” an \texttt{#init} directive is provided in Line 1. By using \( 1 - \text{offset} = -1 \) as starting value for \( t \), once the first external inputs are processed, ground rules treating access data with time stamps 1, 2, and 3 are readily available, which is exactly the range admitted at the first step. Moreover, at a step like 6, the range of admissible time stamps starts at 4, and it increases at later steps; that is, inputs with time stamp 3, declared at step 1, are not anymore supposed to occur in stream data. To enable program simplifications by fixing such atoms to false, online data can be accompanied by \texttt{#forget i}.” directives, and they are utilized in practice to dispose of elapsed input atoms (cf. [3]).

In addition to the cumulative encoding in Listing 2, we also devised a volatile variant (cf. extended draft [2]) in which outdated rules expire along with stream data. Both the cumulative and the volatile encoding have the drawback that the knowledge-intensive logic program part is (gradually) replaced at each step. While this is tolerable in the simple access scenario, for more complex problems, it means that the internal problem representation of a solving component like clasp changes significantly upon processing stream data, so that a potential re-use of learned constraints is limited. In order to reinforce conflict-driven learning, we next demonstrate a modeling approach allowing for the preservation of a static problem representation in view of fixed window capacity.

At the beginning (up to Line 5), our static access control encoding, shown in Listing 3, is similar to the cumulative approach (Listing 2), while the \texttt{#base} part is significantly extended in the sequel. In fact, the constant \texttt{modulo}, calculated in Line 6, takes into account that at most \texttt{window} many consecutive online inputs are jointly available (preceding ones are expired) and that terms representing time stamps may deviate by up to \texttt{offset} (both positively and negatively) from incremental step numbers. Given this, \( \texttt{window} + 2 \times \texttt{offset} \) slots are sufficient to accommodate all distinct time stamps provided as argument values in instances of \texttt{access/3} belonging to a window, and one additional slot is added in Line 6 as separator between the largest and the smallest (possibly) referenced time stamp. The available slots are then arranged in a cycle (via modulo arithmetic) in Line 7, and the counting of denied access attempts per user, implemented in Line 9–11, traverses consecutive slots according to instances of the predicate \texttt{next/2}. Importantly, counting does not rely on transient external inputs, ie. \texttt{access/3}, but instead refers to instances of \texttt{baseaccess/3}, provided by the choice rule in Line 8; in
Listing 3. Static access control encoding

```prolog
#const modulo = window + 2*offset + 1. #init 2 - modulo.

next(T, (T + 1) mod modulo) :- T := 0..modulo - 1.

account(U, closed) :- denied(U, denial, T).
account(U, open) :- user(U), not account(U, closed).

#cumulative t.
#external access(U, S, t + offset) : user(U) : signal(S).
:- access(U, denied, T), T := t + offset,
not baseaccess(U, denied, T mod modulo).

#volatile t : modulo.
:- baseaccess(U, denied, (T + modulo) mod modulo),
not access(U, denied, T) mod modulo.
```

denial many consecutive instances of baseaccess(U, denied, T), needed to close the account of a user U, correspond to respective facts over access/3 in the current window. Finally, to avoid initial guesses over baseaccess/3 wrt. (non-existing) denied accesses lying in the past, "#init 2 - modulo." is included in Line 6. Then, instances of baseaccess/3 for (positive) values "(T + modulo) mod modulo," calculated in Line 21, are (temporarily) enforced to be false when they match access/3 instances with non-positive time stamps.

this way, the atoms and rules in the #base program part can be re-used in determining the status of user accounts wrt. transient access data from a stream.

To get decisions on closing accounts right (via the rules in Line 12–13), synchronization between instances of baseaccess/3 and (transient) facts over access/3 is implemented in the #cumulative and #volatile parts in Listing 3. Beyond declaring inputs in the same way as before (cf. Line 10 in Listing 2), for any non-expired online input of the form access(U, denied, t + offset), the integrity constraint in Line 17–18 enforces the corresponding instance of baseaccess/3, calculated via "(t + offset) mod modulo," to hold. Note that this constraint does not need to be expired explicitly (yet it could be moved to the #volatile block below) because elapsed input atoms render it ineffective anyway. However, the integrity constraint in Line 21–22, enforcing baseaccess/3 counterparts of non-provided facts of the form access(U, denied, t + offset) to be false, must be discharged once the sliding window progresses (by modulo many steps). For instance, when offset = 2 and modulo = 8, input atoms of the form access(U, denied, 3), introduced at the first step, map to baseaccess(U, denied, 3), and the same applies to instances of access(U, denied, 11), becoming available at the ninth incremental step. Since the smallest time stamp that can be mentioned by non-expired inputs at the ninth step is 5 (inputs given before step 7 are expired), the transition of integrity constraints (mapping time stamp 11 instead of 3 to slot 3) is transparent. In addition, as expired inputs with time stamp 4 enforce atoms of the form baseaccess(U, denied, 4) to be false (via the integrity constraint in Line 21–22 instantiated for step 2), denial counting in Line 9–11 stops at atoms baseaccess(U, denied, 3), representing latest stream data at the ninth step. Hence, denial many consecutive instances of baseaccess(U, denied, T), needed to close the account of a user U, correspond to respective facts over access/3 in the current window.
3.2 Overtaking Maneuver Recognition

Our second scenario deals with recognizing the completion of overtaking maneuvers by a car, e.g. for signaling it to the driver. The recognition follows the transitions of the automaton in Figure 1. Starting from state ∅, representing that a maneuver has not yet been initiated, sensor information about being “behind,” “nextto,” or “infront” of another car enables transitions to corresponding states B, N, and F. As indicated by the output “OT!” in F, an overtaking maneuver is completed when F is reached from ∅ via a sequence of “behind,” “nextto,” and “infront” signals. Additional ε transitions model the progression from one time point to the next in the absence of signals: while such transitions are neutral in states ∅, B, and N, the final state F is abandoned (after outputting “OT!”). For instance, the automaton in Figure 1 admits the following trajectory (indicating a state at time point i by “@i” and providing signals in-between states):

(∅@0, behind, B@1, ε, B@2, nextto, N@3, infront, F@4, ε, ∅@5, nextto, ∅@6, behind, B@7, nextto, N@8, ε, N@9).

Here, an overtaking maneuver is completed when F is reached at time 4, and ε transitions preserve B and N, but not F. In the following, we consider overtaking maneuvers that are completed in at most 6 steps; that is, for a given time point i, the automaton in Figure 1 is assumed to start from ∅ at time i−6.

Similar to the static access control encoding in Listing 3, our encoding of overtaking maneuver recognition, shown in Listing 4, uses modulo arithmetic to map time stamps in stream data to a corresponding slot of atoms and rules provided in the #base program part. In more detail, transient (external) facts of the form at(P,C,T) (in which P is behind, nextto, or infront and C refers to a red, blue, or green car) are matched with corresponding instances of baseat/3 by means of the #cumulative and #volatile parts in Line 20–24. As before, these parts implement a transparent shift from steps i to i+modulo, provided that transient stream data is given in

5 Unlike in this simple example scenario, transition systems are usually described compactly in terms of state variables and operators on them, e.g. defined via action languages (cf. [21]). The automaton induced by a compact description can be of exponential size, and an explicit representation like in Figure 1 is often inaccessible in practice.
Listing 4. Static overtaking maneuver recognition encoding

```
#const modulo = 6.
#iinit 2-modulo.
#base.
position(behind; nextto; infront).  # relative positions
car(red; blue; green).  # some cars
time(0..modulo-1).  # time slots

next(T, (T+1) mod modulo) :- time(T), not now(T).
{ now(T) : time(T) } 1.
{ base(T, C, T) : position(P) : car(C) : time(T) }.
baseat(C, T) :- baseat(_, C, T).
state(behind, C, T) :- baseat(behind, C, T).
state(nextto, C, S) :- baseat(nextto, C, S), next(T, S),
{ state(behind, C, T), state(nextto, C, T) }.
state(infront, C, S) :- baseat(infront, C, S), now(S),
next(T, S), state(nextto, C, T).
state(P, C, S) :- state(P, C, T), P != infront,
next(T, S), not baseat(C, S).

#cumulative t.
#external at(P, C, t) : position(P) : car(C).
{ at(P, C, t), not baseat(P, C, t) #mod modulo }.
#volatile t : modulo.
{ base(P, C, (t*modulo) #mod modulo), not at(P, C, t) }.
#volatile t.
{ not now(t #mod modulo). }
```

"#volatile : modulo." blocks. Then, the rules in Line 12–16 model state transitions based on signals, and the one in Line 17–18 implements ε transitions (making use of projection in Line 10) as specified by the automaton in Figure 1. In particular, the completion of an overtaking maneuver in the current step, indicated by deriving infront as state for a car C and a time slot S via the rule in Line 15–16, relies on now(S) (explained below). Also note that state ∅ is left implicit, i.e. it applies to a car C and a slot T if baseat(P, C, T) does not hold for any relative position P, and that the frame axioms represented in Line 17–18 do not apply to infront states.

While a next/2 predicate had also been defined in Listing 3 to arrange the time slots of the #base program in a cycle, the corresponding rule in Line 7 of Listing 4 relies on the absence of now(T) for linking a time slot T to "(T+1) mod modulo." In fact, instances of now/1 are provided by the choice rule in Line 8 and synchronized with the incremental step counter t via the integrity constraint ":- not now(t #mod modulo)." of life span 1 (cf. Line 25–26). Unlike the previous approach to access counting (introducing an empty slot), making the current time slot explicit enables the linearization of a time slot cycle also in the presence of frame axioms, which could propagate into the "past" otherwise. In fact, if prerequisites regarding now/1 were dropped in Line 7 and 15, one could, beginning with at(behind, C, 7) as input and its corresponding atom baseat(behind, C, 1), derive state(infront, C, 4) at step 7 for a car C subject to the trajectory given above. Such a conclusion is clearly unintended, and the technique in Line 7–8 and 25–26, using now/1 to linearize a time slot cycle, provides a general solution for this issue. Finally, "#iinit 2-modulo." is again included in Line 1 to avoid initial guesses over baseat/3 wrt. (non-existing) past signals.
3.3 Online Job Scheduling

After inspecting straightforward data evaluation tasks, we now turn to a combinatorial problem in which job requests of different durations must be scheduled to machines without overlapping one another. Unlike in offline job scheduling [22], where requests are known in advance, we here assume a stream of job requests, provided via (transient) facts $\text{job}(I,M,D,T)$ such that $I$ is a job ID, $M$ is a machine, $D$ is a duration, and $T$ is the arrival time of a request. In addition, we assume a deadline $T+\max\_\text{step}$, for some integer constant $\max\_\text{step}$, by which the execution of a job $I$ submitted at step $T$ must be completed. For instance, an (initial) segment of a job request stream can be as follows:

- (step 1) $T = 0$: $\text{job}(1, 1, 1, 1)$. $\text{job}(2, 1, 5, 1)$. $\text{job}(3, 1, 5, 1)$. $\text{job}(4, 1, 5, 1)$. $\text{job}(5, 1, 5, 1)$.
- (step 21) $T = 21$: $\text{job}(1, 1, 5, 21)$. $\text{job}(2, 1, 5, 21)$. $\text{job}(3, 1, 5, 21)$. $\text{job}(4, 1, 5, 21)$.

That is, five jobs with ID $1$ to $5$ (of durations 1 and 5) are submitted at step 1 and ought to be completed on machine 1 within the deadline $1+\max\_\text{step} = 21$ (taking $\max\_\text{step} = 20$). Four more jobs with ID 1 to 4 of duration 5, submitted at step 21, also need to be executed on machine 1. As a matter of fact, a schedule to finish all jobs within their deadlines must first launch the five jobs submitted at step 1, thus occupying machine 1 at time points up to 21, before the other jobs can use machine 1 from time point 22 to 41. However, when a time-decaying logic program does not admit any answer set at some step (i.e., if there is no schedule meeting all deadlines), the default behavior of oclingo is to increase the incremental step counter until an answer set is obtained. This behavior would lead to the expiration of pending job requests, so that a schedule generated in turn lacks part of the submitted jobs. Since such (partial) schedules are unintended here, we take advantage of the enriched directive "$\#\text{step} i : \delta\$" to express that increases of the step counter must not exceed $i+\delta$, regardless of whether an answer set has been obtained at step $i+\delta$ (or some greater step). In fact, since $\delta = 0$ is used above, oclingo does not increase the step counter beyond $i$, but rather returns "unsatisfiable" as result if there is no answer set.

In Line 1–4, our (static) job scheduling encoding, shown in Listing 5, defines the viable durations, IDs of jobs requested per step, and the available machines in terms of corresponding constants. Furthermore, deadlines for the completion of jobs are obtained by adding $\max\_\text{step}$ (set to 20 in Line 2) to request submission times. As a consequence, jobs with submission times $i \leq j$ such that $j \leq i+\max\_\text{step}$ may need to be scheduled jointly, and the minimum window width $\text{modulo}$ required to accommodate the (maximum) completion times of jointly submitted jobs is calculated accordingly in Line 2. Given the value of $\text{modulo}$, the time slots of the $\#\text{base}$ program part are in Line 5 again arranged in a cycle (similar to access counting in Listing 3). The technique applied in Line 15–19 to map job requests given by transient (external) facts over $\text{job}/4$ to corresponding instances of $\text{basejob}/4$, provided by the choice rule in Line 7, remains the same as in previous static encodings (cf. Listing 3 and 4). But note that job IDs can be shared between jobs submitted at different steps, so that pairs $(I,T)$ of an ID $I$ and a slot $T$ identify job requests uniquely in the sequel.

The rules in Line 8–13 of the $\#\text{base}$ program accomplish the non-overlapping scheduling of submitted jobs such that they are completed within their deadlines. In fact,
### Listing 5. Static online job scheduling encoding

```prolog
#const max_duration = 5. #const max_jobid = 5. #const num_machines = 5.
#const max_step = 20. #const modulo = 2*max_step+1. #init & modulo.
#base.
duration(1..max_duration), jobid(1..max_jobid), machine(1..num_machines).
next(T, (T+1) #mod modulo) :- T := 0..modulo-1.
{ basejob(I,M,D,T) : jobid(I) : machine(M) : duration(D) : next(T,_). } 1.
{ jobstart(I,T,(T..T+max_step+1-D) #mod modulo) } 1 :- basejob(I,_,D,T).
occupy(M,I,T,D, S) :- basejob(I,M,D,T), jobstart(I,T,S).
occupy(M,I,T,D-1,S) :- occupy(M,I,T,D,R), next(R,S), D > 1.
occupy(M,I,T,S) :- occupy(M,I,T,_,S).
:- occupy(M,I1,T1,S), occupy(M,I2,T2,S), (I1,T1) < (I2,T2).
#cumulative t. #external job(I,M,D,t) : jobid(I) : machine(M) : duration(D).
:- job(I,M,D,t), not basejob(I,M,D, t #mod modulo).
#volatile t : modulo.
:- basejob(I,M,D, (t+modulo) #mod modulo), not job(I,M,D,t).
```

The choice rule in Line 8 expresses that a job of duration \( D \) with submission time slot \( T \) must be launched such that its execution finishes at slot \( (T+\text{max}\_\text{step}) \mod \text{modulo} \) (at the latest). Given the slots at which jobs are started, the rules in Line 10–12 propagate the occupation of machines \( M \) wrt. durations \( D \), and the integrity constraint in Line 13 makes sure that the same machine is not occupied by distinct jobs at the same time slot. For instance, the following atoms of an answer set represent starting times such that the jobs requested in the stream segment given above do not overlap and are processed within their deadlines:

- \( \text{jobstart}(1,1,1) \)
- \( \text{jobstart}(2,1,2) \)
- \( \text{jobstart}(3,1,7) \)
- \( \text{jobstart}(4,1,12) \)
- \( \text{jobstart}(5,1,17) \)
- \( \text{jobstart}(1,21,22) \)
- \( \text{jobstart}(2,21,27) \)
- \( \text{jobstart}(3,21,32) \)
- \( \text{jobstart}(4,21,37) \)

Note that the execution of the five jobs submitted at step 1 is finished at their common deadline 21, and the same applies wrt. the deadline 41 of jobs submitted at step 21. Since machine 1 is occupied at all time slots, executing all jobs within their deadlines would no longer be feasible if a job request like \( \text{job}(5,1,1,21) \) were added. In such a case, \textit{oclingo} outputs “unsatisfiable” and waits for new online input, which may shift the window and relax the next query due to the expiration of some job requests.

A future extension of \textit{oclingo} regards optimization (via \texttt{#minimize/#maximize}) wrt. online data, given that solutions violating as few (soft) constraints as possible may be more helpful than just reporting unsatisfiability. In either mode of operation, the static representation of problems over a window of fixed width, illustrated in Listing 3, 4, and 5, enables the re-use of constraints learned upon solving a query for answering further queries asked later on.

Although \textit{oclingo} is still in a prototypical state, we performed some preliminary experiments in order to give an indication of the impact of different encoding variants. As the first two example scenarios model pure data evaluation tasks (not requiring search), experiments with them did not exhibit significant runtimes, and we thus focus on re-
results for online job scheduling. In particular, we assess *oclingo* on the static encoding in Listing 5 as well as a cumulative variant (analog to the cumulative access control encoding in Listing 2); we further consider the standard ASP system *clingo*, processing each query independently via relaunching wrt. the current window contents. Table 2(a) provides average runtimes of the investigated configurations in seconds, grouped by satisfiable (∅S) and unsatisfiable (∅U) queries, on 12 randomly generated data streams with 200 online inputs each. These streams vary in the values used for constants, eg. \(\text{max_jobid} = 5\), \(\text{max_duration} = 3\), \(\text{num_machines} = 5\), and \(\text{max_step} = 15\) with the three “5x3x5,15,1” streams, and the respective numbers of satisfiable (#S) and unsatisfiable (#U) queries. First of all, we observe that the current prototype version of *oclingo* cannot yet compete with *clingo*. The reason for this is that *oclingo*’s underlying grounding and solving components were not designed with expiration in mind, so that they currently still remember the names of expired atoms (while irrelevant constraints referring to them are truly deleted). The resulting low-level overhead in each step explains the advantage of relaunching *clingo* from scratch. When comparing *oclingo*’s performance wrt. encoding variants, the static encoding appears to be generally more effective than its cumulative counterpart, albeit some unsatisfiable queries stemming from the last three example streams in Table 2(a) are solved faster using the latter.

The plot in Figure 2(b) provides a more fine-grained picture by displaying runtimes for individual queries from stream “5x3x5,15,1,” where small bars on the x-axis indicate unsatisfiable queries. While the static encoding yields a greater setup time of *oclingo* at the very beginning, it afterwards dominates the cumulative encoding variant, which requires the instantiation and integration of rules unrolling the horizons of new job requests at each step. Unlike this, the static encoding merely maps input atoms to their representations in the #base part, thus also solving each query wrt. the same (static) set of atoms. As a consequence, after initial unsatisfiable queries (yielding spikes in all configurations’ runtimes), *oclingo* with the static encoding is sometimes able to outperform *clingo* for successive queries remaining unsatisfiable. In fact, when the initial reasons for unsatisfiability remain in the window, follow-up queries are rather easy

<table>
<thead>
<tr>
<th>Stream</th>
<th>#S</th>
<th>#U</th>
<th>Static #S</th>
<th>Static #U</th>
<th>Cumulative #S</th>
<th>Cumulative #U</th>
<th>Relaunch #S</th>
<th>Relaunch #U</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3x3,9,1</td>
<td>162</td>
<td>33</td>
<td>0.02</td>
<td>0.01</td>
<td>0.05</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>3x3x3,9,2</td>
<td>162</td>
<td>33</td>
<td>0.02</td>
<td>0.01</td>
<td>0.05</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>3x3x3,9,3</td>
<td>147</td>
<td>53</td>
<td>0.02</td>
<td>0.01</td>
<td>0.05</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>5x3x5,15,1</td>
<td>164</td>
<td>36</td>
<td>0.17</td>
<td>0.18</td>
<td>0.40</td>
<td>0.48</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>5x3x5,15,2</td>
<td>167</td>
<td>33</td>
<td>0.17</td>
<td>0.47</td>
<td>0.40</td>
<td>0.64</td>
<td>0.04</td>
<td>0.26</td>
</tr>
<tr>
<td>5x3x5,15,3</td>
<td>140</td>
<td>60</td>
<td>0.22</td>
<td>0.81</td>
<td>0.42</td>
<td>0.72</td>
<td>0.04</td>
<td>0.31</td>
</tr>
<tr>
<td>5x5x10,15,1</td>
<td>161</td>
<td>39</td>
<td>1.61</td>
<td>0.94</td>
<td>3.32</td>
<td>3.34</td>
<td>0.14</td>
<td>0.19</td>
</tr>
<tr>
<td>5x5x10,15,2</td>
<td>167</td>
<td>33</td>
<td>1.44</td>
<td>0.88</td>
<td>3.53</td>
<td>2.88</td>
<td>0.14</td>
<td>0.34</td>
</tr>
<tr>
<td>5x5x10,15,3</td>
<td>182</td>
<td>18</td>
<td>1.46</td>
<td>0.79</td>
<td>3.59</td>
<td>4.08</td>
<td>0.14</td>
<td>0.15</td>
</tr>
<tr>
<td>5x10x30,1</td>
<td>185</td>
<td>15</td>
<td>3.29</td>
<td>3.24</td>
<td>4.72</td>
<td>4.73</td>
<td>0.34</td>
<td>0.43</td>
</tr>
<tr>
<td>5x5x10,30,2</td>
<td>180</td>
<td>20</td>
<td>4.09</td>
<td>432.66</td>
<td>5.42</td>
<td>181.44</td>
<td>0.36</td>
<td>0.54</td>
</tr>
<tr>
<td>5x5x10,30,3</td>
<td>199</td>
<td>1</td>
<td>2.91</td>
<td>43.75</td>
<td>5.23</td>
<td>38.04</td>
<td>0.30</td>
<td>16.33</td>
</tr>
<tr>
<td>Total</td>
<td>2030</td>
<td>370</td>
<td>1.39</td>
<td>37.02</td>
<td>2.45</td>
<td>20.26</td>
<td>0.14</td>
<td>3.36</td>
</tr>
</tbody>
</table>
given the previously learned constraints, and we observed that some of these queries could actually be solved without any guessing.

4 Discussion

We have devised novel modeling approaches for continuous stream reasoning based on ASP, utilizing time-decaying logic programs to capture sliding window data in a natural way. While such data is transient and subject to routine expiration, we provided techniques to encode knowledge about potential window contents statically. This reduces the dynamic tasks of re-grounding and integrating rules of an offline encoding in view of new data to matching inputs to a corresponding internal representation. As a consequence, reasoning also concentrates on a fixed propositional representation (whose parts are selectively activated wrt. actual window contents), which enables the re-use of constraints gathered by conflict-driven learning. Although we illustrated modeling principles, including an approach to propagate frame axioms along time slot cycles, on toy domains only, the basic ideas are of general applicability. This offers interesting prospects for implementing knowledge-intense forms of stream reasoning, as required in application areas like ambient assisted living, robotics, or dynamic scheduling.

Our approach to ASP-based stream reasoning is prototypically implemented as an extension of the reactive ASP system oclingo, and preliminary experiments clearly show the need of improved low-level support of data expiration. In fact, we plan to combine the process of redesigning oclingo with the addition of yet missing functionalities, such as optimization in incremental and reactive settings. Future work also regards the consolidation of existing and the addition of further directives to steer incremental grounding and solving. For instance, beyond step-wise #cumulative and #volatile directives, we envisage #assert and #retract statements, as offered by Prolog (cf. [23]), to selectively (de)activate logic program parts. As with traditional ASP methods, the objective of future extensions is to combine high-level declarative modeling with powerful reasoning technology, automating both grounding and search. The investigation of sliding window scenarios performed here provides a first step towards gearing ASP to continuous reasoning tasks. Presupposing appropriate technology support, we think that many dynamic domains may benefit from ASP-based reasoning.

Acknowledgments We are grateful to the anonymous referees for helpful comments. This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-1/2 and by the European Commission within EasyReach (www.easyreach-project.eu) under grant AAL-2009-2-117.

References

3. oclingo: http://www.cs.uni-potsdam.de/oclingo
Two New Definitions of Stable Models of Logic Programs with Generalized Quantifiers

Joohyung Lee and Yunsong Meng

School of Computing, Informatics and Decision Systems Engineering
Arizona State University, Tempe, USA

Abstract. We present alternative definitions of the first-order stable model semantics and its extension to incorporate generalized quantifiers by referring to the familiar notion of a reduct instead of referring to the SM operator in the original definitions. Also, we extend the FLP stable model semantics to allow generalized quantifiers by referring to an operator that is similar to the SM operator. For a reasonable syntactic class of logic programs, we show that the two stable model semantics of generalized quantifiers are interchangeable.

1 Introduction

Most versions of the stable model semantics involve grounding. For instance, according to the FLP semantics from [1; 2], assuming that the domain is \{-1, 1, 2\}, program

\[
\begin{align*}
p(2) & \leftarrow \text{not SUM}(x:p(x)) < 2 \\
p(-1) & \leftarrow \text{SUM}(x:p(x)) > -1 \\
p(1) & \leftarrow p(-1)
\end{align*}
\]

is identified with its ground instance w.r.t the domain:

\[
\begin{align*}
p(2) & \leftarrow \text{not SUM}(\{-1:p(-1), 1:p(1), 2:p(2)\}) < 2 \\
p(-1) & \leftarrow \text{SUM}(\{-1:p(-1), 1:p(1), 2:p(2)\}) > -1 \\
p(1) & \leftarrow p(-1)
\end{align*}
\]

As described in [1], it is straightforward to extend the definition of satisfaction to ground aggregate expressions. For instance, set \{p(-1), p(1)\} does not satisfy the body of the first rule of (2), but satisfies the bodies of the other rules. The FLP reduct of program (2) relative to \{p(-1), p(1)\} consists of the last two rules, and \{p(-1), p(1)\} is its minimal model. Indeed, \{p(-1), p(1)\} is the only FLP answer set of program (2).

On the other hand, according to the semantics from [3], program (2) is identified with some complex propositional formula containing nested implications:

\[
\left(\neg((p(2) \rightarrow p(-1) \lor p(1)) \land (p(1) \land p(2) \rightarrow p(-1))) \land (p(-1) \land p(1) \land p(2) \rightarrow \bot)\right) \rightarrow p(2)
\]

\[
\land \left(p(-1) \rightarrow p(1) \lor p(2) \rightarrow p(-1)\right)
\]

\[
\land \left(p(-1) \rightarrow p(1)\right).
\]
Under the stable model semantics of propositional formulas [3], this formula has two answer sets: \{p(−1), p(1)\} and \{p(−1), p(1), p(2)\}. The relationship between the FLP and the Ferraris semantics was studied in [4, 5].

Unlike the FLP semantics, the definition from [3] is not applicable when the domain is infinite because it would require the representation of an aggregate expression to involve “infinite” conjunctions and disjunctions. This limitation was overcome in the semantics presented in [4, 6], which extends the first-order stable model semantics from [7, 8] to incorporate aggregate expressions. Recently, it was further extended to formulas involving generalized quantifiers [9], which provides a unifying framework of various extensions of the stable model semantics, including programs with aggregates, programs with abstract constraint atoms [10], and programs with nonmonotonic dl-atoms [11].

In this paper, we revisit the first-order stable model semantics and its extension to incorporate generalized quantifiers. We provide an alternative, equivalent definition of a stable model by referring to grounding and reduct instead of the SM operator. Our work is inspired by the work of Truszczynski [12], who introduces infinite conjunctions and disjunctions to account for grounding quantified sentences. Our definition of a stable model can be viewed as a reformulation and a further generalization of his definition to incorporate generalized quantifiers. We define grounding in the same way as done in the FLP semantics, but define a reduct differently so that the semantics agrees with the one by Ferraris [3]. As we explain in Section 3.3, our reduct of program (2) relative to \{p(−1), p(1)\} is

\[
\begin{align*}
\bot & \leftarrow \bot \\
p(−1) & \leftarrow \text{SUM}\{\{-1:p(−1), 1:p(1), 2:\bot\}\} > −1 \\
p(1) & \leftarrow p(−1),
\end{align*}
\] (3)

which is the program obtained from (2) by replacing each maximal subformula that is not satisfied by \{p(−1), p(1)\} with \bot. Set \{p(−1), p(1)\} is an answer set of program (1) as it is a minimal model of the reduct. Likewise the reduct relative to \{p(−1), p(1), p(2)\} is

\[
\begin{align*}
p(2) & \leftarrow \top \\
p(−1) & \leftarrow \text{SUM}\{\{-1:p(−1), 1:p(1), 2:p(2)\}\} > −1 \\
p(1) & \leftarrow p(−1)
\end{align*}
\]

and \{p(−1), p(1), p(2)\} is a minimal model of the program. The semantics is more direct than the one from [3] as it does not involve the complex translation into a propositional formula.

While the FLP semantics in [1] was defined in the context of logic programs with aggregates, it can be straightforwardly extended to allow other “complex atoms.” Indeed, the FLP reduct is the basis of the semantics of HEX programs [13]. In [14], the FLP reduct was applied to provide a semantics of nonmonotonic dl-programs [11]. In [5], the FLP semantics of logic programs with aggregates was generalized to the first-order level. That semantics is defined in terms of the FLP operator, which is similar to the SM operator. This paper further extends the definition to allow generalized quantifiers.

By providing an alternative definition in the way that the other semantics was defined, this paper provides a useful insight into the relationship between the first-order semantics.
stable model semantics and the FLP stable model semantics for programs with generalized quantifiers. While the two semantics behave differently in the general case, we show that they coincide on some reasonable syntactic class of logic programs. This implies that an implementation of one of the semantics can be viewed as an implementation of the other semantics if we limit attention to that class of logic programs.

The paper is organized as follows. Section 2 reviews the first-order stable model semantics and its equivalent definition in terms of grounding and reduct, and Section 3 extends that definition to incorporate generalized quantifiers. Section 4 provides an alternative definition of the FLP semantics with generalized quantifiers via a translation into second-order formulas. Section 5 compares the FLP semantics and the first-order stable model semantics in the general context of programs with generalized quantifiers.

2 First-Order Stable Model Semantics

2.1 Review of First-Order Stable Model Semantics

This review follows [8], a journal version of [7], which distinguishes between intensional and non-intensional predicates.

A formula is defined the same as in first-order logic. A signature consists of function constants and predicate constants. Function constants of arity 0 are also called object constants. We assume the following set of primitive propositional connectives and quantifiers: 

\[ \bot, \top, \land, \lor, \rightarrow, \forall, \exists. \]

\( \neg F \) is an abbreviation of \( F \rightarrow \bot \), and \( F \leftrightarrow G \) stands for \( (F \rightarrow G) \land (G \rightarrow F) \). We distinguish between atoms and atomic formulas as follows: an atom of a signature \( \sigma \) is an \( n \)-ary predicate constant followed by a list of \( n \) terms that can be formed from function constants in \( \sigma \) and object variables; atomic formulas of \( \sigma \) are atoms of \( \sigma \), equalities between terms of \( \sigma \), and the 0-place connectives \( \bot \) and \( \top \).

The stable models of \( F \) relative to a list of predicates \( p = (p_1, \ldots, p_n) \) are defined via the stable model operator with the intensional predicates \( p \), denoted by \( \text{SM}[F; p] \).

Let \( u \) be a list of distinct predicate variables \( u_1, \ldots, u_n \). By \( u = p \) we denote the conjunction of the formulas \( \forall x (u_i(x) \leftrightarrow p_i(x)) \), where \( x \) is a list of distinct object variables of the same length as the arity of \( p_i \), for all \( i = 1, \ldots, n \). By \( u \leq p \) we denote the conjunction of the formulas \( \forall x (u_i(x) \rightarrow p_i(x)) \) for all \( i = 1, \ldots, n \), and \( u < p \) stands for \( (u \leq p) \land \neg(u = p) \). For any first-order sentence \( F \), expression \( \text{SM}[F; p] \) stands for the second-order sentence

\[ F \land \neg \exists u((u < p) \land F^*(u)), \]

where \( F^*(u) \) is defined recursively:

- \( p_i(t)^* = u_i(t) \) for any list \( t \) of terms;
- \( F^* = F \) for any atomic formula \( F \) that does not contain members of \( p \);
- \( (F \land G)^* = F^* \land G^* \);

\(^1\) The intensional predicates \( p \) are the predicates that we “intend to characterize” by \( F \).
A model of a sentence $F$ (in the sense of first-order logic) is called $p$-stable if it satisfies $SM[F; p]$.

**Example 1** Let $F$ be sentence $\forall x(\neg p(x) \rightarrow q(x))$, and let $I$ be an interpretation whose universe is the set of all nonnegative integers $\mathbb{N}$, and $p^I(n) = \text{FALSE}$, $q^I(n) = \text{TRUE}$ for all $n \in \mathbb{N}$. Section 2.4 of [8] tells us that $I$ satisfies $SM[F; pq]$.

### 2.2 Alternative Definition of First-Order Stable Models via Reduct

For any signature $\sigma$ and its interpretation $I$, by $\sigma^I$ we mean the signature obtained from $\sigma$ by adding new object constants $\xi^o$, called object names, for every element $\xi$ in the universe of $I$. We identify an interpretation $I$ of $\sigma$ with its extension to $\sigma^I$ defined by $I(\xi^o) = \xi$.

In order to facilitate defining a reduct, we provide a reformulation of the standard semantics of first-order logic via "a ground formula w.r.t. an interpretation."

**Definition 1.** For any interpretation $I$ of a signature $\sigma$, a ground formula $w.r.t.$ $I$ is defined recursively as follows.

- $p(\xi_1^o, \ldots, \xi_n^o)$, where $p$ is a predicate constant of $\sigma$ and $\xi_i^o$ are object names of $\sigma^I$, is a ground formula $w.r.t.$ $I$;
- $\top$ and $\bot$ are ground formulas $w.r.t.$ $I$;
- If $F$ and $G$ are ground formulas $w.r.t.$ $I$, then $F \land G, F \lor G, F \rightarrow G$ are ground formulas $w.r.t.$ $I$;
- If $S$ is a set of pairs of the form $\xi^o: F$ where $\xi^o$ is an object name in $\sigma^I$ and $F$ is a ground formula $w.r.t.$ $I$, then $\forall(S)$ and $\exists(S)$ are ground formulas $w.r.t.$ $I$.

The following definition describes a process that turns any first-order sentence into a ground formula $w.r.t.$ an interpretation:

**Definition 2.** Let $F$ be any first-order sentence of a signature $\sigma$, and let $I$ be an interpretation of $\sigma$ whose universe is $U$. By $gr^I[F]$ we denote the ground formula $w.r.t.$ $I$, which is obtained by the following process:

- $gr^I[p(t_1, \ldots, t_n)] = p((t_1^I)^\circ, \ldots, (t_n^I)^\circ)$;
- $gr^I[t_1 = t_2] = \begin{cases} \top & \text{if } t_1^I = t_2^I, \\
\bot & \text{otherwise} \end{cases}$;
- $gr^I[\top] = \top$; $gr^I[\bot] = \bot$;
- $gr^I[F \circ G] = gr^I[F] \circ gr^I[G]$ (where $\circ \in \{\land, \lor, \rightarrow\}$);
- $gr^I[QxF(x)] = Q(\{\xi^o: gr^I[F(\xi^o)] \mid \xi \in U\})$ (where $Q \in \{\forall, \exists\}$).

**Definition 3.** For any interpretation $I$ and any ground formula $F$ $w.r.t.$ $I$, the truth value of $F$ under $I$, denoted by $F^I$, is defined recursively as follows.
- \( p(\xi_1^\circ, \ldots, \xi_n^\circ) I = p^I(\xi_1, \ldots, \xi_n) \);
- \( \top I = \text{TRUE}; \quad \bot I = \text{FALSE}; \)
- \( (F \land G) I = \text{TRUE} \) iff \( F I = \text{TRUE} \) and \( G I = \text{TRUE} \);
- \( (F \lor G) I = \text{TRUE} \) iff \( F I = \text{TRUE} \) or \( G I = \text{TRUE} \);
- \( (F \rightarrow G) I = \text{TRUE} \) whenever \( F I = \text{TRUE} \);
- \( \forall(S) I = \text{TRUE} \) iff the set \( \{ \xi : F(\xi) \in S \text{ and } F(\xi I) = \text{TRUE} \} \) is not empty.

We say that \( I \) satisfies \( F \), denoted \( I \models F \), if \( F I = \text{TRUE} \).

**Example 1 continued (I).** \( \text{gr}_I[F] \) is \( \forall(\{n^\circ : (\neg p(n) \rightarrow q(n)) \mid n \in \mathbb{N}\}) \). Clearly, \( I \) satisfies \( \text{gr}_I[F] \).

An interpretation \( I \) of a signature \( \sigma \) can be represented as a pair \( \langle I^{\text{unc}}, I^{\text{pred}} \rangle \), where \( I^{\text{unc}} \) is the restriction of \( I \) to the function constants of \( \sigma \), and \( I^{\text{pred}} \) is the set of atoms, formed using predicate constants from \( \sigma \) and the object names from \( \sigma^I \), which are satisfied by \( I \). For example, interpretation \( I \) in Example 1 can be represented as \( \langle I^{\text{unc}}, \{ q(n^\circ) \mid n \in \mathbb{N}\} \rangle \), where \( I^{\text{unc}} \) maps each integer to itself.

The following proposition is immediate from the definitions:

**Proposition 1.** Let \( \sigma \) be a signature that contains finitely many predicate constants, let \( \sigma^{\text{pred}} \) be the set of predicate constants in \( \sigma \), let \( I = \langle I^{\text{unc}}, I^{\text{pred}} \rangle \) be an interpretation of \( \sigma \), and let \( F \) be a first-order sentence of \( \sigma \). Then \( I \models F \) iff \( I^{\text{pred}} \models \text{gr}_I[F] \).

The introduction of the intermediate form of a ground formula w.r.t. an interpretation helps us define a reduct.

**Definition 4.** For any ground formula \( F \) w.r.t. \( I \), the reduct of \( F \) relative to \( I \), denoted by \( F^\bot \), is obtained by replacing each maximal subformula that is not satisfied by \( I \) with \( \bot \). It can also be defined recursively as follows.

- \( (p(\xi_1^\circ, \ldots, \xi_n^\circ))^\bot = \begin{cases} p(\xi_1^\circ, \ldots, \xi_n^\circ) & \text{if } I \models p(\xi_1^\circ, \ldots, \xi_n^\circ), \\ \bot & \text{otherwise}; \end{cases} \)
- \( \top^\bot = \top; \quad \bot^\bot = \bot; \)
- \( (F \land G)^\bot = \begin{cases} F^\bot \land G^\bot & \text{if } I \models F \land G \quad (\circ \in \{\land, \lor, \rightarrow\}), \\ \bot & \text{otherwise}; \end{cases} \)
- \( Q(S)^\bot = \begin{cases} Q(\{\xi : F(\xi) \in S \}) & \text{if } I \models Q(S) \quad (Q \in \{\forall, \exists\}), \\ \bot & \text{otherwise}. \end{cases} \)

The following theorem tells us how first-order stable models can be characterized in terms of grounding and reduct.

**Theorem 1.** Let \( \sigma \) be a signature that contains finitely many predicate constants, let \( \sigma^{\text{pred}} \) be the set of predicate constants in \( \sigma \), let \( I = \langle I^{\text{unc}}, I^{\text{pred}} \rangle \) be an interpretation of \( \sigma \), and let \( F \) be a first-order sentence of \( \sigma \). \( I \) satisfies \( \text{SM}[F; \sigma^{\text{pred}}] \) iff \( I^{\text{pred}} \) is a minimal set of atoms that satisfies \( \langle \text{gr}_I[F]\rangle^\bot \).
Example 1 continued (II). The reduct of \( \text{gr}_I[F] \) relative to \( I \), \( (\text{gr}_I[F])^I \), is
\[
\forall(\{n^\circ: (\neg \bot \rightarrow q(n^\circ)) \mid n \in \mathbb{N}\}),
\]
which is equivalent to \( \forall(\{n^\circ: q(n^\circ) \mid n \in \mathbb{N}\}) \).
Clearly, \( I^{\text{red}} = \{q(n^\circ) \mid n \in \mathbb{N}\} \) is a minimal set of atoms that satisfies \( (\text{gr}_I[F])^I \).

2.3 Relation to Infinitary Formulas by Truszczynski

The definitions of grounding and a reduct in the previous section are inspired by the work of Truszczynski [12], where he introduces infinite conjunctions and disjunctions to account for the result of grounding \( \forall \) and \( \exists \) w.r.t. a given interpretation. Differences between the two approaches are illustrated in the following example:

Example 2 Consider the formula \( F = \forall x \ p(x) \) and the interpretation \( I \) whose universe is the set of all nonnegative integers \( \mathbb{N} \). According to [12], grounding of \( F \) w.r.t. \( I \) results in the infinitary propositional formula
\[
\{p(n^\circ) \mid n \in \mathbb{N}\}.
\]
On the other hand, formula \( \text{gr}_I[F] \) is
\[
\forall(\{n^\circ: p(n^\circ) \mid n \in \mathbb{N}\}).
\]

Our definition of a reduct is essentially equivalent to the one defined in [12]. In the next section, we extend our definition to incorporate generalized quantifiers.

3 Stable Models of Formulas with Generalized Quantifiers

3.1 Review: Formulas with Generalized Quantifiers

We follow the definition of a formula with generalized quantifiers from [15, Section 5] (that is to say, with Lindström quantifiers [16] without the isomorphism closure condition).

We assume a set \( Q \) of symbols for generalized quantifiers. Each symbol in \( Q \) is associated with a tuple of nonnegative integers \( \langle n_1, \ldots, n_k \rangle \) \( (k \geq 0, \text{and each } n_i \geq 0) \), called the type. A \( (GQ-) \)formula (with the set \( Q \) of generalized quantifiers) is defined in a recursive way:

- an atomic formula (in the sense of first-order logic) is a GQ-formula;
- if \( F_1, \ldots, F_k \) \( (k \geq 0) \) are GQ-formulas and \( Q \) is a generalized quantifier of type \( \langle n_1, \ldots, n_k \rangle \) in \( Q \), then
  \[
  Q[x_1] \ldots [x_k](F_1(x_1), \ldots, F_k(x_k)) \tag{4}
  \]
is a GQ-formula, where each \( x_i \) \( (1 \leq i \leq k) \) is a list of distinct object variables whose length is \( n_i \).
We say that an occurrence of a variable $x$ in a GQ-formula $F$ is bound if it belongs to a subformula of $F$ that has the form $Q[x_1] \ldots [x_k](F_1(x_1), \ldots, F_k(x_k))$ such that $x$ is in some $x_i$. Otherwise the occurrence is free. We say that $x$ is free in $F$ if $F$ contains a free occurrence of $x$. A (GQ-)sentence is a GQ-formula with no free variables.

We assume that $Q$ contains type $\langle \rangle$ quantifiers $Q_{\bot}$ and $Q_{\top}$, type $\langle 0,0 \rangle$ quantifiers $Q_{\land}, Q_{\lor}, Q_{\rightarrow}$, and type $\langle 1 \rangle$ quantifiers $Q_{\forall}, Q_{\exists}$. Each of them corresponds to the standard logical connectives and quantifiers — $\bot, \top, \land, \lor, \rightarrow, \forall, \exists$. These generalized quantifiers will often be written in the familiar form. For example, we write $F \land G$ in place of $Q_{\land}[(F,G)]$, and write $\forall x F(x)$ in place of $Q_{\forall}[x](F(x))$.

As in first-order logic, an interpretation $I$ consists of the universe $U$ and the evaluation of predicate constants and function constants. For each generalized quantifier $Q$ of type $\langle n_1, \ldots, n_k \rangle$, $Q^U$ is a function from $\mathcal{P}(U^{n_1}) \times \cdots \times \mathcal{P}(U^{n_k})$ to $\{ \text{true}, \text{false} \}$, where $\mathcal{P}(U^n)$ denotes the power set of $U^n$.

**Example 3** Besides the standard connectives and quantifiers, the following are some examples of generalized quantifiers.

- type $\langle 1 \rangle$ quantifier $Q_{\leq 2}$ such that $Q^U_{\leq 2}(R) = \text{true} \iff |R| \leq 2$;
- type $\langle 1 \rangle$ quantifier $Q_{\text{majority}}$ such that $Q^U_{\text{majority}}(R) = \text{true} \iff |R| > |U \setminus R|$;
- type $(1,1)$ quantifier $Q_{(\text{sum},<)}$ such that $Q^U_{(\text{sum},<)}(R_1, R_2) = \text{true}$
  - if $\text{SUM}(R_1)$ is defined,
  - $R_2 = \{ b \}$, where $b$ is an integer, and
  - $\text{SUM}(R_1) < b$.

Given a sentence $F$ of $\sigma^I$, $F^I$ is defined recursively as follows:

- $p(t_1, \ldots, t_n)^I = p^I(t_1^I, \ldots, t_n^I)$,
- $(t_1 = t_2)^I = (t_1^I = t_2^I)$,
- For a generalized quantifier $Q$ of type $\langle n_1, \ldots, n_k \rangle$,
  
  $$(Q[x_1] \ldots [x_k](F_1(x_1), \ldots, F_k(x_k))^I = Q^U(⟨x_1 : F_1(x_1))^I, \ldots, (x_k : F_k(x_k))^I),$$

where $(x_i : F_i(x_i))^I = \{ \xi \in U^{n_i} \mid (F_i(\xi^x))^I = \text{true}\}$.

We assume that, for the standard logical connectives and quantifiers $Q$, functions $Q^U$ have the standard meaning:

- $Q^U_{\bot}(R) = \text{true} \iff R = U$; 
- $Q^U_{\top}(R) = \text{true} \iff R \cap U \neq \emptyset$; 
- $Q^U_{\land}(R_1, R_2) = \text{true} \iff R_1 = R_2 = \{ \epsilon \}$; 
- $Q^U_{\lor}(R_1, R_2) = \text{true} \iff R_1 = \{ \epsilon \}$ or $R_2 = \{ \epsilon \}$; 
- $Q^U_{\rightarrow}(R_1, R_2) = \text{false} \iff R_1 = \emptyset$ or $R_2 = \{ \epsilon \}$; 
- $Q^U_{\forall}() = \text{true}$; 
- $Q^U_{\exists}() = \text{false}$.

\(^2\) It is clear from the type of the quantifier that $R$ is any subset of $U$. We will skip such explanation.

\(^3\) $\epsilon$ denotes the empty tuple. For any interpretation $I$, $U^0 = \{ \epsilon \}$. For $I$ to satisfy $Q_{\land}[(F,G)]$, both $(\epsilon : F)^I$ and $(\epsilon : G)^I$ have to be $\{ \epsilon \}$, which means that $F^I = G^I = \text{true}$.
We say that an interpretation $I$ satisfies a GQ-sentence $F$, or is a model of $F$, and write $I \models F$, if $F_I = \text{TRUE}$. A GQ-sentence $F$ is logically valid if every interpretation satisfies $F$. A GQ-formula with free variables is said to be logically valid if its universal closure is logically valid.

**Example 4** Program (1) in the introduction is identified with the following GQ-formula:

$\neg Q_{\text{SUM},<}[x][y](p(x), y = 2) \rightarrow p(2)$

$\land (Q_{\text{SUM},>}[x][y](p(x), y = -1) \rightarrow p(-1))$

$\land (p(-1) \rightarrow p(1))$.

Consider two Herbrand interpretations of the universe $U = \{-1, 1, 2\}$: $I_1 = \{p(-1), p(1)\}$ and $I_2 = \{p(-1), p(1), p(2)\}$. We have $(Q_{\text{SUM},<}[x][y](p(x), y = 2))^{I_1} = \text{TRUE}$ since

- $(x : p(x))^{I_1} = \{-1, 1\}$ and $(y : y = 2)^{I_1} = \{2\}$;
- $Q_{\text{SUM},<}^U(\{-1, 1\}, \{2\}) = \text{TRUE}.$

Similarly, $(Q_{\text{SUM},>}[x][y](p(x), y = -1))^{I_2} = \text{TRUE}$ since

- $(x : p(x))^{I_2} = \{-1, 1, 2\}$ and $(y : y = -1)^{I_2} = \{-1\}$;
- $Q_{\text{SUM},>}^U(\{-1, 1, 2\}, \{-1\}) = \text{TRUE}.$

Consequently, both $I_1$ and $I_2$ satisfy $F_1$.

### 3.2 Review: SM-Based Definition of Stable Models of GQ-Formulas

For any GQ-formula $F$ and any list of predicates $p = (p_1, \ldots, p_n)$, formula $\text{SM}[F; p]$ is defined as

$$F \land \neg \exists u((u < p) \land F^*(u)),$$

where $F^*(u)$ is defined recursively:

- $p_i(t)^* = u_i(t)$ for any list $t$ of terms;
- $F^* = F$ for any atomic formula $F$ that does not contain members of $p$;
- $(Q[x_1] \ldots [x_k](F_1(x_1), \ldots, F_k(x_k)))^* = Q[x_1] \ldots [x_k](F_1^*(x_1), \ldots, F_k^*(x_k)) \land Q[x_1] \ldots [x_k](F_1(x_1), \ldots, F_k(x_k))$.

When $F$ is a sentence, the models of $\text{SM}[F; p]$ are called the $p$-stable models of $F$: they are the models of $F$ that are “stable” on $p$. We often simply write $\text{SM}[F]$ in place of $\text{SM}[F; p]$ when $p$ is the list of all predicate constants occurring in $F$, and call $p$-stable models simply stable models.

As explained in [17], this definition of a stable model is a proper generalization of the first-order stable model semantics.

**Example 4 continued (I).** For GQ-sentence $F_1$ considered earlier, $\text{SM}[F_1]$ is

$$F_1 \land \neg \exists u(u < p \land F_1^*(u)),$$

(5)
where $F_1^*(u)$ is equivalent to the conjunction of $F_1$ and
\[
(-Q_{(\text{SUM,} <)}[x][y](p(x), y = 2) \rightarrow u(2)) \\
\land (Q_{(\text{SUM,} >)}[x][y](u(x), y = -1) \land Q_{(\text{SUM,} >)}[x][y](p(x), y = -1)) \rightarrow u(-1)) \\
\land (u(-1) \rightarrow u(1)).
\]

The equivalence can be explained by Proposition 1 from [9], which simplifies the transformation for monotone and antimonotone GQs. $I_1$ and $I_2$ considered earlier satisfy (5) and thus are stable models of $F_1$.

### 3.3 Reduct-Based Definition of Stable Models of GQ-Formulas

The reduct-based definition of stable models presented in Section 2.2 can be extended to GQ-formulas as follows.

Let $I$ be an interpretation of a signature $\sigma$. As before, we assume a set $Q$ of generalized quantifiers, which contains all propositional connectives and standard quantifiers.

**Definition 5.** A ground GQ-formula w.r.t. $I$ is defined recursively as follows:

- $p(\xi_1, \ldots, \xi_n)$, where $p$ is a predicate constant of $\sigma$ and $\xi_i$ are object names of $\sigma^I$, is a ground GQ-formula w.r.t. $I$;
- $Q \in Q$ of type $\langle n_1, \ldots, n_k \rangle$, if each $S_i$ is a set of pairs of the form $\xi^S: F$ where $\xi^S$ is a list of object names from $\sigma^I$ whose length is $n_i$, and $F$ is a ground GQ-formula w.r.t. $I$, then
  \[ Q(S_1, \ldots, S_k) \]
  is a ground GQ-formula w.r.t. $I$.

The following definition of grounding turns any GQ-sentence into a ground GQ-formula w.r.t. an interpretation:

**Definition 6.** Let $F$ be a GQ-sentence of a signature $\sigma$, and let $I$ be an interpretation of $\sigma$. By $gr_I[F]$ we denote the ground GQ-formula w.r.t. $I$ that is obtained by the process similar to the one in Definition 2 except that the last two clauses are replaced by the following single clause:

- $gr_I[Q[x_1] \ldots [x_k][F_1(x_1), \ldots, F_k(x_k)]] = Q(S_1, \ldots, S_k)$ where $S_i = \{ \xi^S: gr_I[F_i(\xi^S)] \mid \xi^S \text{ is a list of object names from } \sigma^I \text{ whose length is } n_i \}$.

For any interpretation $I$ and any ground GQ-formula $F$ w.r.t. $I$, the satisfaction relation $I \models F$ is defined recursively as follows.

**Definition 7.** For any interpretation $I$ and any ground GQ-formula $F$ w.r.t. $I$, the satisfaction relation $I \models F$ is defined similar to Definition 3 except that the last five clauses are replaced by the following single clause:

- $Q(S_1, \ldots, S_k)^I = Q^{U}(S_1^I, \ldots, S_k^I)$ where $S_i^I = \{ \xi \mid \xi^S \in S_i, F(\xi^S)^I = \text{TRUE} \}$. 

Title Suppressed Due to Excessive Length 139
Example 4 continued (II). For Herbrand interpretation \( I_1 = \{ p(-1), p(1) \} \), formula \( \text{gr}_{I_1}[F_1] \) is 4
\[
\neg Q_{(\text{SUM}, \langle \rangle)}(\{-1 : p(-1), 1 : p(1), 2 : p(2)\}, \{-1 : \bot, 1 : \bot, 2 : \top\}) \rightarrow p(2) \\
\wedge (Q_{(\text{SUM}, \langle \rangle)}(\{-1 : p(-1), 1 : p(1), 2 : p(2)\}, \{-1 : \top, 1 : \bot, 2 : \bot\}) \rightarrow p(-1)) \\
\wedge (p(-1) \rightarrow p(1)).
\]

\( I_1 \) satisfies \( Q_{(\text{SUM}, \langle \rangle)}(\{-1 : p(-1), 1 : p(1), 2 : p(2)\}, \{-1 : \bot, 1 : \bot, 2 : \top\}) \) because \( I_1 \models p(-1), I_1 \models p(1), I_1 \not\models p(2) \), and
\[
Q_{(\text{SUM}, \langle \rangle)}(\{-1, 1\}, \{2\}) = \text{TRUE}.
\]

\( I_1 \) satisfies \( Q_{(\text{SUM}, \langle \rangle)}(\{-1 : p(-1), 1 : p(1), 2 : p(2)\}, \{-1 : \top, 1 : \bot, 2 : \bot\}) \) because
\[
Q_{(\text{SUM}, \langle \rangle)}(\{-1, 1\}, \{-1\}) = \text{TRUE}.
\]

Consequently, \( I_1 \) satisfies (6).

Proposition 2. Let \( \sigma \) be a signature that contains finitely many predicate constants, let \( \sigma^{\text{pred}} \) be the set of predicate constants in \( \sigma \), let \( I = \langle I^{\text{unc}}, I^{\text{pred}} \rangle \) be an interpretation of \( \sigma \), and let \( F \) be a GQ-sentence of \( \sigma \). Then \( I \models F \iff I^{\text{pred}} \models \text{gr}_I[F] \).

Definition 8. For any GQ-formula \( F \) w.r.t. \( I \), the reduct of \( F \) relative to \( I \), denoted by \( F^L \), is defined in the same way as in Definition 4 by replacing the last two clauses with the following single clause:
\[
- (Q(S_1, \ldots, S_k))^L = \begin{cases} 
Q(S_1^L, \ldots, S_k^L) & \text{if } I \models Q(S_1, \ldots, S_k), \\
\bot & \text{otherwise}; 
\end{cases}
\]
where \( S_i^L = \{ \xi^F : (F(\xi^F))^L \models \xi^F : F(\xi^F) \in S_i \} \).

Theorem 2. Let \( \sigma \) be a signature that contains finitely many predicate constants, let \( \sigma^{\text{pred}} \) be the set of predicate constants in \( \sigma \), let \( I = \langle I^{\text{unc}}, I^{\text{pred}} \rangle \) be an interpretation of \( \sigma \), and let \( F \) be a GQ-sentence of \( \sigma \). \( I \models \text{SM}[F; \sigma^{\text{pred}}] \iff I^{\text{pred}} \) is a minimal set of atoms that satisfies \( (\text{gr}_I[F])^L \).

Example 4 continued (III). Interpretation \( I_1 \) considered earlier can be identified with the tuple \( \langle I^{\text{unc}}, \{ p(-1), p(1) \} \rangle \) where \( I^{\text{unc}} \) maps every term to itself. The reduct \( (\text{gr}_{I_1}[F_1])^L \) is
\[
(\bot \rightarrow \bot) \\
\wedge (Q_{(\text{SUM}, \langle \rangle)}(\{-1 : p(-1), 1 : p(1), 2 : \bot\}, \{-1 : \top, 1 : \bot, 2 : \bot\}) \rightarrow p(-1)) \\
\wedge (p(-1) \rightarrow p(1)),
\]
which is the GQ-formula representation of (3). We can check that \( \{ p(-1), p(1) \} \) is a minimal model of the reduct.

Extending Theorem 2 to allow an arbitrary list of intensional predicates, rather than \( \sigma^{\text{pred}} \), is straightforward in view of Proposition 1 from [18].

\[4\] For simplicity, we write \(-1, 1, 2\) instead of their object names \((-1)^\diamond, 1^\diamond, 2^\diamond\).
4 FLP Semantics of Programs with Generalized Quantifiers

The FLP stable model semantics [1] is an alternative way to define stable models. It is the basis of HEX programs, an extension of the stable model semantics with higher-order and external atoms, which is implemented in system DLV-HEX. The first-order generalization of the FLP stable model semantics for programs with aggregates was given in [5], using the FLP operator that is similar to the SM operator. In this section we show how it can be extended to allow generalized quantifiers.

4.1 FLP Semantics of Programs with Generalized Quantifiers

A (general) rule is of the form

\[ H \leftarrow B \]  \hspace{1cm} (7)

where \( H \) and \( B \) are arbitrary GQ-formulas. A (general) program is a finite set of rules.

Let \( p \) be a list of distinct predicate constants \( p_1, \ldots, p_n \), and let \( u \) be a list of distinct predicate variables \( u_1, \ldots, u_n \). For any formula \( G \), formula \( G(u) \) is obtained from \( G \) by replacing all occurrences of predicates from \( p \) with the corresponding predicate variables from \( u \).

Let \( \Pi \) be a finite program whose rules have the form (7). The GQ-representation \( \Pi^{GQ} \) of \( \Pi \) is the conjunction of the universal closures of \( B \rightarrow H \) for all rules (7) in \( \Pi \). By FLP[\( \Pi; p \)] we denote the second-order formula

\[ \Pi^{GQ} \land \neg \exists u (u < p \land \Pi^{\triangle}(u)) \]

where \( \Pi^{\triangle}(u) \) is defined as the conjunction of the universal closures of

\[ B \land B(u) \rightarrow H(u) \]

for all rules \( H \leftarrow B \) in \( \Pi \).

We will often simply write FLP[\( \Pi \)] instead of FLP[\( \Pi; p \)] when \( p \) is the list of all predicate constants occurring in \( \Pi \), and call a model of FLP[\( \Pi \)] an FLP-stable model of \( \Pi \).

Example 4 continued (IV). For formula \( F_1 \) considered earlier, FLP[\( F_1 \)] is

\[ F_1 \land \neg \exists u (u < p \land F^{\triangle}_1(u)) \]  \hspace{1cm} (8)

where \( F^{\triangle}_1(u) \) is

\[
(-Q(SUM,>)[x][y](p(x), y=2) \land \neg Q(SUM,<)[x][y](u(x), y=2) \rightarrow u(2)) \\
\land (-Q(SUM,>)[x][y](p(x), y=-1) \land \neg Q(SUM,<)[x][y](u(x), y=-1) \rightarrow u(-1)) \\
\land (p(-1) \land u(-1) \rightarrow u(1)) .
\]

\( I_1 \) considered earlier satisfies (8) but \( I_2 \) does not.
5 Comparing the FLP Semantics and the First-Order Stable Model Semantics

In this section, we show a class of programs with GQs for which the FLP semantics and the first-order stable model semantics coincide.

The following definition is from [17]. We say that a generalized quantifier $Q$ is monote in the $i$-th argument position if the following holds for any universe $U$: if $Q^U(R_1, \ldots, R_k) = \text{TRUE}$ and $R_i \subseteq R'_i \subseteq U^n$, then

$$Q^U(R_1, \ldots, R_{i-1}, R'_i, R_{i+1}, \ldots, R_k) = \text{TRUE}.$$ 

Consider a program $\Pi$ consisting of rules of the form

$$A_1; \ldots; A_l \leftarrow E_1, \ldots, E_m, \text{not } E_{m+1}, \ldots, \text{not } E_n$$

($l \geq 0; n \geq m \geq 0$), where each $A_i$ is an atomic formula and each $E_i$ is an atomic formula or a GQ-formula (4) such that all $F_1(x_1), \ldots, F_k(x_k)$ are atomic formulas. Furthermore we require that, for every GQ-formula (4) in one of $E_{m+1}, \ldots, E_n$, $Q$ is monotone in all its argument positions.

**Proposition 3.** Let $\Pi$ be a program whose syntax is described as above, and let $F$ be the GQ-representation of $\Pi$. Then $\text{FLP}[\Pi; p]$ is equivalent to $\text{SM}[F; p]$.

**Example 5** Consider the following one-rule program:

$$p(a) \leftarrow \text{not } Q_{\leq 0}[x] p(x). \quad (9)$$

This program does not belong to the syntactic class of programs stated in Proposition 3 since $Q_{\leq 0}[x] p(x)$ is not monotone in $\{1\}$. Indeed, both $\emptyset$ and $\{p(a)\}$ satisfy $\text{SM}[\Pi; p]$, but only $\emptyset$ satisfies $\text{FLP}[\Pi; p]$.

Conditions under which the FLP semantics coincides with the first-order stable model semantics has been studied in [4; 5] in the context of logic programs with aggregates.

6 Conclusion

We introduced two definitions of a stable model. One is a reformulation of the first-order stable model semantics and its extension to allow generalized quantifiers by referring to grounding and reduct, and the other is a reformulation of the FLP semantics and its extension to allow generalized quantifiers by referring to a translation into second-order logic. These new definitions help us understand the relationship between the FLP semantics and the first-order stable model semantics, and their extensions. For the class of programs where the two semantics coincide, system DLV-HEX can be viewed as an implementation of the stable model semantics of GQ-formulas; A recent extension of system F2LP [19] to allow “complex” atoms may be considered as a front-end to DLV-HEX to implement the generalized FLP semantics.
Acknowledgements

We are grateful to Vladimir Lifschitz for useful discussions related to this paper. We are also grateful to Joseph Babb and the anonymous referees for their useful comments. This work was partially supported by the National Science Foundation under Grant IIS-0916116 and by the South Korea IT R&D program MKE/KIAT 2010-TD-300404-001.

References

Lloyd-Topor Completion and General Stable Models

Vladimir Lifschitz and Fangkai Yang
Department of Computer Science
The University of Texas at Austin
{vl,fkyang}@cs.utexas.edu

Abstract. We investigate the relationship between the generalization of program completion defined in 1984 by Lloyd and Topor and the generalization of the stable model semantics introduced recently by Ferraris et al. The main theorem can be used to characterize, in some cases, the general stable models of a logic program by a first-order formula. The proof uses Truszczynski’s stable model semantics of infinitary propositional formulas.

1 Introduction

The theorem by François Fages [1] describing a case when the stable model semantics is equivalent to program completion is one of the most important results in the theory of stable models. It was generalized in [2–4], it has led to the invention of loop formulas [5], and it has had a significant impact on the design of answer set solvers.

The general stable model semantics defined in [6] characterizes the stable models of a first-order sentence \( F \) as arbitrary models of a certain second-order sentence, denoted by \( SM[F] \); logic programs are viewed there as first-order sentences written in “logic programming notation.” In this note we define an extension of Fages’ theorem that can be used as a tool for transforming \( SM[F] \), in some cases, into an equivalent first-order formula. That extension refers to the version of program completion introduced by John Lloyd and Rodney Topor in [7]. Their definition allows the body of a rule to contain propositional connectives and quantifiers.

Earlier work in this direction is reported in [6] and [8]. These papers do not mention completion in the sense of Lloyd and Topor explicitly. Instead, they discuss ways to convert a logic program to “Clark normal form” by strongly equivalent transformations [9, 10] and completing programs in this normal form by replacing implications with equivalences. But this is essentially what Lloyd-Topor completion does.

The following examples illustrate some of the issues involved. Let \( F \) be the program

\[
p(a), \quad q(b), \quad p(x) \leftarrow q(x),
\]

1 To be precise, the definition of SM in that paper requires that a set of “intensional predicates” be specified. In the examples below, we assume that all predicate symbols occurring in \( F \) are intensional.
or, in other words, the sentence

\[ p(a) \land q(b) \land \forall x(q(x) \rightarrow p(x)). \]

The Clark normal form of (1) is tight in the sense of [6], and Theorem 11 from that paper shows that SM[F] in this case is equivalent to the conjunction of the completed definitions of \( p \) and \( q \):

\[
\begin{align*}
\forall x(p(x) \leftrightarrow x = a \lor q(x)), \\
\forall x(q(x) \leftrightarrow x = b).
\end{align*}
\]

Let now \( F \) be the program

\[
\begin{align*}
p(x) &\leftarrow q(x), \\
q(a) &\leftarrow p(b).
\end{align*}
\]

This program is not tight in the sense of [6], so that the above-mentioned theorem is not applicable. In fact, SM[F] is stronger in this case than the conjunction of the completed definitions

\[
\begin{align*}
\forall x(p(x) \leftrightarrow q(x)), \\
\forall x(q(x) \leftrightarrow x = a \land p(b)).
\end{align*}
\]

A counterexample is provided by any interpretation that treats each of the symbols \( p, q \) as a singleton such that its element is equal to both \( a \) and \( b \). Such a (non-Herbrand) interpretation satisfies (4), but it is not a stable model of (3). (In stable models of (3) both \( p \) and \( q \) are empty.)

Program (3) is, however, atomic-tight in the sense of [8, Section 5.1.1]. Corollary 5 from that paper allows us to conclude that the equivalence between SM[F] and (4) is entailed by the unique name assumption \( a \neq b \). It follows that the result of applying SM to the program obtained from (4) by adding the constraint

\[ \leftarrow a = b \]

is equivalent to the conjunction of the completion sentences (4) with \( a \neq b \). This example illustrates the role of a property more general than the logical equivalence between SM[F] and the completion of \( F \): it may be useful to know when the equivalence between these two formulas is entailed by a certain set of assumptions. This information may be relevant if we are interested in a logic program obtained from \( F \) by adding constraints.

The result of applying SM to the program

\[
\begin{align*}
p(a) &\leftarrow p(b), \\
q(c) &\leftarrow q(d), \\
\leftarrow a = b, \\
\leftarrow c = d
\end{align*}
\]

is equivalent to the conjunction of the formulas

\[
\begin{align*}
\forall x(p(x) \leftrightarrow x = a \land p(b)), \\
\forall x(q(x) \leftrightarrow x = c \land q(d)), \\
\quad a \neq b, \\
\quad c \neq d.
\end{align*}
\]
This claim cannot be justified, however, by a reference to Corollary 5 from [8]. The program in this example is atomic-tight, but it does not contain constraints corresponding to some of the unique name axioms, for instance $a \neq c$. We will show how our claim follows from the main theorem stated below.

We will discuss also an example illustrating limitations of earlier work that is related to describing dynamic domains in answer set programming (ASP). The program in that example is not atomic-tight because of rules expressing the commonsense law of inertia. We will show nevertheless that the process of completion can be used to characterize its stable models by a first-order formula.

The class of tight programs is defined in [6] in terms of predicate dependency graphs; that definition is reproduced in Section 3 below. The definition of an atomic-tight program in [8] refers to more informative “first-order dependency graphs.” Our approach is based on an alternative solution to the problem of making predicate dependency graphs more informative, “rule dependency graphs.”

After reviewing some background material in Sections 2 and 3, we define rule dependency graphs in Section 4, state the main theorem and give examples of its use in Sections 5 and 6, and outline its proof in Sections 7 and 8.

2 Review: Operator SM, Lloyd-Topor Programs, and Completion

In this paper, a formula is a first-order formula that may contain the propositional connectives $\bot$ (logical falsity), $\land$, $\lor$, and $\rightarrow$, and the quantifiers $\forall$, $\exists$. We treat $\neg F$ as an abbreviation for $F \rightarrow \bot$; $\top$ stands for $\bot \rightarrow \bot$; $F \leftrightarrow G$ stands for $(F \rightarrow G) \land (G \rightarrow F)$.

For any first-order sentence $F$ and any tuple $p$ of distinct predicate constants (other than equality) $SM_p[F]$ is the conjunction of $F$ with a second-order “stability condition”; see [6, Section 2] for details. The members of $p$ are called intensional, and the other predicate constants are extensional. We will drop the subscript in the symbol $SM_p$ when $p$ is the list of all predicate symbols occurring in $F$. For any sentence $F$, a $p$-stable (or simply stable) model of $F$ is an interpretation of the underlying signature that satisfies $SM_p[F]$.

A Lloyd-Topor program is a finite set of rules of the form

$$ p(t) \leftarrow G, \quad (7) $$

where $t$ is a tuple of terms, and $G$ is a formula. We will identify a program with the sentence obtained by conjoining the formulas

$$ \forall (G \rightarrow p(t)) $$

for all its rules (7). ($\forall F$ stands for the universal closure of $F$.)

Let $\Pi$ be a Lloyd-Topor program, and $p$ a predicate constant (other than equality). Let

$$ p(t^i) \leftarrow G^i \quad (i = 1, 2, \ldots) \quad (8) $$

be all rules of $\Pi$ that contain $p$ in the head. The definition of $p$ in $\Pi$ is the rule

$$ p(x) \leftarrow \bigvee_i \exists y^i (x = t^i \land G^i), \quad (9) $$
where \( x \) is a list of distinct variables not appearing in any of the rules (8), and \( y^i \) is the list of free variables of (8). The **completed definition of \( p \) in \( \Pi \)** is the formula

\[
\forall x \left( p(x) \leftrightarrow \bigvee_i \exists y^i (x = t^i \land G^i) \right). \tag{10}
\]

For instance, the completed definitions of \( p \) and \( q \) in program (1) are the formulas

\[
\forall x_1 (p(x_1) \iff x_1 = a \lor \exists x (x_1 = x \land q(x)));
\forall x_1 (q(x_1) \iff x_1 = b),
\]

which can be equivalently rewritten as (2).

By \( \text{Comp}[\Pi] \) we denote the conjunction of the completed definitions of all predicate constants \( p \) in \( \Pi \). This sentence is similar to the completion of \( \Pi \) in the sense of [7, Section 2], except that it does not include Clark equality axioms.

### 3 Review: Tight Programs

We will review now the definition of tightness from [6, Section 7.3]. In application to a Lloyd-Topor program \( \Pi \), when all predicate constants occurring in \( \Pi \) are treated as intensional, that definition can be stated as follows.

An occurrence of an expression in a first-order formula is negated if it belongs to a subformula of the form \( \neg F \) (that is, \( F \rightarrow \perp \)), and nonnegated otherwise. The **predicate dependency graph of \( \Pi \)** is the directed graph that has

- all predicate constants occurring in \( \Pi \) as its vertices, and
- an edge from \( p \) to \( q \) whenever \( \Pi \) contains a rule (7) with \( p \) in the head such that its body \( G \) has a positive\(^3\) nonnegated occurrence of \( q \).

We say that \( \Pi \) is tight if the predicate dependency graph of \( \Pi \) is acyclic.

For example, the predicate dependency graph of program (1) has a single edge, from \( p \) to \( q \). The predicate dependency graph of program (3) has two edges, from \( p \) to \( q \) and from \( q \) to \( p \). The predicate dependency graph of the program

\[
\begin{align*}
p(a, b) \\
q(x, y) & \leftarrow p(y, x) \land \neg p(x, y)
\end{align*} \tag{11}
\]

has a single edge, from \( q \) to \( p \) (because one of the occurrences of \( p \) in the body of the second rule is nonnegated). The predicate dependency graph of the program

\[
\begin{align*}
p(x) & \leftarrow q(x), \\
q(x) & \leftarrow r(x), \\
r(x) & \leftarrow s(x)
\end{align*} \tag{12}
\]

\(^2\) By \( x = t^i \) we denote the conjunction of the equalities between members of the tuple \( x \) and the corresponding members of the tuple \( t^i \).

\(^3\) Recall that an occurrence of an expression in a first-order formula is called positive if the number of implications containing that occurrence in the antecedent is even.
Proposition 1. If a Lloyd-Topor program $\Pi$ is tight then $\text{SM}[\Pi]$ is equivalent to $\text{Comp}[\Pi]$.

This is an easy corollary to the theorem from [6] mentioned in the introduction. Indeed, consider the set $\Pi'$ of the definitions (9) of all predicate constants $p$ in $\Pi$. It can be viewed as a formula in Clark normal form in the sense of [6, Section 6.1]. It is tight, because it has the same predicate dependency graph as $\Pi$. By Theorem 11 from [6], $\text{SM}[\Pi']$ is equivalent to the completion of $\Pi'$ in the sense of [6, Section 6.1], which is identical to $\text{Comp}[\Pi]$. It remains to observe that $\Pi$ is intuitionistically equivalent to $\Pi'$, so that $\text{SM}[\Pi]$ is equivalent to $\text{SM}[\Pi']$ [6, Section 5.1].

4  Rule Dependency Graph

We are interested in conditions on a Lloyd-Topor program $\Pi$ ensuring that the equivalence

$$\text{SM}[\Pi] \leftrightarrow \text{Comp}[\Pi]$$

is entailed by a given set of assumptions $\Gamma$. Proposition 1 gives a solution for the special case when $\Gamma$ is empty. The following definition will help us answer the more general question.

The rule dependency graph of a Lloyd-Topor program $\Pi$ is the directed graph that has

- rules of $\Pi$, with variables (both free and bound) renamed arbitrarily, as its vertices, and
- an edge from a rule $p(t) \leftarrow G$ to a rule $p'(t') \leftarrow G'$, labeled by an atomic formula $p'(s)$, if $p'(s)$ has a positive nonnegated occurrence in $G$.

Unlike the predicate dependency graph, the rule dependency graph of a program is usually infinite. For example, the rule dependency graph of program (11) has the vertices $p(a, b)$ and

$$q(x_1, y_1) \leftarrow p(y_1, x_1) \land \neg p(x_1, y_1)$$

for arbitrary pairs of distinct variables $x_1, y_1$. It has an edge from each vertex (13) to $p(a, b)$, labeled $p(y_1, x_1)$. The rule dependency graph of program (12) has edges of two kinds:

- from $p(x_1) \leftarrow q(x_1)$ to $q(x_2) \leftarrow r(x_2)$, labeled $q(x_1)$, and
- from $q(x_1) \leftarrow r(x_1)$ to $r(x_2) \leftarrow s(x_2)$, labeled $r(x_1)$

for arbitrary variables $x_1, x_2$.

The rule dependency graph of a program is “dual” to its predicate dependency graph, in the following sense. The vertices of the predicate dependency graph are predicate symbols, and the presence of an edge from $p$ to $q$ is determined by the existence
of a rule that contains certain occurrences of $p$ and $q$. The vertices of the rule dependency graph are rules, and the presence of an edge from $R_1$ to $R_2$ is determined by the existence of a predicate symbol with certain occurrences in $R_1$ and $R_2$.

There is a simple characterization of tightness in terms of rule dependency graphs:

**Proposition 2.** A Lloyd-Topor program $\Pi$ is tight iff there exists $n$ such that the rule dependency graph of $\Pi$ has no paths of length $n$.

**Proof.** Assume that $\Pi$ is tight, and let $n$ be the number of predicate symbols occurring in $\Pi$. Then the rule dependency graph of $\Pi$ has no paths of length $n + 1$. Indeed, assume that such a path exists:

$$R_0 \xrightarrow{p_1} R_1 \xrightarrow{p_2} R_2 \xrightarrow{p_3} \ldots \xrightarrow{p_{n+1}} R_{n+1}.$$ 

Each of the rules $R_i$ ($1 \leq i \leq n$) contains $p_i$ in the head and a positive nonnegated occurrence of $p_{i+1}$ in the body. Consequently the predicate dependency graph of $\Pi$ has an edge from $p_i$ to $p_{i+1}$, so that $p_1, \ldots, p_{n+1}$ is a path in that graph; contradiction.

Now assume that $\Pi$ is not tight. Then there is an infinite path $p_1, p_2, \ldots$ in the predicate dependency graph of $\Pi$. Let $R_i$ be a rule of $\Pi$ that has $p_i$ in the head and a positive nonnegated occurrence of $p_{i+1}$ in the body. Then the rule dependency graph of $\Pi$ has an infinite path of the form

$$R_1 \xrightarrow{p_2} R_2 \xrightarrow{p_3} \ldots.$$

The main theorem, stated in the next section, refers to finite paths in the rule dependency graph of a program $\Pi$ that satisfy an additional condition: the rules at their vertices have no common variables (neither free nor bound). Such paths will be called **chains**.

**Corollary 1.** A Lloyd-Topor program $\Pi$ is tight iff there exists $n$ such that $\Pi$ has no chains of length $n$.

Indeed, any finite path in the rule dependency graph of $\Pi$ can be converted into a chain of the same length by renaming variables.

## 5 Main Theorem

Let $C$ be a chain

$$
p_0(t^0) \leftarrow \text{Body}_0
\downarrow p_1(s^1)
p_1(t^1) \leftarrow \text{Body}_1
\downarrow p_2(s^2)
\ldots
\downarrow p_n(s^n)
p_n(t^n) \leftarrow \text{Body}_n
$$

(14)
in a Lloyd-Topor program $\Pi$. The corresponding chain formula $F_C$ is the conjunction

$$\bigwedge_{i=1}^n s_i = t_i \land \bigwedge_{i=0}^n Body_i.$$ 

For instance, if $C$ is the chain

$$q(x_1, y_1) \leftarrow p(y_1, x_1) \land \neg p(x_1, y_1)$$

$$\downarrow p(y_1, x_1)$$

$$p(a, b)$$

in program (11) then $F_C$ is

$$y_1 = a \land x_1 = b \land p(y_1, x_1) \land \neg p(x_1, y_1).$$

Let $\Gamma$ be a set of sentences. About a Lloyd-Topor program $\Pi$ we will say that it is tight relative to $\Gamma$, or $\Gamma$-tight, if there exists a positive integer $n$ such that, for every chain $C$ in $\Pi$ of length $n$,

$$\Gamma, \text{Comp}[\Pi] \models \neg \forall \neg F_C.$$ 

**Main Theorem.** If a Lloyd-Topor program $\Pi$ is $\Gamma$-tight then

$$\Gamma \models \text{SM}[\Pi] \iff \text{Comp}[\Pi].$$

Corollary 1 shows that every tight program is trivially $\Gamma$-tight even when $\Gamma$ is empty. Consequently the main theorem can be viewed as a generalization of Proposition 1.

Tightness in the sense of Section 3 is a syntactic condition that is easy to verify; $\Gamma$-tightness is not. Nevertheless, the main theorem is useful because it may allow us to reduce the problem of characterizing the stable models of a program by a first-order formula to verifying an entailment in first-order logic.

Here are some examples. In each case, to verify $\Gamma$-tightness we take $n = 1$. We will check the entailment in the definition of $\Gamma$-tightness by deriving a contradiction from (some subset of) the assumptions $\Gamma$, $\text{Comp}[\Pi]$, and $F_C$.

**Example 1.** The one-rule program

$$p(a) \leftarrow p(x) \land x \neq a$$

is tight relative to $\emptyset$. Indeed, any chain of length 1 has the form

$$p(a) \leftarrow p(x_1) \land x_1 \neq a$$

$$\downarrow p(x_1)$$

$$p(a) \leftarrow p(x_2) \land x_2 \neq a.$$ 

The corresponding chain formula

$$x_1 = a \land p(x_1) \land x_1 \neq a \land p(x_2) \land x_2 \neq a.$$
is contradictory. Thus the stable models of this program are described by its completion, even though the program is not tight (and not even atomic-tight).

**Example 2.** Let \( \Pi \) be the program consisting of the first 2 rules of (5):

\[
\begin{align*}
p(a) &\leftarrow p(b), \\
q(c) &\leftarrow q(d).
\end{align*}
\]

To justify the claim about (5) made in the introduction, we will check that \( \Pi \) is tight relative to \( \{ a \neq b, c \neq d \} \). There are two chains of length 1:

\[
\begin{align*}
p(a) &\leftarrow p(b) \\
&\downarrow p(b) \\
p(a) &\leftarrow p(b)
\end{align*}
\]

and

\[
\begin{align*}
q(c) &\leftarrow q(d) \\
&\downarrow q(d) \\
q(c) &\leftarrow q(d)
\end{align*}
\]

The corresponding chain formulas are

\[
b = a \land p(b) \land p(b)
\]

and

\[
d = c \land q(d) \land q(d).
\]

Each of them contradicts \( \Gamma' \).

**Example 3.** Let us check that program (3) is tight relative to \( \{ a \neq b \} \). Its chains of length 1 are

\[
\begin{align*}
p(x_1) &\leftarrow q(x_1) \\
&\downarrow q(x_1) \\
q(a) &\leftarrow p(b)
\end{align*}
\]

and

\[
\begin{align*}
q(a) &\leftarrow p(b) \\
&\downarrow q(b) \\
p(x_1) &\leftarrow q(x_1)
\end{align*}
\]

for an arbitrary variable \( x_1 \). The corresponding chain formulas include the conjunctive term \( p(b) \). Using the completion (4) of the program, we derive \( b = a \), which contradicts \( \Gamma' \).

### 6 A Larger Example

Programs found in actual applications of ASP usually involve constructs that are not allowed in Lloyd-Topor programs, such as choice rules and constraints. Choice rules have the form

\[
\{ p(t) \} \leftarrow G.
\]
We view this expression as shorthand for the sentence
\[ \neg \forall (G \rightarrow p(t) \lor \neg p(t)). \]

A constraint \[ \leftarrow G \] is shorthand for the sentence \[ \neg \forall \neg G. \] Such sentences do not correspond to any rules in the sense of Section 2.

Nevertheless, the main theorem stated above can sometimes help us characterize the stable models of a “realistic” program by a first-order formula. In this section we discuss an example of this kind.

The logic program \( M \) described below encodes commonsense knowledge about moving objects from one location to another. Its signature consists of

- the object constants \( \hat{0}, \ldots, \hat{k} \), where \( k \) is a fixed nonnegative integer;
- the unary predicate constants \( \text{object}, \text{place}, \text{and step} \); they correspond to the three types of individuals under consideration;
- the binary predicate constant \( \text{next} \); it describes the temporal order of steps;
- the ternary predicate constants \( \text{at} \) and \( \text{move} \); they represent the fluents and actions that we are interested in.

The predicate constants \( \text{step}, \text{next}, \) and \( \text{at} \) are intensional; the other three are not. (The fact that some predicates are extensional is the first sign that \( M \) is not a Lloyd-Topor program.) The program consists of the following rules:

(i) the facts
\[
\text{step}(\hat{0}), \text{step}(\hat{1}), \ldots, \text{step}(\hat{k}); \\
\text{next}(\hat{0}, \hat{1}), \text{next}(\hat{1}, \hat{2}), \ldots, \text{next}(\hat{k-1}, \hat{k});
\]

(ii) the unique name constraints
\[ \leftarrow \hat{i} = \hat{j} \quad (1 \leq i < j \leq k); \]

(iii) the constraints describing the arguments of \( \text{at} \) and \( \text{move}: \)

\[ \leftarrow \text{at}(x, y, z) \land \neg(\text{object}(x) \land \text{place}(y) \land \text{step}(z)) \]

and
\[ \leftarrow \text{move}(x, y, z) \land \neg(\text{object}(x) \land \text{place}(y) \land \text{step}(z)); \]

(iv) the uniqueness of location constraint
\[ \leftarrow \text{at}(x, y_1, z) \land \text{at}(x, y_2, z) \land y_1 \neq y_2; \]

(v) the existence of location constraint
\[ \leftarrow \text{object}(x) \land \text{step}(z) \land \neg \exists y \text{ at}(x, y, z); \]

(vi) the rule expressing the effect of moving an object:
\[ \text{at}(x, y, u) \leftarrow \text{move}(x, y, z) \land \text{next}(z, u); \]
(vii) the choice rule expressing that initially an object can be anywhere:
\{at(x, y, 0)\} ← object(x) ∧ place(y);
(viii) the choice rule expressing the commonsense law of inertia:\footnote{This representation of inertia follows the example of \cite[Figure 1]{?}.}
\{at(x, y, u)\} ← at(x, y, z) ∧ next(z, u).

Program \(M\) is not atomic-tight, so that methods of \cite{8} are not directly applicable to it. Nevertheless, we can describe the stable models of this program without the use of second-order quantifiers. In the statement of the proposition below, \(p\) stands for the list of intensional predicates \(\text{step}, \text{next}\) and \(\text{at}\), and \(H\) is the conjunction of the universal closures of the formulas
\[
\begin{align*}
\&\hat{i} \neq \hat{j} \quad (1 \leq i < j \leq k), \\
\& at(x, y, z) \rightarrow object(x) \land place(y) \land \text{step}(z), \\
\& move(x, y, z) \rightarrow object(x) \land place(y) \land \text{step}(z), \\
\& at(x, y_1, z) \land at(x, y_2, z) \rightarrow y_1 = y_2, \\
\& object(x) \land \text{step}(z) \rightarrow \exists y \ at(x, y, z).
\end{align*}
\]

**Proposition 3.** \(\text{SM}_p[M]\) is equivalent to the conjunction of \(H\) with the universal closures of the formulas
\[
\begin{align*}
\text{step}(z) & \leftrightarrow \bigvee_{i=0}^{k} z = \hat{i}, \\
\text{next}(z, u) & \leftrightarrow \bigvee_{i=0}^{k-1} (z = \hat{i} \land u = \hat{i+1}), \\
\text{at}(x, y, i+1) & \leftrightarrow (move(x, y, i) \lor (at(x, y, i) \land \neg \exists w \ move(x, w, i))) \\
& \quad (i = 0, \ldots, k - 1).
\end{align*}
\]

Recall that the effect of adding a constraint to a logic program is to eliminate its stable models that violate that constraint \cite[Theorem 3]{6}. An interpretation satisfies \(H\) iff it does not violate any of the constraints (ii)–(v). So the statement of Proposition 3 can be summarized as follows: the contribution of rules (i) and (vi)–(viii), under the stable model semantics, amounts to providing explicit definitions for \(\text{step}\) and \(\text{next}\), and “successor state formulas” for \(\text{at}\).

The proof of Proposition 3 refers to the Lloyd-Topor program \(II\) consisting of rules (i), (vi),
\[
\begin{align*}
\text{(vii')} \ at(x, y, 0) & \leftrightarrow object(x) \land place(y) \land \neg at(x, y, 0), \\
\text{(viii')} \ at(x, y, u) & \leftrightarrow at(x, y, z) \land next(z, u) \land \neg at(x, y, t_2),
\end{align*}
\]
and
\[
\begin{align*}
\text{object}(x) & \leftrightarrow \neg \neg \text{object}(x), \\
\text{place}(y) & \leftrightarrow \neg \neg \text{place}(y), \\
\text{move}(x, y, z) & \leftrightarrow \neg \neg \text{move}(x, y, z).
\end{align*}
\]

\footnote{This representation of inertia follows the example of \cite[Figure 1]{?}.}
It is easy to see that SM\_{p}[M] is equivalent to SM[\Pi] \land H. Indeed, consider the program M′ obtained from M by adding rules (18). These rules are strongly equivalent to the choice rules
\{object(x)\}, \{place(y)\}, \{move(x, y, z)\}.

Consequently SM\_{p}[M] is equivalent to SM[\Pi′] \land H [6, Theorem 2]. It remains to notice that (vii) is strongly equivalent to (vii′), and (viii) is strongly equivalent to (viii′).

Furthermore—and this is the key step in the proof of Proposition 3—the second-order formula SM[\Pi] \land H is equivalent to the first-order formula Comp[\Pi] \land H, in view of our main theorem and the following fact:

**Lemma 1.** Program II is H-tight.

To derive Proposition 3 from the lemma, we only need to observe that (15) and (16) are the completed definitions of step and next in II, and that the completed definition of at can be transformed into (17) under assumptions (15), (16), and H.

**Proof of Lemma 1.** Consider a chain in II of length k + 2:

\[
R_0 \xrightarrow{p_1} \ldots \xrightarrow{p_k} R_{k+1} \ldots \xrightarrow{p_{k+2}} R_{k+2}.
\]

Each \(R_i\) is obtained from one of the rules (i), (vi), (vii′), (viii′), (18) by renaming variables. Each \(p_i\) occurs in the head of \(R_i\) and has a positive nonnegated occurrence in \(R_{i-1}\). Since there are no nonnegated predicate symbols in the bodies of rules (i) and (18), we conclude that \(R_0, \ldots, R_{k+1}\) are obtained from other rules of II, that is, from (vi), (vii′), and (viii′). Since the predicate constant in the head of each of these three rules is \(at\), each of \(p_1, \ldots, p_{k+1}\) is the symbol \(at\). Since there are no nonnegated occurrences of \(at\) in the bodies of (vi) and (vii′), we conclude that \(R_0, \ldots, R_k\) are obtained by renaming variables in (viii′). This means that chain (18) has the form

\[
\begin{align*}
\text{at}(x_0, y_0, u_0) & \leftarrow \text{at}(x_0, y_0, z_0) \land \text{next}(z_0, u_0) \land \neg \text{at}(x_0, y_0, u_0) \\
\text{at}(x_1, y_1, u_1) & \leftarrow \text{at}(x_1, y_1, z_1) \land \text{next}(z_1, u_1) \land \neg \text{at}(x_1, y_1, u_1) \\
\vdots \\
\text{at}(x_k, y_k, u_k) & \leftarrow \text{at}(x_k, y_k, z_k) \land \text{next}(z_k, u_k) \land \neg \text{at}(x_k, y_k, u_k) \\
\text{at}(x_k, y_k, u_k) & \leftarrow R_{k+1} \\
\text{at}(x_k, y_k, u_k) & \leftarrow \ldots \\
\text{at}(x_k, y_k, u_k) & \leftarrow R_{k+2}.
\end{align*}
\]

The corresponding chain formula contains the conjunctive terms

\[z_0 = u_1, z_1 = u_2, \ldots, z_{k-1} = u_k\]

and

\[\text{next}(z_0, u_0), \text{next}(z_1, u_1), \ldots, \text{next}(z_k, u_k)\].
From these formulas we derive
\[ \text{next}(u_1, u_0), \text{next}(u_2, u_1), \ldots, \text{next}(u_{k+1}, u_k), \] (20)

where \( u_{k+1} \) stands for \( z_k \). Using the completed definition of \text{next}, we conclude:
\[ u_i = \hat{0} \lor \cdots \lor u_i = \hat{k} \quad (0 \leq i \leq k+1). \]

Consider the case when
\[ u_i = \hat{j}_i \quad (0 \leq i \leq k+1) \]
for some numbers \( j_0, \ldots, j_{k+1} \in \{0, \ldots, k\} \). There exists at least one subscript \( i \) such that \( j_i \neq j_{i+1} + 1 \), because otherwise we would have
\[ j_0 = j_1 + 1 = j_2 + 2 = \cdots = j_{k+1} + k + 1, \]
which is impossible because \( j_0, j_{k+1} \in \{0, \ldots, k\} \). By the choice of \( i \), from the completed definition of \text{next} and the unique name assumption (included in \( H \)) we can derive \( \neg \text{next}(j_{i+1}, j_i) \). Consequently \( \neg \text{next}(u_{i+1}, u_i) \), which contradicts (20).

7 Review: Stable Models of Infinitary Formulas

Our proof of the main theorem employs the method proposed (for a different purpose) by Miroslaw Truszczyński [11], and in this section we review some of the definitions and results of that paper. The stable model semantics of propositional formulas due to Paolo Ferraris [12] is extended there to formulas with infinitely long conjunctions and disjunctions, and that generalization is related to the operator SM.

Let \( \mathcal{A} \) be a set of propositional atoms. The sets \( \mathcal{F}_0, \mathcal{F}_1, \ldots \) are defined as follows:

- \( \mathcal{F}_0 = \mathcal{A} \cup \{ \bot \} \);
- \( \mathcal{F}_{i+1} \) consists of expressions \( \mathcal{H}^\ell \) and \( \mathcal{H}^\Delta \), for all subsets \( \mathcal{H} \) of \( \mathcal{F}_0 \cup \ldots \cup \mathcal{F}_i \), and of expressions \( F \to G \), where \( F, G \in \mathcal{F}_0 \cup \ldots \cup \mathcal{F}_i \).

An infinitary formula (over \( \mathcal{A} \)) is an element of \( \bigcup_{i=0}^{\infty} \mathcal{F}_i \).

A (propositional) interpretation is a subset of \( \mathcal{A} \). The satisfaction relation between an interpretation and an infinitary formula is defined in a natural way. The definition of the reduct \( F^I \) of a formula \( F \) relative to an interpretation \( I \) proposed in [12] is extended to infinitary formulas as follows:

- \( \bot^I = \bot \);
- For \( A \in \mathcal{A} \), \( A^I = \bot \) if \( I \nsubseteq A \); otherwise \( A^I = A \).
- \( (\mathcal{H}^\ell)^I = \bot \) if \( I \nsubseteq \mathcal{H}^\ell \); otherwise \( (\mathcal{H}^\ell)^I = \{ G^I \mid G \in \mathcal{H} \}^\ell \).
- \( (\mathcal{H}^\Delta)^I = \bot \) if \( I \nsubseteq \mathcal{H}^\Delta \); otherwise \( (\mathcal{H}^\Delta)^I = \{ G^I \mid G \in \mathcal{H} \}^\Delta \).
- \( (G \to H)^I = \bot \) if \( I \nsubseteq G \to H \); otherwise \( (G \to H)^I = G^I \to H^I \).

An interpretation \( I \) is a stable model of an infinitary formula \( F \) if \( I \) is a minimal model of \( F^I \). An interpretation \( I \) satisfies \( F^I \) iff it satisfies \( F \) [11, Proposition 1], so that stable models of \( F \) are models of \( F \).
Infinitary formulas are used to encode first-order sentences as follows. For any interpretation $I$ in the sense of first-order logic, let $A$ be the set of ground atoms formed from the predicate constants of the underlying signature and the “names” $\xi^*$ of elements $\xi$ of the universe $|I|$—new objects constants that are in a 1–1 correspondence with elements of $|I|$. By $I'$ we denote the set of atoms from $A$ that are satisfied by $I$. In the definition below, $t$ stands for the value assigned to the ground term $i$ by the interpretation $I$. The \textit{grounding} of a first-order sentence $F$ relative to $I$ (symbolically, $gr_I(F)$) is the infinitary formula over $A$ constructed as follows:

- $gr_I(\bot) = \bot$.
- $gr_I(p(t_1, \ldots, t_k)) = p((t^*_1, \ldots, t^*_k))$.
- $gr_I(t_1 = t_2) = \top$, if $t^*_1 = t^*_2$, and $\bot$ otherwise.
- If $F = G \lor H$, $gr_I(F) = gr_I(G) \lor gr_I(H)$ (the case of $\land$ is analogous).
- If $F = G \rightarrow H$, $gr_I(F) = gr_I(G) \rightarrow gr_I(H)$.
- If $F = \exists x G(x)$, $gr_I(F) = \{gr_I(G(u^*)) | u \in |I|\}^\land$.
- If $F = \forall x G(x)$, $gr_I(F) = \{gr_I(G(u^*)) | u \in |I|\}^\land$.

It is easy to check that $gr_I$ is a faithful translation in the following sense: $I$ satisfies a first-order sentence $F$ iff $I'$ satisfies $gr_I(F)$.

This transformation is also faithful in the sense of the stable model semantics: $I$ satisfies $SM[F]$ iff $I'$ is a stable model of $gr_I(F)$ [11, Theorem 5]. This is why infinitary formulas can be used for proving properties of the operator $SM$.

8 Proof Outline

In the statement of the main theorem, the implication left-to-right

$$SM[I] \rightarrow Comp[I]$$

is logically valid for any Lloyd-Topor program $I$. This fact follows from [6, Theorem 11] by the argument used in the proof of Proposition 1 above. In this section we outline the proof in the other direction:

\textit{If a Lloyd-Topor program $I$ is $\Gamma$-tight,}

\textit{and an interpretation $I$ satisfies both $\Gamma$ and $Comp[I]$,}

\textit{then $I$ satisfies $SM[I]$}.

This assertion follows from three lemmas. The first of them expresses a Fages-style property of infinitary formulas similar to Theorem 1 from [3]. It deals with \textit{infinitary programs}—conjunctions of (possibly infinitely many) implications $G \rightarrow A$ with $A \in A$. Such an implication will be called an \textit{(infinitary) rule} with the \textit{head} $A$ and \textit{body} $G$, and we will write it as $A \leftarrow G$. For instance, if $I$ is a Lloyd-Topor program then, for any interpretation $I$, $gr_I(I)$ is an infinitary program. We say that an interpretation $I$ is \textit{supported} by an infinitary program $I$ if each atom $A \in I$ is the head of a rule $A \leftarrow G$ of $I$ such that $I \models G$. The lemma shows that under some condition the
stable models of an infinitary program $\Pi$ can be characterized as the interpretations that satisfy $\Pi$ and are supported by $\Pi$.

The condition refers to the set of positive nonnegated atoms of an infinitary formula. This set, denoted by $\text{Pnn}(\mathcal{F})$, and the set of negative nonnegated atoms of $\mathcal{F}$, denoted by $\text{Nnn}(\mathcal{F})$, are defined recursively, as follows:

- $\text{Pnn}(\bot) = \emptyset$.
- For $A \in \mathcal{A}$, $\text{Pnn}(A) = \{A\}$.
- $\text{Pnn}(\mathcal{H}^\wedge) = \text{Pnn}(\mathcal{H}^\lor) = \bigcup_{H \in \mathcal{H}} \text{Pnn}(H)$.
- $\text{Pnn}(G \rightarrow H) = \begin{cases} \emptyset & \text{if } H = \bot, \\ \text{Nnn}(G) \cup \text{Pnn}(H) & \text{otherwise.} \end{cases}$

- $\text{Nnn}(\bot) = \emptyset$.
- For $A \in \mathcal{A}$, $\text{Nnn}(A) = \emptyset$.
- $\text{Nnn}(\mathcal{H}^\wedge) = \text{Nnn}(\mathcal{H}^\lor) = \bigcup_{H \in \mathcal{H}} \text{Nnn}(H)$.
- $\text{Nnn}(G \rightarrow H) = \begin{cases} \emptyset & \text{if } H = \bot, \\ \text{Pnn}(G) \cup \text{Nnn}(H) & \text{otherwise.} \end{cases}$

Let $\Pi$ be an infinitary program, and $I$ a propositional interpretation. About atoms $A, A' \in I$ we say that $A'$ is a parent of $A$ relative to $\Pi$ and $I$ if $\Pi$ has a rule $A \leftarrow G$ with the head $A$ such that $I \models G$ and $A'$ is a positive nonnegated atom of $G$. We say that $\Pi$ is tight on $I$ if there is no infinite sequence $A_0, A_1, \ldots$ of elements of $I$ such that for every $i$, $A_{i+1}$ is a parent of $A_i$ relative to $\mathcal{F}$ and $I$.

**Lemma 2.** For any model $I$ of an infinitary program $\Pi$ such that $\Pi$ is tight on $I$, $I$ is stable iff $I$ is supported by $\Pi$.

The next lemma relates the $\Gamma'$-tightness condition from the statement of the main theorem to tightness on an interpretation defined above.

**Lemma 3.** If a Lloyd-Topor program $\Pi$ is $\Gamma'$-tight, and an interpretation $I$ satisfies both $\Gamma$ and Comp[$\Pi$], then $\text{gr}_I(\Pi)$ is tight on $\Gamma'$.

Finally, models of Comp[$\Pi$] can be characterized in terms of satisfaction and supportedness:

**Lemma 4.** For any Lloyd-Topor program $\Pi$, an interpretation $I$ satisfies Comp[$\Pi$] iff $\Gamma'$ satisfies $\text{gr}_I(\Pi)$ and is supported by $\text{gr}_I(\Pi)$.

Proofs of Lemmas 2–4 can be found in the longer version of the paper, posted at http://www.cs.utexas.edu/users/vl/papers/ltc-long.pdf.

9 Conclusion

We proposed a new method for representing SM($\mathcal{F}$) in the language of first-order logic. It is more general than the approach of [6]. Its relationship with the ideas of [8] requires further study. This method allows us, in particular, to prove the equivalence of some ASP descriptions of dynamic domains to axiomatizations based on successor state axioms.

The use of the stable model semantics of infinitary formulas [11] in the proof of the main theorem illustrates the potential of that semantics as a tool for the study of the operator SM.
Acknowledgements

We are grateful to Joohyung Lee and to the anonymous referees for useful comments.

References

Extending FO(ID) with Knowledge Producing Definitions: Preliminary Results

Joost Vennekens¹ and Marc Denecker²

¹ Campus De Nayer, Lessius Mechelen, KU Leuven
De Nayerlaan 5
2860 Sint-Katelijne-Waver, Belgium
² Dept. Computer Science, KU Leuven
Celestijnenlaan 200A
3001 Heverlee, Belgium

Abstract. Previous research into the relation between ASP and classical logic has identified at least two different ways in which the former extends the latter. First, ASP programs typically contain sets of rules that can be naturally interpreted as inductive definitions, and the language FO(ID) has shown that such inductive definitions can elegantly be added to classical logic in a modular way. Second, there is of course also the well-known epistemic component of ASP, which was mainly emphasized in the early papers on stable model semantics. To investigate whether this kind of knowledge can also be added in a similarly modular way, the language of Ordered Epistemic Logic was presented in recent work. However, this logic views the epistemic component as entirely separate from the inductive definition component, thus ignoring any possible interplay between the two. In this paper, we present a language that extends the inductive definition construct found in FO(ID) with an epistemic component, making such interplay possible. The eventual goal of this work is to discover whether it is really appropriate to view the epistemic component and the inductive definition component of ASP as two separate extensions of classical logic, or whether there is also something of importance in the combination of the two.

1 Introduction

Today, Answer Set Programming (ASP) is a vibrant domain, boasting both mature technologies and successful real-world applications. The roots of ASP lie in the fields of Logic Programming (LP) and Non-monotonic Reasoning (NMR). Both of these were initially motivated by dissatisfaction with classical first-order logic (FO), be it its computational properties (in the case of LP) or its suitability for representing common-sense knowledge (in the case of NMR). The current success of ASP suggests that, to a large extent, this domain was indeed able to overcome these problems of classical logic.

It is, however, not yet quite clear how precisely this was done. That is to say, the relation between ASP and classical logic is, in our opinion, not yet fully
understood. Currently, ASP still stands as an alternative to FO: to effectively write ASP programs, one basically has to leave behind all methodologies, tools and intuitive understandings of classical logic and start anew in a different setting. This paper is part of a research project that attempts to close this gap [10]. The aim is to investigate whether and how the achievements of ASP can be reformulated as modular improvements or extensions of classical logic. Ultimately, we would like to be able to characterize ASP as a set of specific solutions to a number of orthogonal problems/limitations of classical logic, such that someone working in classical logic could add as many or as few “ASP-style features” to his knowledge base as is needed for that particular application. Of course, the motivation for this research is not purely practical. By reformulating the contributions of ASP in the classical framework, we also hope to provide a synthesis that will eventually lead to an increased understanding of classical and computational logic, and their role in problem solving.

Ironically, ASP’s relation to classical logic seems currently best understood when it comes to computational aspects. For instance, [20] showed that the typical ASP way of encoding search problems can be captured quite elegantly in a classical context by the notion of modal expansion: given a theory $T$ in an alphabet $\Sigma$ and an interpretation $I_o$ for some subvocabulary $\Sigma_o \subseteq \Sigma$, find an interpretation $I$ that extends $I_o$ to the entire vocabulary such that $I \models T$. Indeed, the 2011 edition of the ASP-competition [9] has had two modal expansion systems for (extensions of) classical logic among its competitors: Enfragmo [1] and IDP [19]. On a more technical level, also the similarities between current ASP solvers and SAT solvers are of course well understood (e.g., [16, 11, 15]).

When it comes to the knowledge representation properties of ASP (i.e., the intuitive meaning of expressions and the modeling methodologies that have to be followed), the relation to classical logic is less clear. As pointed out by [5], one of the key problems lies in the interpretation of the semantic structures: whereas an answer set in ASP is traditionally interpreted in an epistemic way, as a representation of the knowledge of some rational agent, classical logic is based on the Tarskian view of a model as a representation of a possible objective state of affairs. Nevertheless, a series of papers by Denecker et al. [4, 8, 7] has shown that a substantial portion of ASP can be understood as a combination of classical FO axioms and inductive definitions.

An inductive definitions is a well-understood mathematical construct, that is usually represented by a set of natural language if-then-statements. As shown in [4, 8, 7], we can view a set of normal logic programming rules as a formal representation of such an inductive definition. For instance, the following pair of rules:

$$\begin{align*}
T(x, y) & \leftarrow T(x, z) \land T(z, y). \\
T(x, y) & \leftarrow E(x, y).
\end{align*}$$

define $T$ as the transitive closure of $E$. This is of course not overly surprising and it indeed goes back to the views of Van Gelder [21] and Clark [2]. Nevertheless, taking this observation seriously immediately suggests a clean and well-defined “plugin” that can modularly add an ASP-style component to a classical FO the-
ory. Indeed, because an inductive definition is nothing more than a generalization of the way in which relations are usually (non-inductively) defined by means of an FO equivalence, there should be nothing problematic (either conceptually or mathematically) about allowing such “sets of rules that form an inductive definition” anywhere one is allowed to use an equivalence. FO(ID) (previously known as ID-logic) is the language that does precisely this: it extends FO, in a completely modular way, with a rule-based representation for inductive definition [3]. Representing a search problem as a model expansion problem in FO(ID) often yields results that are almost identical to a Generate-Define-Test program in ASP [17], apart from minor syntactic details [18, 5]. In this way, FO(ID) therefore fits nicely into our stated goal, by identifying one concrete way in which ASP improves upon FO and packing this up in a language construct that can be added to an existing FO knowledge base at will.

While FO(ID) seems able to naturally represent already a surprisingly large part of existing ASP practice, it by no means covers everything. One class of examples that remains out of scope is that of the epistemic examples that originally motivated the stable model semantics [12, 13]. This is epitomized by the well-known interview example [14], which is expressed in ASP as:

\[
\begin{align*}
&Eligible(x) \leftarrow HighGPA(x). \\
&Eligible(x) \leftarrow Minority(x), FairGPA(x). \\
&\neg Eligible(x) \leftarrow \neg FairGPA(x). \\
&Interview(x) \leftarrow \neg Eligible(x), \neg \neg Eligible(x).
\end{align*}
\]

Recent efforts have attempted to reformulate also this kind of example in a way that explains its relation to FO. In [23], the language of Ordered Epistemic Logic (OEL) is developed for this. Here, an ordered set of FO theories is considered, and each theory \( T \) is allowed to make use of modal operators \( K_T \) that refer to the knowledge entailed by a theory \( T' < T \). The interview example would consist of two theories \( T_1, T_2 \) with \( T_1 < T_2 \), such that \( T_1 \) is a normal FO knowledge base containing facts about \( Minority, HighGPA \) and \( FairGPA \), and a definition of \( Eligible \). The theory \( T_2 \) consists of the single equivalence:

\[
\forall x \, Interview(x) \iff \neg K_{T_1} Eligible(x) \land \neg K_{T_1} \neg Eligible(x).
\]

This logic extends FO in a way which is completely orthogonal to FO(ID). There is nothing to prevent the different knowledge bases of an OEL theory from containing, in addition to regular FO formulas, also inductive definitions, but if they do, there is no interplay with the epistemic operators. In other words, FO(ID) and OEL both isolate one particular non-classical aspect of ASP, and show how it can be modularly added to FO, but they do so independently. This presupposes that there is nothing of importance in ASP’s combination of epistemic and “inductive definition” reasoning. However, it does not seem a priori obvious that this is the case.

In this paper, we will therefore investigate how an epistemic component can be added to the inductive definition construct of FO(ID) itself. The key idea
here is to allow both a relation and an agent’s knowledge about this relation to be defined together in a single knowledge producing definition, as we will call it. The semantics of such a knowledge producing definition is defined by a constructive process that creates, in parallel, the relations that are being defined and the agent’s knowledge about them. In this way, we obtain a language in which, unlike the FO(ID)+OEL approach, interaction between the epistemic and definitional component is possible. The hope is that such a language might shed more light on the epistemic component of ASP and its relation to both classical FO and the inductive definitions of FO(ID).

The work presented in this paper is still at a preliminary stage, but we will attempt to sketch interesting avenues for future research. Because one of our main design goals is to make our approximate knowledge structures integrate seamlessly with the inductive definition construct as it already exists in FO(ID), we will need to spend some time recalling the details of this, before we can develop our extension.

2 Preliminaries: the semantics of inductive definitions

Inductive definitions are constructive. In mathematical texts, they are typically represented by a set of if-then statements, which may be applied to add new elements to the relation(s) that is (or are, in the case of a definition by simultaneous induction) being defined. The formal representation of such a definition in FO(ID) is by a set of rules of the form

\[ \forall x \ P(t) \leftarrow \phi, \]

where \( P(t) \) is an atom and \( \phi \) an FO formula. For monotone definitions, the relation being defined is simply the least relation that is closed under application of the rules, and it can be constructed by exhaustively applying them. For non-monotone definitions, the relation being defined is no longer the least relation closed under the rules, and there may, in fact, be many minimal relations closed under the rules instead of a single least one. In mathematical texts, such a non-monotone definition is always accompanied by a well-founded order over which the induction is performed. While the characterization as a least set breaks down, the constructive characterization still works: the defined relation can still be constructed by repeated application of the rules, provided that these rules are applied in an order that respects the given well-founded order of induction.

For instance, the standard definition of satisfaction in classical logic is a definition over the subformula order, which means that we may only apply a rule that derives that \( I \models \phi \) for some \( \phi \) after all rules that could derive \( I \models \psi \) for a subformula \( \psi \) of \( \phi \) have been applied.

Of course, for a correct inductive definition, it is important that the structure of the rules also respects the well-founded order. For instance, in an definition over the subformula order, it makes no sense for a rule to define whether a formula is satisfied in terms of the satisfaction of a larger formula. Therefore, the structure of the rules and the well-founded order are not independent. In
fact, the well-founded order is already entirely implicit in the structure of the rules! As shown in [8] and a series of prior papers, the well-founded semantics (WFS) [22] can actually be seen as a mathematical construct to recover the well-founded order from the structure of the rules.

For simplicity, in the technical material of this paper, we will restrict attention to ground formulas only.

Since its original definition, a number of alternative ways of defining the WFS have been developed. One of these is to start from the following method of evaluating a formula $\phi$ in a pair of interpretations $(I,J)$:

- For an atom $P(t)$, $(I,J) \models P(t)$ iff $I \models P(t)$,
- For a formula $\neg \psi$, $(I,J) \models \neg \psi$ iff $(J,I) \not\models \psi$,
- For a formula $\psi_1 \lor \psi_2$, $(I,J) \models \psi_1 \lor \psi_2$ iff $(I,J) \models \psi_1$ or $(I,J) \models \psi_2$,
- For a formula $\psi_1 \land \psi_2$, $(I,J) \models \psi_1 \land \psi_2$ iff $(I,J) \models \psi_1$ and $(I,J) \models \psi_2$.

The crux of this definition lies in the case for negation, which switches the roles of $I$ and $J$, thus ensuring that positive occurrences of atoms are evaluated in $I$ and negative occurrences in $J$. We will call a pair $(I,J)$ for which $I \leq J$ an approximating pair, because it can be seen as an approximation of the set of interpretations $K$ such that $I \leq K$ and $K \leq J$. Indeed, if $(I,J) \models \phi$, according to the above definition, then $K \models \phi$ for all such $K$. Moreover, if $K \models \phi$ for at least one such $K$, then $(J,I) \models \phi$. For pairs $(I,J)$ such that $I = J$, the evaluation $(I,J) \models \phi$ reduces to classical satisfaction $I \models \phi$. Pairs for which this is the case are called total.

The WFS can then be defined as the unique limit of a sequence of pairs of interpretations $(I_i,J_i)_{i \geq 0}$. This sequence starts from the least precise pair of interpretations $(\bot_\Sigma, \top_\Sigma)$, where $\bot_\Sigma$ is the interpretation in which all atoms are false and $\top_\Sigma$ the interpretation in which all atoms are true. There are then two acceptable ways of going from $(I_i,J_i)$ to $(I_{i+1},J_{i+1})$:

- Either $J_{i+1} = J_i$ and $I_{i+1}$ is the union $I_i \cup \{P(t)\}$, with $P(t)$ an atom for which there exists a rule of the form $P(t) \leftarrow \phi$ with $(I_i,J_i) \models \phi$;
- Or $I_{i+1} = I_i$ and $J_{i+1}$ is such that $I_i \leq J_{i+1} \leq J_i$ and for all atoms $P(t)$ in the set difference $J_i - J_{i+1}$ and all rules $P(t) \leftarrow \phi$, $(J_{i+1},I_i) \not\models \phi$.

Intuitively, the first of these two cases allows us to derive the head of a rule once it is certain that its body is satisfied (in the sense that $K \models \phi$ for all $K$ approximated by $(I_i,J_i)$). The second case allows us to assume that a set of atoms must all be false if this assumption would make us certain that all bodies $\phi$ of rules with one of these atoms in the head are false (i.e., $K \not\models \phi$ for all $K$ approximated by $(I_i,J_{i+1})$). The set $J_{i+1} - J_i$ of atoms that are falsified in this operation is known as an unfounded set.

A sequence constructed in this way is called an induction sequence. The well-founded model (WFM) is now precisely the unique limit $(V,W)$ to which all such induction sequences converge. If this WFM is total (i.e., $V = W$), then the definition completely determines the extension of the predicates it defines. Clearly, this is a desirable property for an inductive definition. Therefore, FO(ID) allows only total models.
So far, we have tacitly assumed that all predicates in the vocabulary $\Sigma$ are defined by the definition. In mathematics, however, this is rarely the case, since most definitions serve to define some relation(s) in terms of some other relation(s). This is also possible in FO(ID). For a definition $\Delta$ (i.e., a set of rules of form (1)), the predicate symbols appearing in the head of at least one of these rules are called the defined predicates of $\Delta$. The set of all defined predicates is denoted as $Def(\Delta)$. The remaining predicates (i.e., those that belong to $\Sigma - Def(\Delta)$) are called open, and the set of all such predicates is denoted by $Op(\Delta)$. The purpose of a definition is then to characterize the defined predicates $Def(\Delta)$ in terms of the open predicates $Op(\Delta)$. Formally, this is done by parametrizing the construction process by an interpretation for the open predicates: for an interpretation $O$ of $Op(\Delta)$, an induction sequence given $O$ is defined as a sequence of interpretation $(I_i, J_i)_{i \geq 0}$, in which all interpretations $I_i$ and $J_i$ extend the given interpretation $O$. The starting point of this sequence is the pair $(I_0, J_0)$ such that $I_0 = O + \bot_{Def(\Delta)}$ and $J_0 = O + \top_{Def(\Delta)}$.

An interpretation $I$ is then called a model of a definition $\Delta$, denoted $I \models \Delta$, if the unique limit of the induction sequences for $\Delta$ given $I$ to the open predicates of $\Delta$ is precisely the total pair $(I, I)$. FO(ID) now consists of classical first-order logic FO extended with these definitions. While some versions of this logic allow boolean combinations of classical formulas and inductive definitions, we will, for simplicity, restrict attention in this paper to FO(ID) theories that consist of precisely one FO formula $\phi$ and one inductive definition $\Delta$. For such a theory $T = \{\phi, \Delta\}$, we define that $T$ is a model of $T$, denoted $I \models \Delta$, if both $I \models \phi$ (in the classical sense) and $I \models \Delta$ (as defined above).

### 3 Knowledge producing definitions

The goal of this paper is take the concept of an inductive definition as it exists in FO(ID) and extend it by allowing definitions that not only define the objective extension of their defined predicates, but, at the same time, also define a rational agent’s knowledge about these predicates. A modal literal is a formula of the form $K\psi$ where $\psi$ is an FO formula. By FO(K), we denote the language that extends FO by allowing modal literals to appear anywhere an atom $P(t)$ may appear. Note that FO(K) therefore does not allow nesting of the operator $K$. A knowledge producing definition $\kappa$ is a set of rules of either the form

$$\forall x \ K\psi \leftarrow \phi, \quad (2)$$

or

$$\forall x \ P(t) \leftarrow \phi. \quad (3)$$

Here, $K\psi$ is a modal literal and $P(t)$ an atom. In both cases, $\phi$ is an FO(K) formula. Again, in our formal treatment, we will always assume that these rules have already been appropriately grounded. The defined predicates of a knowledge producing definition $\kappa$ are all the predicates $P$ that appear in the head of a rule.
of form (3). All other predicates, including those that appear only in the formula
ψ of a rule of form (2), are open.

For this formalism, the basic semantic structure will consist of a pair of an
interpretation I, representing the real world, and a set of interpretations W,
representing the agent’s knowledge about the world. We call such a pair (I, W)
a knowledge structure and call it consistent if I ∈ W. It is obvious how to evaluate
a knowledge formula in such a knowledge structure.

**Definition 1.** For a knowledge formula φ and a knowledge structure S = (I, W),
we define S |= φ as follows:

- For an atom P(t), S |= P(t) iff I |= P(t),
- For a modal literal Kψ, S |= Kψ iff for each J in W, (J, W) |= ψ,
- The other cases are defined as usual.

Like the WFS, which construct a single interpretation K through a series
of increasingly precise approximations by pairs of interpretations (Ii, Ji), our
semantics will construct a single knowledge structure (I, W) through a series
of increasingly precise approximations of it. Part of these approximations will
again be a pair of interpretations (Ii, Ji), which form an increasingly precise
sequence of approximations of the real extension K of the defined predicates.
At each stage of this approximating sequence, we will also keep track of the
agent’s knowledge about it. Therefore, at each step i, we will also have a set
Wi of pairs of interpretations; if (Ii, Ji) ∈ Wi, then this means that the agent
considers it possible that (Ii, Ji) occurs somewhere in the real approximating
sequence. The way in which we will ensure this property, is to, on the one hand,
apply the same derivation rules we apply to the real sequence to the pairs that
the agent considers possible. On the other hand, when the agent’s knowledge
increases due to a modal literal in the head of a rule, this will eliminate some of
the possibilities (i.e., some of these pairs (I′, J′) are removed from Wi), but it
will not change any of the possibilities themselves (i.e., the remaining pairs do
not change).

The following will serve as our basic semantic structure.

**Definition 2.** An approximate knowledge structure A is a pair ((I, J), W) of
an approximating pair (I, J) and a set W of approximating pairs (I′, J′).

Evaluating an FO(K) formula in such an approximate knowledge structure
is again a matter of switching the approximating pairs when negation is en-
countered. Formally, we define that, for an approximate knowledge structure
A = ((I, J), W),

- For an atom P(t), A |= P(t) iff I |= P(t);
- For a modal literal Kψ, A |= Kψ iff for each (I′, J′) ∈ W, ((I′, J′), W) |= ψ;
- For a formula ¬φ, A |= ¬φ iff ((J, I), W) |= φ, where W = {(J′, I′) | (I′, J′) ∈ W};
- The other cases are defined as usual.
We now construct an increasingly precise sequence \((A_i)_{i \geq 0}\) of approximate knowledge structures. If we project this sequence unto the approximating pair \((I, J)\) of each approximate knowledge structures \(((I, J), W)\), the result will essentially be just a regular induction sequence. The sequence again takes as input a knowledge structure \((O, M)\) for the open predicates, and its starting point is then the approximate knowledge structure

\[
A_0 = ((O + \bot_{Def(\kappa)}, O + \top_{Def(\kappa)}), \{(O' + \bot_{Def(\kappa)}, O' + \top_{Def(\kappa)}) | O' \in M\}).
\]

We construct subsequent elements of the sequence by applying one of the following operations.

**Operation 1.** \(A_{i+1} = ((I_i + \{P(t)\}, J_i), W_i)\) where \(A_i = ((I_i, J_i), W_i)\) such that there is a rule \(r\) of the form \(P(t) \leftarrow \phi\) with \(((I_i, J_i), W_i) \models \phi\).

This operation is just the obvious analogue to the first operation used in building normal induction sequences. The only difference is that the agent’s knowledge \(W_i\) is dragged along as an additional argument, which is used to evaluate occurrences of modal literals in the rule bodies.

**Operation 2.** \(A_{i+1} = ((I_i, J_i), W_{i+1})\) where \(A_i = ((I_i, J_i), W_i)\) and \(W_{i+1} = W_i - \{(I_i', J_i')\} + \{(I_i' + \{P(t)\}, J_i)\}\) such that there is a rule \(r\) of the form \(P(t) \leftarrow \phi\) with \(((I_i', J_i), W_i) \models \phi\).

This operation is essentially the same as the previous one, with the only difference being that it is now not applied to the approximating pair \((I_i, J_i)\), but to one of the approximating pairs \((I_i', J_i')\) in \(W_i\). Where the previous two operations are analogous to the production operation of the normal induction sequence, the next two mimic the unfounded set operation.

**Operation 3.** \(A_{i+1} = ((I_i, J_{i+1}), W_i)\) where \(A_i = ((I_i, J_i), W_i)\) and \(J_{i+1}\) is such that \(I_i \leq J_{i+1} \leq J_i\) and for all atoms \(P(t) \in J_i - J_{i+1}\) and all rules \(r\) of the form \(P(t) \leftarrow \phi\), it holds that \((J_{i+1}, I_i) \not\models \phi\).

Again, this operation can either be performed on the approximating pair \((I_i, J_i)\), as above, or on one of the pairs in the set \(W_i\), as below:

**Operation 4.** Or \(A_{i+1} = ((I_i, J_i), W_{i+1})\) where \(A_i = ((I_i, J_i), W_i)\) and \(W_{i+1} = W_i - \{(I_i', J_i')\} + \{(I_i' + \{P(t)\}, J_i)\}\) and \(J_{i+1}'\) is such that \(I_i' \leq J_{i+1}' \leq J_i'\) and for all atoms \(P(t) \in J_i' - J_{i+1}'\) and all rules \(r\) of the form \(P(t) \leftarrow \phi\), it holds that \(((J_{i+1}', I_i'), W_i) \not\models \phi\).

The final operation takes care of the effect of knowledge producing rules.

**Operation 5.** \(A_{i+1} = ((I_i, J_i), W_{i+1})\) where \(A_i = ((I_i, J_i), W_i)\) and \(W_{i+1} = \{(I, J) \in W_i | (J, I) \models \psi\}\) and there exists a rule \(r\) of the form \(K\psi \leftarrow \phi\) such that \(((I_i, J_i), W_i) \models \phi\).

Note that the condition for removing a pair \((I, J)\) from \(W_i\) is that \((J, I) \not\models \psi\), i.e., that no interpretation approximated by \((I, J)\) still satisfies \(\psi\).

The semantics of a knowledge producing definition is now defined in terms of sequences that are constructed by these operations.
Definition 3. Let $\kappa$ by a knowledge producing definition. Let $(O, M)$ be a knowledge structure describing the agent’s initial knowledge about the open predicates. A knowledge derivation sequence is a sequence $(A_i)_{i \geq 0}$ of approximate knowledge structures, starting from

$$A_0 = ((O + \perp_{\text{Def}(\kappa)}, O + \top_{\text{Def}(\kappa)}), \{(O' + \perp_{\text{Def}(\kappa)}, O' + \top_{\text{Def}(\kappa)}) \mid O' \in M\}),$$

such that each $A_{i+1}$ is obtained from $A_i$ by applying one of the five operations defined above and, to prevent the same operation from being applied again and again, $A_{i+1} \neq A_i$. Such a sequence is called complete if there is no way to extend it further without violating this condition. It is called sound if every operation that is used to construct $A_{i+1}$ from $A_i$ remains applicable in all $A_j$ with $j > i$. It is called total if it terminates in an approximate knowledge structure $((I, J), W)$ such that $I = J$ and for each $(I', J') \in W$ also $I' = J'$.

The condition of totality is borrowed from FO(ID), where, as mentioned, it is used to ensure that each inductive definition correctly and completely defines the relations it sets out to define. It is therefore also a natural requirement in our context.

The condition of soundness is meant to avoid situations in which the sequence ends up contradicting itself, as it might for the following example:

$$\{Kp \leftarrow \neg Kp.\}$$

Here, the fact that the agent does not know $p$ will produce the knowledge that $p$. This is not only conceptually problematic, but also creates the practical problem that the order in which operations are applied might have an effect on the final outcome. For instance, the knowledge definition

$$\begin{cases} Kq \leftarrow \neg Kp. \\ Kp \leftarrow \neg Kq. \end{cases}$$

has both a derivation sequence that starts with the first rule and therefore ends up knowing $q$ but not knowing $p$, and one that starts with the second rule and ends up knowing $p$ but not $q$.

This can only happen with sequences that are not sound, as the following proposition shows.

Proposition 1. All complete and sound knowledge derivation sequences that start from the same knowledge structure $(O, M)$ terminate in the same approximate knowledge structure $A$.

Proof (sketch). At any particular point in the derivation sequence, many operations may be applicable. We have to show that it does not matter which of these we choose. First, because the sequence is sound, we know that even if we do not choose to apply an operation now, we will always get a chance to apply it later. Second, the effect of an operation usually does not depend on when it is
executed. The only exception to this is Operation 5, because the set of approximating pairs may of course change throughout the sequence. However, the only changes to this set are that (1) pairs are removed and that (2) pairs become more precise (i.e., that either the first element $I$ of a pair $(I, J)$ becomes larger or that $J$ becomes smaller, so that fewer interpretations $K$ lie between $I$ and $J$). Clearly, (1) is not a problem, because if a pair gets removed later on any way, it does not matter if we already remove it now or not. Also (2) is not a problem, because the condition for removing a pair $(I, J)$, namely that $(J, I) \neq \phi$, also implies that, for any more precise pair $(I', J')$ (i.e., such that $I \leq I'$ and $J' \leq J$), it will be the case that $(J', I') \neq \phi$ as well. The only effect of postponing the application of an Operation 5 to a later stage is therefore that we might end up removing more pairs than if we had applied it now. However, if we apply the operation now, then at the later stage we will have another Operation 5 available, namely the one that removes precisely those pairs that form the difference. Since this operation will remain applicable and therefore must eventually be applied, the end result will be the same.

Moreover, the only way in which it is possible to obtain unsound derivations is by negated modal literals.

**Proposition 2.** For a knowledge producing definition in which each body of a rule contain only positive occurrences of modal literals $K\psi$, each knowledge derivation sequence is sound.

**Proof (sketch).** The differences between one approximate knowledge structure $A_i = ((I_i, J_i), W_i)$ and an approximate knowledge structure $A_j = ((I_j, J_j), W_j)$ that occurs later in the derivation sequence are:

- $I_i \leq I_j$ and $J_j \leq J_i$: this implies that whenever $((I_i, J_i), W) \models \phi$ for some $W$ and $\phi$, also $((I_j, J_j), W) \models \phi$.
- $W_j$ consists of pairs $(I'_j, J'_j)$ for which there exists a corresponding pair $(I'_i, J'_i) \in W_i$ such that, again, $I'_i \leq I'_j$ and $J'_j \leq J'_i$, and therefore whenever $((I'_i, J'_i), W) \models \phi$ also $((I'_j, J'_j), W) \models \phi$.
- for some $(I'_j, J'_j) \in W_j$, there may not exist a corresponding pair in $W_i$.

Putting the second and third point together, it is obvious that $W_j$ always knows everything that $W_i$ knows, and possibly more. Therefore, only rule bodies containing a negative occurrence of a modal literal may becomes false in $W_j$ after they were true in $W_i$.

This result seems to suggest that it might be possible to impose syntactic constraints on a knowledge producing definition to ensure that it only has sound derivations, by limiting the way in which negated modal literals are allowed to appear. However, it is not enough to, e.g., just require that these definitions are stratified, because gaining new knowledge about one predicate may have “side
effects” where also knowledge about another predicate is produced. For instance, consider the following definition which, at first sight, contains no cycles at all:

\[
\begin{align*}
q &\leftarrow p. \\
Kq &\leftarrow r. \\
r &\leftarrow \neg Kp.
\end{align*}
\]

Even though there are no syntactic cycles, once the agent learns \( q \), this will also produce the knowledge that \( p \) (since \( q \) is only true in worlds where \( p \) also holds). Because of the \( \neg Kp \) in the body of the rule for \( r \), this will lead to an unsound derivation.

Our approach in this paper will be to ignore the existence of unsound derivations and view the unique limit of the sound derivations as the semantics of a knowledge producing definition—at least, if this unique limit is total. If the limit is not total (given a particular knowledge structure \( S \) for the open predicates), then, just like in FO(ID), the knowledge producing definition simply has no models (for that particular \( S \)). If no sound derivations exists (for that \( S \)), then again the knowledge producing definition has no models (for that \( S \)).

## 4 Adding knowledge producing definitions to FO

The previous section defined the concept of a knowledge producing definition in isolation. Of course, our goal is to add this construct to FO(K), in the same way as FO(ID) has added inductive definitions to FO. We will again consider only theories of the form \( T = \{ \phi, \kappa \} \), were \( \phi \) is now an FO(K) formula and \( \kappa \) a knowledge producing definition. We define that a consistent knowledge structure \( S = (I, W) \) is a weak model of \( T \) if \( S \models \phi \) and the approximate knowledge structure \( ((I, I), \{(J, J) \mid J \in W\}) \) is the unique total limit of each sound derivation sequence that starts from \((I_{Op(\kappa)}, \{L_{Op(\kappa)} \mid L \in W\})\).

A problem with these weak models is they might contain knowledge that is not warranted. Consider, for instance, the following example:

\[\{q \leftarrow Kp\}\]

Here, there is no reason at all for knowing \( p \), yet this knowledge producing definition has a weak model \( (\{p, q\}, \{\{p, q\}\}) \). To avoid such models, we introduce the following concept.

**Definition 4.** Let \( T = \{ \phi, \kappa \} \), were \( \phi \) is an FO(K) formula and \( \kappa \) a knowledge producing definition. A knowledge structure \( S = (I, W) \) is a strong model of \( T \), denoted \( S \models T \), if \( S \models \phi \) and the approximate knowledge structure \( ((I, I), \{(J, J) \mid J \in W\}) \) is the unique total limit of each sound derivation sequence that starts from \((I_{Op(\kappa)}, O)\) where \( O \) is the set of all \( L_{Op(\kappa)} \) such that there exists a \( W' \) for which \((L, W')\) is a weak model of \( T \).

By always using the set \( O \) as the set of possible worlds for the open predicates, this definition prevents the knowledge producing definition from arbitrarily knowing more about its open predicates than it should.
5 Examples

In the logic we have now defined, it is straightforward to represent the Interview example.

\[
\begin{align*}
\text{Eligible}(x) & \leftarrow (\text{HighGPA}(x) \lor (\text{Minority}(x) \land \text{FairGPA}(x))). \\
\text{Interview}(x) & \leftarrow \neg K \text{Eligible}(x) \land \neg K \neg \text{Eligible}(x).
\end{align*}
\]

\[\forall x \text{ HighGPA}(x) \iff x = \text{Mary}.\]
\[\forall x \text{ FairGPA}(x) \iff x = \text{John}.\]
\[\text{Minority}(\text{Mary}).\]

Here, the soundness condition on the derivation sequence means that we first have to apply the first rule of the knowledge producing definition to exhaustion, before starting with the second rule. There are two possible interpretations for the open predicates of the definition that satisfy the FO part of the theory:

\[O_1 = \{\text{HighGPA}(\text{Mary}), \text{FairGPA}(\text{John}), \text{Minority}(\text{Mary})\}, \]
\[O_2 = \{\text{HighGPA}(\text{Mary}), \text{FairGPA}(\text{John}), \text{Minority}(\text{Mary}), \text{Minority}(\text{John})\}.\]

Each strong model will therefore have to be produced by a derivation sequence in which \(W_0 = \{O_1, O_2\}\) is used as the agent’s knowledge about the open predicates. Let us first consider a sound derivation sequence that interprets the open predicates by the knowledge structure \(S_1 = (O_1, W_0)\). The first step is \(A_0 = (A, \{A, B\})\) with:

\[A = (O_1 + \bot_{\text{Def}(\kappa)}, O_1 + \top_{\text{Def}(\kappa)})\]
\[B = (O_2 + \bot_{\text{Def}(\kappa)}, O_2 + \top_{\text{Def}(\kappa)})\]

In the approximating pair \(A\), we can apply the first rule of the knowledge producing definition to derive that \(\text{Eligible}(\text{Mary})\), and we can also apply an unfounded set operation to derive that \(\text{Eligible}(\text{John})\) is false. This leads to the new approximating pair (we abbreviate the names of the predicates):

\[A' = (O_1 + \{\text{Elig}(\text{Mary})\}, O_1 + \{\text{Elig}(\text{Mary}), \text{Int}(\text{Mary}), \text{Int}(\text{John})\})\]

In the approximating pair \(B\), on the other hand, we can derive that both \(\text{Eligible}(\text{John})\) and \(\text{Eligible}(\text{Mary})\). This leads to the new pair:

\[B' = (O_2 + \{\text{Elig}(\text{Mary}), \text{Elig}(\text{John})\},\]
\[O_2 + \{\text{Elig}(\text{Mary}), \text{Elig}(\text{John}), \text{Int}(\text{Mary}), \text{Int}(\text{John})\})\]

After applying these six operations (two times two for \(A\), and two for \(B\)) to the approximate knowledge structure \(W_0\), we will therefore eventually end up in

\[W_6 = (A', \{A', B'\})\]
Now, \( \{A', B'\} \models K \operatorname{Elig}(Mary) \), because \( \operatorname{Elig}(Mary) \) holds in both underestimates \( O_1 + \{\operatorname{Elig}(Mary)\} \) and \( O_2 + \{\operatorname{Elig}(Mary), \operatorname{Elig}(John)\} \). Also, \( \{A', B'\} \models \neg K \operatorname{Elig}(John) \), since \( \operatorname{Elig}(John) \) does not hold in the overestimate (the negation switches the pairs) of \( A' \), namely \( O_1 + \{\operatorname{Elig}(Mary), \operatorname{Int}(Mary), \operatorname{Int}(John)\} \). Finally, \( \{A', B'\} \models \neg K \neg \operatorname{Elig}(John) \) holds as well, since \( \operatorname{Elig}(John) \) does not belong to the underestimate (the two negations switch the pairs twice) of \( A' \), namely \( O_1 + \{\operatorname{Elig}(Mary)\} \). Therefore, we can apply to both \( A' \) and \( B' \) an un-founded set operation (Operations 3 and 4) to derive that \( Mary \) should not be interviewed, and we can apply the last rule of the definition to derive that \( John \) should. After six more steps (two times two for \( John \) interviewed, and we can apply the last rule of the definition to derive that \( John \) should. Namely \( O_3 \) and \( \{\operatorname{Elig}(Mary)\} \) does not belong to the underestimate (the two negations switch the pairs twice) of \( A' \), namely \( O_1 + \{\operatorname{Elig}(Mary)\} \). Therefore, we can apply to both \( A' \) and \( B' \) an un-founded set operation (Operations 3 and 4) to derive that \( Mary \) should not be interviewed, and we can apply the last rule of the definition to derive that \( John \) should. After six more steps (two times two for \( John \), and two for \( B' \), we therefore end up in the limit \( W_9 = (A'', \{A'', B''\}) \), where

\[
A'' = (I, I) \text{ with } I = O_1 + \{\operatorname{Elig}(Mary), \operatorname{Int}(John)\}, \\
B'' = (J, J) \text{ with } J = O_1 + \{\operatorname{Elig}(Mary), \operatorname{Elig}(John), \operatorname{Int}(John)\}
\]

This structure is total and therefore the single knowledge structure \( (I, \{I, J\}) \) that it approximates is a model of this theory. By a similar reasoning, there is also a second model, namely \( (J, \{I, J\}) \).

### 5.1 Another version of the Interview example

In the above version of this example, we are basically already encoding the solution to the problem by ordering the agent to interview everyone whose eligibility is not known. Using a knowledge producing definition, however, it is also possible to let the semantics do more of the work.

\[
\left\{
\begin{aligned}
\operatorname{Eligible}(x) &\leftarrow (\operatorname{HighGPA}(x) \lor (\operatorname{Minority}(x) \land \operatorname{FairGPA}(x))). \\
K \operatorname{Minority}(x) &\leftarrow \operatorname{Interview}(x) \land \operatorname{Minority}(x). \\
K \neg \operatorname{Minority}(x) &\leftarrow \operatorname{Interview}(x) \land \neg \operatorname{Minority}(x).
\end{aligned}
\right\}
\]

\[
(\forall x \, \operatorname{HighGPA}(x) \iff x = \operatorname{Mary}) \land (\forall x \, \operatorname{FairGPA}(x) \iff x = \operatorname{John}) \land \operatorname{Minority}(\operatorname{Mary}).
\]

\[
\forall x \, K \operatorname{Eligible}(x) \lor K \neg \operatorname{Eligible}(x).
\]

Here, we are just telling the agent that the action of interviewing a candidate will reveal his minority status, without explicitly saying who should be interviewed. The \( \text{FO}(K) \) constraint then orders the agent to make sure that for each candidate, it is known whether he is eligible of not. Generating models for this theory will then correctly produce plans in which people whose minority status is unknown will be interviewed.

### 5.2 Sensing actions

A successful application area of ASP is planning. This also falls naturally in the scope of \( \text{FO}(ID) \), since theories in the situation or event calculus are essentially just an inductive definition of the values of the fluents at different points in time.
Here is an example of a theory in FO(ID) that represents a small action domain in which there is a dirty glass that can be cleaned by wiping it.

\[
\begin{align*}
\{ & \text{Clean}(t+1) \leftarrow \text{Wipe}(t) \lor \text{Clean}(t). \\
& \text{Clean}(0) \leftarrow \text{InitClean}. \}
\end{align*}
\]

\[\neg \text{InitClean} \]

If the agent now does not know whether the glass is initially clean, we may be interested in finding a plan that will allow it to know with certainty that it will be clean at a certain point in time. This can be accomplished by just adding, e.g., \( K \text{Clean}(2) \) as an FO(K) constraint to the theory. More interestingly, knowledge producing definitions can also be used to add sensing actions, such as an action \( \text{Inspect} \) that allows the agent to discover whether the glass is clean.

\[
\begin{align*}
\{ & \text{Clean}(t+1) \leftarrow \text{Wipe}(t) \lor \text{Clean}(t). \\
& \text{Clean}(0) \leftarrow \text{InitClean}. \\
& K \text{Clean}(t+1) \leftarrow \text{Inspect} \land \text{Clean}(t). \\
& K \neg \text{Clean}(t+1) \leftarrow \text{Inspect} \land \neg \text{Clean}(t). \}
\end{align*}
\]

\[ K \text{Clean}(2). \]

6 Discussion

ASP is able to express epistemic examples by interpreting an answer set as a set of literals that are \textit{believed} by a rational agent. This is one of the most radical ways in which ASP departs from classical logic, in which models or interpretations always represent the \textit{objective} state of the world. In the classical setting, one typically resorts to sets of interpretations (or the related concept of a Kripke structure) to represent beliefs.

In order to relate this aspect of ASP to classical logic, or to even integrate the two, it is necessary to construct a formalisation which sticks to these classical semantics objects. In this paper, we have introduced knowledge producing definitions for this purpose. As our examples have shown, these are able to mimic the ASP representation of, e.g., the \textit{Interview} example, while at the same time also introducing some interesting new possibilities, such as the ability to distinguish between some atom \( P(t) \) becoming objectively true (by having \( P(t) \) in the head of a rule) and the agent learning this atom (by having \( K P(t) \) in the head).

References

Author Index

Alviano, M., 3
Balai, E., 18
Balduccini, M., 34
Charwat, G., 49
Chintabathina, S., 66
Denecker, M., 158
Eiter, T., 81
Faber, W., 3
Fichte, J., 97
Fink, M., 81
Gebser, M., 112
Gelfond, M., 18, 34
Grote, T., 112
Kaminski, R., 112
Krennwallner, T., 81
Lee, J., 128
Lifschitz, V., 142
Meng, Y., 128
Obermeier, P., 112
Redl, C., 81
Sabuncu, O., 112
Schüller, P., 81
Schaub, T., 112
Szeider, S., 97
Vennekens, J., 158
Wallner, J., 49
Woltran, S., 49
Yang, F., 142
Zhang, Y., 18