# Answer Set Programming and Other Computing Paradigms

Sixth International Workshop
August 25th 2013, Istanbul, Turkey

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 SAT, which has led to a 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), constraint satisfaction, and general means of external computation.

This volume contains the papers presented at the sixth workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2013) held on August 25$^{th}$, 2013 in Istanbul, co-located with the 29$^{th}$ International Conference on Logic Programming (ICLP 2013). 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.

Twelve 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 related or more expressive settings; methods and techniques to compute semantics, respectively to build systems potentially exploiting exiting ASP solvers; 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.

We would like to take this opportunity to thank all authors, PC members, reviewers, speakers, and participants for making this workshop happen. We very much appreciated the support of the ICLP General Co-Chairs Esra Erdem and Joohyung Lee, and the ICLP Workshop Chair Hans Tompits. Moreover, we acknowledge the EasyChair system which we used for organizing the submission and review processes.

August 2013

Michael Fink
Yuliya Lierler
(Organizers ASPOCP 2013)

# Organisation

## Executive Committee

| | |
|---|---|
| Workshop Chairs: | Michael Fink<br>(Vienna University of Technology, Austria) |
| | Yuliya Lierler<br>(University of Nebraska at Omaha, USA) |

## Programme Committee

| | |
|---|---|
| Marcello Balduccini | Kodak Research Labs, USA |
| Pedro Cabalar | Coruña University, Spain |
| Sandeep Chintabathina | University of Arkansas at Pine Bluff, USA |
| Selim T. Erdogan | Independent Researcher, Turkey |
| Wolfgang Faber | University of Calabria, Italy |
| Cristina Feier | University of Oxford, UK |
| Gregory Gelfond | Arizona State University, USA |
| Martin Gebser | University of Potsdam, Germany |
| Giovambattista Ianni | University of Calabria, Italy |
| Daniela Inclezan | Miami University in Ohio, USA |
| Joohyung Lee | Arizona State University, USA |
| Joao Leite | New University of Lisbon, Portugal |
| Vladimir Lifschitz | University of Texas at Austin, USA |
| Marco Maratea | University of Genoa, Italy |
| Alessandro Mosca | Free University of Bolzano, Italy |
| Emilia Oikarinen | Aalto University, Finland |
| David Pearce | Universidad Politécnica de Madrid, Spain |
| Axel Polleres | Vienna University of Economics and Business, Austria |
| Guillermo R. Simari | Universidad Nacional del Sur, Argentina |
| Evgenia Ternovska | Simon Fraser University, Canada |
| Hans Tompits | Vienna University of Technology, Austria |
| Tran Cao Son | New Mexico State University, USA |
| Miroslaw Truszczynski | University of Kentucky, USA |
| Joost Vennekens | Catholic University of Leuven, Belgium |
| Marina De Vos | University of Bath, UK |
| Stefan Woltran | Vienna University of Technology, Austria |
| Fangkai Yang | University of Texas at Austin, USA |
| Jia-Huai You | University of Alberta, Canada |

## Additional Reviewers

| | |
|---|---|
| Bernhard Bliem | Vienna University of Technology, Austria |
| Günther Charwat | Vienna University of Technology, Austria |
| Michael Gelfond | Texas Tech University, USA |
| Holger Jost | University of Potsdam, Germany |
| Shahab Tasharrofi | Simon Fraser University, Canada |
| Concepción Vidal | Coruña University, Spain |

# Table of Contents

## I Papers

# Part I

# Papers

# Properties of Answer Set Programming with Convex Generalized Atoms

Mario Alviano and Wolfgang Faber

Department of Mathematics and Computer Science
University of Calabria
87036 Rende (CS), Italy
{alviano,faber}@mat.unical.it

**Abstract.** In recent years, Answer Set Programming (ASP), logic programming under the stable model or answer set semantics, has seen several extensions by generalizing the notion of an atom in these programs: be it aggregate atoms, HEX atoms, generalized quantifiers, or abstract constraints, the idea is to have more complicated satisfaction patterns in the lattice of Herbrand interpretations than traditional, simple atoms. In this paper we refer to any of these constructs as generalized atoms. Several semantics with differing characteristics have been proposed for these extensions, rendering the big picture somewhat blurry. In this paper, we analyze the class of programs that have convex generalized atoms (originally proposed by Liu and Truszczyński in [10]) in rule bodies and show that for this class many of the proposed semantics coincide. This is an interesting result, since recently it has been shown that this class is the precise complexity boundary for the FLP semantics. We investigate whether similar results also hold for other semantics, and discuss the implications of our findings.

## 1 Introduction

Various extensions of the basic Answer Set Programming language have been proposed by allowing more general atoms in rule bodies, for example aggregate atoms, HEX atoms, dl-atoms, generalized quantifiers, or abstract constraints. A number of semantics have been proposed for such programs, most notably the FLP semantics [7] and a number of coinciding semantics that we will collectively refer to as PSP semantics (from Pelov, Son, and Pontelli) [13, 17]. All of these semantics coincide with traditional ASP semantics when no generalized atoms are present. Moreover, they coincide on programs that have atomic rule heads and contain only monotonic generalized atoms. In [9] it is furthermore hinted that the semantics also coincide on programs that have atomic rule heads and contain only convex generalized atoms. However, no formal proof is available for this claim, and the informal explanation given in [9] is not as general as it could be, as we will show.

In this paper, we undertake a deeper investigation on the similarities and differences between the FLP and PSP semantics. In order to do this, we consider a simplified, yet expressive propositional language: sets of rules with atomic heads and bodies that are formed of a single "structure," which are functions mapping interpretations to Boolean values[1]. Clearly, structures encompass atoms, literals, and conjunctions thereof, but can

[1] Note that (apart from the name) there is no connection to structures in first-order logic.

represent any propositional formula, generalized atom, or conjunctions of generalized atoms. Each structure has an associated domain, which is the set of propositional atoms on which the structure's truth valuation depends. We can then classify the structures by their semantic properties, in particular, we will focus on the class of convex structures, which have single contiguous areas of truth in the lattice of interpretations. Convex structures include atoms and literals, and they are closed under conjunction (but not under negation or disjunction).

We first formally prove the claim that the FLP and PSP semantics coincide on programs with convex structures, as originally reported in [9]. We will then move on to the main focus of this paper, trying to understand whether there is any larger class for which the semantics coincide. It is known that for programs with general structures all PSP answer sets are FLP answer sets, but not all FLP answer sets are PSP answer sets. The precise boundary for exhibiting the semantic difference is instead unknown.

We will approach this question using complexity arguments. Recently, we could show that convex structures form the precise boundary for a complexity jump in the polynomial hierarchy on cautious reasoning (but most other decision problems as well) for the FLP semantics. Cautious reasoning is $\Pi_2^P$-complete for the FLP semantics when allowing any non-convex structure and its variants (renaming atoms) in the input program, but it is $coNP$-complete for convex structures. When considering the PSP semantics, cautious reasoning is also $\Pi_2^P$-complete when allowing any kind of structures in the input. This follows from a result in [13], and we provide an alternative proof in this paper. Analyzing this proof, it becomes clear that there is a different source of complexity for PSP than for FLP.

We then show that this different source of complexity also yields a different shape of the boundary for the complexity jump in PSP. Indeed, we first show that for a simple non-convex structure, cautious reasoning is still in $coNP$ for the PSP semantics, while the problem is $\Pi_2^P$-hard in the presence of this structure for the FLP semantics. It turns out that the same argument works for many non-convex structures, in particular, for all structures with a domain size bounded by a constant. The domain size therefore serves as a parameter that simplifies the complexity of the problem for the PSP semantics (unless the polynomial hierarchy collapses to its first level). This also means that the complexity boundary for PSP has a non-uniform shape, in the sense that an infinite number of different non-convex structures must be available for obtaining $\Pi_2^P$-hardness for cautious reasoning. This is in contrast to the FLP semantics, where the presence of a single non-convex structure is sufficient.

## 2 Syntax and Semantics

In this section we first introduce the syntax used in the paper. This is mainly based on the notion of structures, i.e., functions mapping interpretations into Boolean truth values. Then, we introduce few semantic notions and in particular we characterize structures in terms of monotonicity. Finally, we define the two semantics analyzed in this paper, namely FLP and PSP.

### 2.1 Syntax

Let $\mathcal{U}$ be a fixed, countable set of propositional atoms. An interpretation $I$ is a subset of $\mathcal{U}$. A structure $S$ on $\mathcal{U}$ is a mapping of interpretations into Boolean truth values. Each structure $S$ has an associated, finite domain $D_S \subset \mathcal{U}$, indicating those atoms that are relevant to the structure.

*Example 1.* A structure $S_1$ modeling a conjunction $a_1, \ldots, a_n$ ($n \geq 0$) of propositional atoms is such that $D_{S_1} = \{a_1, \ldots, a_n\}$ and, for every interpretation $I$, $S_1$ maps $I$ to true if and only if $D_{S_1} \subseteq I$.

A structure $S_2$ modeling a conjunction $a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$ ($n \geq m \geq 0$) of literals, where $a_1, \ldots, a_n$ are propositional atoms and $not$ denotes *negation as failure*, is such that $D_{S_2} = \{a_1, \ldots, a_n\}$ and, for every interpretation $I$, $S_2$ maps $I$ to true if and only if $\{a_1, \ldots, a_m\} \subseteq I$ and $\{a_{m+1}, \ldots, a_n\} \cap I = \emptyset$.

A structure $S_3$ modeling an aggregate $COUNT(\{a_1, \ldots, a_n\}) \neq k$ ($n \geq k \geq 0$), where $a_1, \ldots, a_n$ are propositional atoms, is such that $D_{S_3} = \{a_1, \ldots, a_n\}$ and, for every interpretation $I$, $S_3$ maps $I$ to true if and only if $|D_{S_3} \cap I| \neq k$.

A general rule $r$ is of the following form:

$$H(r) \leftarrow B(r) \tag{1}$$

where $H(r)$ is a propositional atom in $\mathcal{U}$ referred as the head of $r$, and $B(r)$ is a structure on $\mathcal{U}$ called the body of $r$. A general program $P$ is a set of general rules.

*Example 2.* Let $S_4$ map to true any interpretation $I$ such that $I \cap \{a, b\} \neq \{b\}$, and let $S_5$ map to true any interpretation $I$ such that $I \cap \{a, b\} \neq \{a\}$. Hence, program $P_1 = \{a \leftarrow S_4; b \leftarrow S_5\}$ is equivalent to the following program with aggregates:

$$a \leftarrow SUM(\{a = 1, b = -1\}) \geq 0$$
$$b \leftarrow SUM(\{a = -1, b = 1\}) \geq 0$$

Note that no particular assumption is made on the syntax of rule bodies; in the case of normal propositional logic programs these structures are conjunctions of literals. We assume that structures are closed under propositional variants, that is, if $S$ is a structure, for any bijection $\sigma : \mathcal{U} \to \mathcal{U}$, also $S\sigma$ is a structure, and the associated domain is $D_{S\sigma} = \{\sigma(a) \mid a \in D_S\}$.

*Example 3.* Consider $S_4$ and $S_5$ from Example 2, and a bijection $\sigma_1$ such that $\sigma_1(a) = b$. Hence, $S_5 = S_4\sigma_1$, that is, $S_5$ is a variant of $S_4$.

Given a set of structures $\mathbf{S}$, by datalog$^{\mathbf{S}}$ we refer to the class of programs that may contain only the following rule bodies: structures corresponding to conjunctions of atoms, any structure $S \in \mathbf{S}$, or any of its variants $S\sigma$.

*Example 4.* For every $n \geq m \geq 0$, let $S^{m,n}$ denote the structure $S_2$ from Example 1. The class of normal datalog programs is datalog$^{\{S^{m,n} \mid n \geq m \geq 0\}}$.

Note that this syntax does not explicitly allow for negated structures. One can, however, choose the complementary structure for simulating negation. This would be akin to the "negation as complement" interpretation of negated aggregates that is prevalent in the literature.

### 2.2 Semantics

Let $I \subseteq \mathcal{U}$ be an interpretation. $I$ is a model for a structure $S$, denoted $I \models S$, if $S$ maps $I$ to true. Otherwise, if $S$ maps $I$ to false, $I$ is not a model of $S$, denoted $I \not\models S$. We require that atoms outside the domain of $S$ are irrelevant for modelhood, that is, for any interpretation $I$ and $X \subseteq \mathcal{U} \setminus D_S$ it holds that $I \models S$ if and only if $I \cup X \models S$. Moreover, for any bijection $\sigma : \mathcal{U} \to \mathcal{U}$, let $I\sigma = \{\sigma(a) \mid a \in I\}$, and we require that $I\sigma \models S\sigma$ if and only if $I \models S$. $I$ is a model of a rule $r$ of the form (1), denoted $I \models r$, if $H(r) \in I$ whenever $I \models B(r)$. $I$ is a model of a program $P$, denoted $I \models P$, if $I \models r$ for every rule $r \in P$.

*Example 5.* Consider program $P_1$ from Example 2. It can be observed that $\emptyset \not\models P_1$ and $\{a, b\} \not\models P_1$ (both rules have true bodies but false heads), while $\{a\} \models P_1$ and $\{b\} \models P_1$.

Structures can be characterized in terms of *monotonicity* as follows.

**Definition 1 (Monotone Structures).** *A structure $S$ is monotonic if for all pairs $X, Y$ of interpretations such that $X \subset Y$, $X \models S$ implies $Y \models S$.*

**Definition 2 (Antimonotone Structures).** *A structure $S$ is antimonotonic if for all pairs $Y, Z$ of interpretations such that $Y \subset Z$, $Z \models S$ implies $Y \models S$.*

**Definition 3 (Convex Structures).** *A structure $S$ is convex if for all triples $X, Y, Z$ of interpretations such that $X \subset Y \subset Z$, $X \models S$ and $Z \models S$ implies $Y \models S$.*

Note that monotonic and antimonotonic structures are convex. Moreover, note that convex structures are closed under conjunction (but not under disjunction or negation).

*Example 6.* Structure $S^{m,n}$ from Example 4 is convex in general; it is monotonic if $m = n$, and antimonotonic if $m = 0$. Structure $S_3$ from Example 1, instead, is non-convex if $n > k > 0$; it is monotonic if $k = 0$, and antimonotonic if $n = k$.

We first describe a reduct-based semantics, usually referred to as FLP, which has been described and analyzed in [6, 7].

**Definition 4 (FLP Reduct).** *The FLP reduct $P^I$ of a program $P$ with respect to $I$ is defined as the set $\{r \in P \mid I \models B(r)\}$.*

**Definition 5 (FLP Answer Sets).** *$I$ is an FLP answer set of $P$ if $I \models P^I$ and for each $J \subset I$ it holds that $J \not\models P^I$.*

*Example 7.* Consider program $P_1$ from Example 2 and the interpretation $\{a\}$. The reduct $P_1^{\{a\}}$ is $\{a \leftarrow S_4\}$. Since $\{a\}$ is a minimal model of the reduct, $\{a\}$ is an FLP answer set of $P_1$. Similarly, it can be observed that $\{b\}$ is another FLP answer set. Actually, these are the only FLP answer sets of the program.

We will next describe a different semantics, using the definition of [17], called "fix-point answer set" in that paper. Theorem 3 in [17] shows that it is actually equivalent to the two-valued fix-point of ultimate approximations of generalized atoms in [13][2], and therefore with stable models for ultimate approximations of aggregates as defined in [14]. We will refer to it as PSP to abbreviate Pelov/Son/Pontelli, the names most frequently associated with this semantics.

**Definition 6 (Conditional Satisfaction).** *A structure $S$ on $\mathcal{U}$ is* conditionally satisfied *by a pair of interpretations $(I, M)$, denoted $(I, M) \models S$, if $J \models S$ for each $J$ such that $I \subseteq J \subseteq M$.*

**Definition 7 (PSP Answer Sets).** *An interpretation $M$ is a PSP answer set if $M$ is the least fixpoint of the following operator:*

$$K_M^P(I) = \{H(r) \mid r \in P \wedge (I, M) \models B(r)\}. \tag{2}$$

*Example 8.* Consider program $P_1$ from Example 2 and the interpretation $\{a\}$. The least fixpoint of $K_{\{a\}}^{P_1}$ is $\{a\}$. In fact, $\emptyset \models S_4$ and $\{a\} \models S_4$, hence $(\emptyset, \{a\}) \models S_4$, while $\{a\} \not\models S_5$ and thus $(\emptyset, \{a\}) \not\models S_5$ and $(\{a\}, \{a\}) \not\models S_5$. Therefore, $\{a\}$ is a PSP answer set. Also $\{b\}$ is a PSP answer set.

On programs considered in this paper, PSP answer sets also coincide with "answer sets" defined in [18] (by virtue of Proposition 10 in [18]) and "well-justified FLP answer sets" of [15] (by virtue of Theorem 5 in [15]). The latter is particularly interesting, as it is defined by first forming the FLP reduct. Indeed, as shown in [15], the operator $K_M^P$ can be equivalently defined as follows:

$$K_M^P(I) = \{H(r) \mid r \in P^M \wedge (I, M) \models B(r)\}. \tag{3}$$

There are several other semantic definitions on programs that have some restrictions on the admissible structures, which also coincide with the PSP semantics on programs as defined in this paper with the respective structure restriction. Examples are [12] for monotonic structures (that are also allowed to occur in rule heads in that paper), or [16] that allows for structures corresponding to cardinality and weight constraints and largely coincide with the PSP semantics (see [13] for a discussion on structures on which the semantics coincides).

In this paper we are mainly interested in cautious reasoning, defined next.

**Definition 8 (Cautious Reasoning).** *A propositional atom $a$ is a cautious consequence of a program $P$ under FLP (resp. PSP) semantics, denoted $P \models_c^{FLP} a$ (resp. $P \models_c^{PSP} a$), if $a$ belongs to all FLP (resp. PSP) answer set of $P$.*

*Example 9.* Consider program $P_1$ from Example 2. We have $P_1 \not\models_c^{FLP} a$ and $P_1 \not\models_c^{FLP} b$, and similar for PSP semantics. If we add $a \leftarrow b$ and $b \leftarrow a$ to the program, then there is only one FLP answer set, namely $\{a, b\}$, and no PSP answer sets. In this case $a$ and $b$ are cautious consequences of the program (under both semantics).

[2] There is an even closer relationship, as the operator $K_M^P(I)$ of [17] coincides with $\phi_P^{aggr,1}(I, M)$ defined in [13], as shown in the appendix of [17]

## 3 Exploring the Relationship between the FLP and PSP Semantics

In this section, we examine in detail how the FLP and PSP semantics relate. We shall proceed in three steps. First, we formally prove that FLP and PSP semantics coincide on programs with convex structures in Section 3.1. Next, we turn towards complexity as a tool to understand whether there can be any larger class of coinciding programs. We start in Section 3.2 with a result that shows that programs without restrictions exhibit the same complexity under both FLP and PSP semantics. However, it is known that the semantics do not coincide for programs without restrictions, and we examine the complexity proofs to highlight the different complexity sources. These findings are then applied in Section 3.3 in order to identify programs with bounded non-convex structures, on which the complexities for FLP and PSP semantics differ. Under usual complexity assumptions, this also implies that programs with convex aggregates is the largest class of programs on which FLP and PSP coincide.

### 3.1 Unison: Convex Structures

In this section we show that for programs with convex aggregates the FLP and PSP semantics coincide. In [9] it is stated that many semantics (and in particular, FLP and PSP) "agree on [...] programs with convex aggregates" because "they can be regarded as special programs with monotone constraints." However, the comment on regarding convex aggregates as monotone constraints relies on a transformation described in [10] that transforms convex structures into conjunctions of positive and negated monotone constraints. Since our language does not explicitly allow negation, and in particular since convex structures are not closed under negation, we next prove in a more direct manner that the FLP and PSP semantics coincide on convex structures.

One direction of the proof relies on the well-known more general fact that each PSP answer set is also an FLP answer set. This has been stated as Theorem 2 in [17] and Proposition 8.1 in [14].

**Theorem 1.** *Let $P$ be program whose body structures are convex, and let $M$ be an interpretation. $M$ is an FLP answer set of $P$ if and only if $M$ is an PSP answer set of $P$.*

*Proof.* The left implication follows from Theorem 2 in [17]. For the right implication, let $M$ be an FLP answer set of $P$. Let $K_0 := \emptyset$, $K_{i+1} := K_M^P(K_i)$ for $i \geq 0$, and let $K$ be the fixpoint of this sequence. Since $M$ is a minimal model of $P^M$ by definition of FLP answer set, we can prove the claim by showing (i) $K \models P^M$ and (ii) $K \subseteq M$.

(i) Consider a rule $r \in P^M$ such that $K \models B(r)$. We have to show $H(r) \in K$. Since $r \in P^M$, $M \models B(r)$ holds. Thus, $(K, M) \models B(r)$ and therefore $H(r) \in K$.

(ii) We prove $K_i \subseteq M$ for each $i \geq 0$. We use induction on $i$. The base case is trivially true as $K_0 = \emptyset \subseteq M$. Suppose $K_i \subseteq M$ for some $i \geq 0$ in order to prove $K_{i+1} \subseteq M$. By definition of $K_M^P$, for each $a \in K_{i+1}$ there is $r \in P^M$ such that $H(r) = a$ and $(K_i, M) \models B(r)$. Thus, $M \models B(r)$, which implies $a \in M$. □

Therefore, programs with convex structures form a class of programs for which the FLP and PSP semantics coincide. In the following, we will show that it is likely also the largest class for which this holds.

### 3.2 Consonance: Complexity of Unrestricted Structures

In this section we will examine the computational impact of allowing non-convex structures. We will limit ourselves to structures for which the truth value with respect to an interpretation can be determined in polynomial time. Moreover, we will focus on cautious reasoning, but similar considerations apply also to related problems such as brave reasoning, answer set existence, or answer set checking.

It is known that cautious reasoning over programs with arbitrary structures under the FLP semantics is $\Pi_2^P$-complete in general, as shown in [7]. Pelov has shown $\Sigma_2^P$-completeness for deciding the existence of PSP answer sets in [13], from which $\Pi_2^P$-completeness for cautious reasoning under the PSP semantics can be derived. We formally state this result now and provide a different proof than Pelov's that will more directly lead to the subsequent considerations.

**Theorem 2.** *Cautious reasoning under PSP semantics is $\Pi_2^P$-complete.*

*Proof.* Membership follows by Corollary 1 of [17]. For the hardness, we provide a reduction from 2-QBF$_\forall$. Let $\Psi = \forall x_1 \cdots \forall x_m \exists y_1 \cdots \exists y_n \; E$, where $E$ is in 3CNF. Formula $\Psi$ is equivalent to $\neg\Psi'$, where $\Psi' = \exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n \; E'$, and $E'$ is a 3DNF equivalent to $\neg E$ and obtained by applying De Morgan's laws. To prove the claim we construct a program $P_\Psi$ such that $P_\Psi \models_c^{PSP} w$ ($w$ a fresh atom) if and only if $\Psi$ is valid, i.e., iff $\Psi'$ is invalid.

Let $E' = (l_{1,1} \wedge l_{1,2} \wedge l_{1,3}) \vee \cdots \vee (l_{k,1} \wedge l_{k,2} \wedge l_{k,3})$, for some $k \geq 1$. Program $P_\Psi$ is the following:

$$
\begin{array}{llll}
x_i^T \leftarrow not\ x_i^F & x_i^F \leftarrow not\ x_i^T & i \in \{1,\ldots,m\} & (4)\\
y_i^T \leftarrow not\ y_i^F & y_i^F \leftarrow not\ y_i^T & i \in \{1,\ldots,n\} & (5)\\
y_i^T \leftarrow sat & y_i^F \leftarrow sat & i \in \{1,\ldots,n\} & (6)\\
sat \leftarrow \mu(E') & & & (7)\\
w \leftarrow not\ sat & & & (8)
\end{array}
$$

where $\mu$ is defined recursively as follows:

- $\mu(E') := (\mu(l_{1,1}) \wedge \mu(l_{1,2}) \wedge \mu(l_{1,3})) \vee \cdots \vee (\mu(l_{k,1}) \wedge \mu(l_{k,2}) \wedge \mu(l_{k,3}))$;
- $\mu(x_i) := x_i^T$ and $\mu(\neg\, x_i) := x_i^F$ for all $i = 1,\ldots,m$;
- $\mu(y_i) := y_i^T$ and $\mu(\neg\, y_i) := y_i^F$ for all $i = 1,\ldots,n$.

Note that structure $\mu(E')$ can also be encoded by means of a *sum* aggregate as shown in [1].

Rules (4)–(5) force each PSP answer set of $P_\Psi$ to contain at least one of $x_i^T$, $x_i^F$ ($i \in \{1,\ldots,m\}$), and one of $y_j^T$, $y_j^F$ ($j \in \{1,\ldots,m\}$), respectively, encoding an assignment of the propositional variables in $\Psi'$. Rules (6) are used to simulate universality of the $y$ variables, as described later. Having an assignment, rule (7) derives $sat$ if the assignment satisfies some disjunct of $E'$ (and hence also $E'$ itself). Finally, rule (8) derives $w$ if $sat$ is false.

We first show that $\Psi$ not valid implies $P_\Psi \not\models_c^{PSP} w$. If $\Psi$ is not valid, $\Psi'$ is valid. Hence, there is an assignment $\nu$ for $x_1,\ldots,x_m$ such that no extension to $y_1,\ldots,y_n$ satisfies $E$, i.e., all these extensions satisfy $E'$. Let us consider the following interpretation

(which is also a model of $P_\Psi$):

$$M = \{x_i^T \mid \nu(x_i) = 1,\ i = 1, \ldots, m\} \cup \{x_i^F \mid \nu(x_i) = 0,\ i = 1, \ldots, m\} \\ \cup\ \{y_i^T, y_i^F \mid i = 1, \ldots, n\} \cup \{sat\}$$

We claim that $M$ is a PSP answer set of $P_\Psi$. In fact, $K_M^{P_\Psi}(\emptyset) \supseteq \{x_i^T \mid \nu(x_i) = 1,\ i = 1, \ldots, m\} \cup \{x_i^F \mid \nu(x_i) = 0,\ i = 1, \ldots, m\}$ because of rules (4) in $P_\Psi^M$. Since any assignment for the $y$s satisfies at least a disjunct of $E'$, from rule (7) we derive $sat \in K_M^{P_\Psi}(K_M^{P_\Psi}(\emptyset))$. Hence, rules (6) force all $y$ atoms to belong to $K_M^{P_\Psi}(K_M^{P_\Psi}(K_M^{P_\Psi}(\emptyset)))$, which is thus the least fixpoint of $K_M^{P_\Psi}$ and coincides with $M$.

Now we show that $P_\Psi \not\models_c^{PSP} w$ implies that $\Psi$ is not valid. To this end, let $M$ be a PSP answer set of $P_\Psi$ such that $w \notin M$. Hence, by rule (8) we have that $M \models sat$. From $sat \in M$ and rules (6), we have $y_i^T, y_i^F \in M$ for all $i = 1, \ldots, n$. And $M$ contains either $x_i^T$ or $x_i^F$ for $i = 1, \ldots, m$ because of rules (4). Suppose by contradiction that $\Psi$ is valid. Thus, for all assignments of $x_1, \ldots, x_m$, there is an assignment for $y_1, \ldots, y_n$ such that $E$ is true, i.e., $E'$ is false. We can show that the least fixpoint of $K_M^{P_\Psi}$ is $K_M^{P_\Psi}(\emptyset) = \{x_i^T \mid \nu(x_i) = 1,\ i = 1, \ldots, m\} \cup \{x_i^F \mid \nu(x_i) = 0,\ i = 1, \ldots, m\}$. In fact, $sat$ cannot be derived because $K_M^{P_\Psi}(\emptyset) \not\models \mu(E')$. We thus have a contradiction with the assumption that $M$ is a PSP answer set of $P_\Psi$. □

It is also known that the complexity drops to $coNP$ if structures in body rules are constrained to be convex. This appears to be "folklore" knowledge and can be argued to follow from results in [10]. An easy way to see membership in $coNP$ is that all convex structures can be decomposed into a conjunction of a monotonic and an antimonotonic structure, for which membership in $coNP$ has been shown in [7].

It is instructive to note a crucial difference between the $\Pi_2^P$-hardness proofs in [7] (and a similar one in [8]) and the proofs for Theorem 2 and the $\Sigma_2^P$ result for PSP in [13].

The fundamental tool in the FLP hardness proofs is the availability of structures $S_1, S_2$ that allow for encoding "need to have either atom $x^T$ or $x^F$, or both of them, but the latter only upon forcing the truth of both atoms." $S_1, S_2$ have domains $D_{S_1} = D_{S_2} = \{x^T, x^F\}$ and the following satisfaction patterns:

$$\begin{array}{llll} \emptyset \models S_1 & \{x^T\} \models S_1 & \{x^F\} \not\models S_1 & \{x^T, x^F\} \models S_1 \\ \emptyset \models S_2 & \{x^T\} \not\models S_2 & \{x^F\} \models S_2 & \{x^T, x^F\} \models S_2 \end{array}$$

The reductions then use these structures in a similar way than disjunction is used in the classic $\Sigma_2^P$-hardness proofs in [3]. In particular, the same structures are used for all instances to be reduced.

On the other hand, in the PSP hardness proofs, one dedicated structure is used for each instance of the problem reduced from (2QBF in Theorem 2). Indeed, a construction using structures $S_1, S_2$ as described earlier is not feasible for PSP, because $(\emptyset, \{x^T, x^F\}) \not\models S_1$ and $(\emptyset, \{x^T, x^F\}) \not\models S_2$. This is because there is one satisfaction "hole" between $\emptyset$ and $\{x^T, x^F\}$ for both $S_1$ and $S_2$. In the next section, we will exploit this difference.

### 3.3 Dissonance: Complexity of Non-convex Structures with Bounded Domains

In this section, we look more carefully at programs with non-convex structures and identify computational differences between the FLP and PSP semantics. In [2] it has been shown that any non-convex structure (plus all of its variants) can be used in order to implement $S_1$ and $S_2$. This result makes it clear that the presence of any non-convex structure that is closed under variants causes a complexity increase for the FLP semantics (unless the polynomial hierarchy collapses). From the above considerations, it is immediately clear that the same construction is not feasible for PSP. It turns out that also no alternative way exists to obtain a similar result, and that the difference in the $\Pi_2^P$-hardness proofs for FLP and PSP is intrinsic.

We start by considering a simple non-convex structure $\mathring{A}$ with $D_{\mathring{A}} = \{x, y\}$ and $I \models \mathring{A}$ if and only if $|I \cap D_{\mathring{A}}| \neq 1$. Therefore, $\mathring{A}$ behaves like a cardinality constraint $COUNT(\{x, y\}) \neq 1$.

**Proposition 1.** *Deciding whether an interpretation $M$ is a PSP answer set of a datalog$^{\{\mathring{A}\}}$ program $P$ is feasible in polynomial time, in particular $DTIME(m^2)$, where $m$ is the number of rules in $P$.*

*Proof.* For any interpretation, testing whether $(I, M) \models \mathring{A}\sigma$ (for a variant $\mathring{A}\sigma$ of $\mathring{A}$) can be done by examining $|I \cap D_{\mathring{A}\sigma}| = i$ and $|M \cap D_{\mathring{A}\sigma}| = j$ and returning false if either one of $i$, $j$ is 1, or if $i = 0$ and $j = 2$. Alternatively, in a less syntax dependent way, one can test whether $M \models \mathring{A}\sigma$ and $(I \cup J) \models \mathring{A}\sigma$ for each $J \subseteq (M \cap D_{\mathring{A}\sigma}) \setminus (I \cap D_{\mathring{A}\sigma})$. Since there are at most 4 different $J$ for each $I$, either method is feasible in constant time.

For determining whether $M$ is a PSP answer set of $P$, we can check whether it is the least fixpoint of $K_M^P$. Computing the least fixpoint takes at most $m$ applications of $K_M^P$ (where $m$ is the number of rules in $P$). Each application of $K_M^P$ involves in turn at most $m$ tests for $(I, M) \models \mathring{A}\sigma$. □

Given Proposition 1 it follows that cautious reasoning is still in $coNP$ for datalog$^{\{\mathring{A}\}}$ programs under the PSP semantics.

**Proposition 2.** *Given a datalog$^{\{\mathring{A}\}}$ program $P$ and an atom $a$, deciding $P \models_c^{PSP} a$ is in $coNP$.*

*Proof.* The complement has an immediate nondeterministic polynomial time algorithm: guess an interpretation $M$ and verify in polynomial time that $a \notin M$ and that $M$ is a PSP answer set of $P$ (by virtue of Proposition 1). □

It follows that for datalog$^{\{\mathring{A}\}}$ cautious reasoning (and also answer set existence and brave reasoning) is more complex for the FLP semantics than for the PSP semantics (unless the polynomial hierarchy collapses to its first level).

Examining this result and its proof carefully, we can see that it depends on the fact that each $D_{\mathring{A}\sigma}$ contains 2 elements and therefore at most 4 satisfaction tests are needed to determine $(I, M) \models \mathring{A}\sigma$. Indeed, we can apply similar reasoning whenever the domains of involved structures are smaller than a given bound.

**Theorem 3.** *Let $P$ be a program. If $k$ is an upper bound for the domain size of any structure occurring in $P$, then checking whether a given interpretation $M$ is a PSP answer set of $P$ is decidable in $DTIME(2^k m^2 p(n))$, where $m$ is the number of rules in $P$ and $p(n)$ is the polynomial function (in terms of the input size $n$) bounding determining satisfaction of any aggregate in $P$.*

*Proof.* We show that the least fixpoint of $K_M^P$ can be computed in time $O(2^k m^2 p(n))$. In the worst case, each application of the operator derives at most one new atom, and thus the fixpoint is reached after at most $m$ applications of the operator. Each application requires at most the evaluation of all rules of $P$, and thus at most $m^2$ rule evaluations are sufficient. To evaluate a rule, the truth of the body has to be checked w.r.t. at most $2^k$ interpretations (similar to Proposition 1, in which $k = 2$), each requiring $p(n)$ time. We thus obtain the bound $O(2^k m^2 p(n))$. □

This means that actually most languages with non-convex structures exhibit a complexity gap between the FLP and PSP semantics. There is a uniformity issue here, which we informally noted earlier when examining the $\Pi_2^P$-hardness proof for cautious reasoning under PSP. We can now formalize this, as it follows from Theorem 3 that we need an infinite number of inherently different non-convex structures in order to obtain $\Pi_2^P$ hardness.

**Corollary 1.** *Let $\mathbf{S}$ be any finite set of structures, possibly including non-convex structures. Cautious reasoning over datalog$^{\mathbf{S}}$ is in $coNP$ under the PSP semantics.*

This means that there is also a clear difference in uniformity between the complexity boundary of the FLP and the PSP semantics, respectively. It also means that it is impossible to simulate the FLP semantics in a compact way using the PSP semantics on the class of programs with bounded domain structures, unless the polynomial hierarchy collapses to its first level. The general picture of our complexity results is shown in Figure 1. We can see that the complexity transition from $coNP$ to $\Pi_2^P$ is different for the FLP and PSP semantics, respectively. The solid line between convex and non-convex structures denotes a crisp transition for FLP, while the dashed line between bounded non-convex and unbounded non-convex structures is a rougher transition.

## 4 Discussion

Looking at Figure 1, the transition from $coNP$ to $\Pi_2^P$ appears somewhat irregular for PSP, as the availability of single non-convex structures does not cause the transition, but only their union. However, in practice the availability of an infinite number of different structures is not unusual: indeed, if aggregates are considered, the presence of one aggregate function and suitable comparison relations usually gives rise to such an infinite repertoire of structures.

*Example 10.* Consider the availability of $COUNT$ over any set of atoms and the comparison relation $\neq$. The structures generated by aggregates of the form $COUNT(S) \neq i$ do not have a bound on the domains of non-convex aggregates. Indeed, for any structure $COUNT(\{a_1, \ldots, a_k\}) \neq 1$, which is non-convex and for which the domain size is $k$,

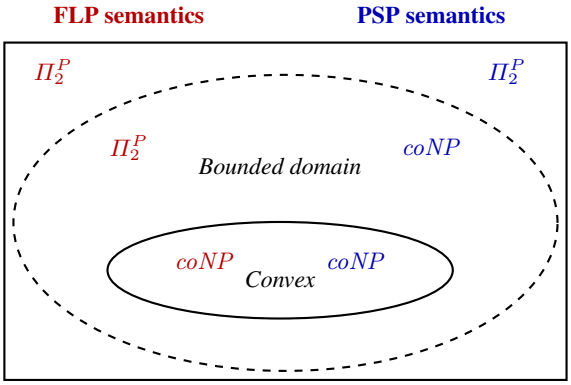



**Fig. 1.** Complexity of cautious reasoning

one can formulate also $COUNT(\{a_1, \ldots, a_{k+1}\}) \neq 1$, which is also non-convex and has a larger domain.

However, as noted earlier, for expressing $\Pi_2^P$-hard problems, one needs a non-uniform approach for PSP, in the sense that a dedicated aggregate has to be formulated for each problem instance, whereas for FLP one can re-use the same aggregates for all problem instances.

In practical terms, our results imply that for programs containing only convex structures, techniques as those presented in [1] for FLP can be used for computing answer sets also for PSP, and techniques presented for PSP can be used for FLP in turn. It also means that this is the largest class for which this can be done with currently available methods in an efficient way. There are several examples for convex structures that are easy to identify syntactically: count aggregates with equality guards, sum aggregates with positive summands and equality guards, dl-atoms that do not involve $\uplus$ and rely on a tractable Description Logic [4]. However many others are in general not convex, for example sum aggregates that involve both positive and negative summands, times aggregates that involve the factor 0, average aggregates, dl-atoms with $\uplus$, and so on. It is still possible to find special cases of such structures that are convex, but that requires deeper analyses.

The results also immediately imply impossibility and possibility results for rewritability: unless the polynomial hierarchy collapses to its first level, it is not possible in the FLP semantics to rewrite a program with non-convex structures into one containing only convex structures (for example, a program not containing any generalized atoms), unless disjunction or similar constructs are allowed in rule heads. On the other hand, such rewritings are possible for the PSP semantics if the non-convex structures are guaranteed to have bounded domains. This seems to be most important for dl-programs, where such rewritings are sought after.

The semantics considered in this paper encompass several approaches suggested for programs that couple answer set programming with description logics. The approaches presented in [5] and [11] directly employ the FLP semantics, while the approach of [15]

is shown to be equivalent to the PSP semantics. There are other proposals, such as [4], which appears to be different from both FLP and PSP already on convex structure. In future work we plan to relate also these other semantics with FLP and PSP and attempt to identify the largest coinciding classes of programs.

## References

1. Alviano, M., Calimeri, F., Faber, W., Leone, N., Perri, S.: Unfounded Sets and Well-Founded Semantics of Answer Set Programs with Aggregates. Journal of Artificial Intelligence Research 42, 487–527 (2011)
2. Alviano, M., Faber, W.: The complexity boundary of answer set programming with generalized atoms under the flp semantics. In: Cabalar, P., Tran, S.C. (eds.) Logic Programming and Nonmonotonic Reasoning — 12th International Conference (LPNMR 2013). Springer Verlag (Sep 2013), accepted for publication
3. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. Annals of Mathematics and Artificial Intelligence 15(3/4), 289–323 (1995)
4. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artif. Intell. 172(12–13), 1495–1539 (2008)
5. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: International Joint Conference on Artificial Intelligence (IJCAI) 2005. pp. 90–96. Edinburgh, UK (Aug 2005)
6. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004). Lecture Notes in AI (LNAI), vol. 3229, pp. 200–212. Springer Verlag (Sep 2004)
7. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence 175(1), 278–298 (2011), special Issue: John McCarthy's Legacy
8. Ferraris, P.: Answer Sets for Propositional Theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings. Lecture Notes in Computer Science, vol. 3662, pp. 119–131. Springer Verlag (2005)
9. Liu, L., Pontelli, E., Son, T.C., Truszczyński, M.: Logic programs with abstract constraint atoms: The role of computations. Artificial Intelligence 174(3–4), 295–315 (2010)
10. Liu, L., Truszczyński, M.: Properties and applications of programs with monotone and convex constraints. Journal of Artificial Intelligence Research 27, 299–334 (2006)
11. Lukasiewicz, T.: A novel combination of answer set programming with description logics for the semantic web. IEEE Transactions on Knowledge and Data Engineering 22(11), 1577–1592 (2010)
12. Marek, V.W., Niemelä, I., Truszczyński, M.: Logic Programming with Monotone Cardinality Atom. In: Lifschitz, V., Niemelä, I. (eds.) Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7). LNAI, vol. 2923, pp. 154–166. Springer (Jan 2004)
13. Pelov, N.: Semantics of Logic Programs with Aggregates. Ph.D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium (Apr 2004)
14. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. Theory and Practice of Logic Programming 7(3), 301–353 (2007)

15. Shen, Y.D., Wang, K.: FLP semantics without circular justifications for general logic programs. In: Hoffmann, J., Selman, B. (eds.) Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI-12) (Jul 2012)
16. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138, 181–234 (Jun 2002)
17. Son, T.C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP. Theory and Practice of Logic Programming 7, 355–375 (May 2007)
18. Son, T.C., Pontelli, E., Tu, P.H.: Answer Sets for Logic Programs with Arbitrary Abstract Constraint Atoms. Journal of Artificial Intelligence Research 29, 353–389 (2007)

# Hybrid Automated Reasoning Tools: from Black-box to Clear-box Integration

Marcello Balduccini[1] and Yulia Lierler[2]

[1] College of Information Science and Technology
Drexel University
marcello.balduccini@gmail.com
[2] Computer Science Department
University of Nebraska at Omaha
ylierler@unomaha.edu

**Abstract** Recently, researchers in answer set programming and constraint programming spent significant efforts in the development of hybrid languages and solving algorithms combining the strengths of these traditionally separate fields. These efforts resulted in a new research area: constraint answer set programming (CASP). CASP languages and systems proved to be largely successful at providing efficient solutions to problems involving hybrid reasoning tasks, such as scheduling problems with elements of planning. Yet, the development of CASP systems is difficult, requiring non-trivial expertise in multiple areas. This suggests a need for a study identifying general development principles of hybrid systems. Once these principles and their implications are well understood, the development of hybrid languages and systems may become a well-established and well-understood routine process. As a step in this direction, in this paper we conduct a case study aimed at evaluating various integration schemas of CASP methods.

## 1 Introduction

Knowledge representation and automated reasoning are areas of Artificial Intelligence dedicated to understanding and automating various aspects of reasoning. Such traditionally separate fields of AI as answer set programming (ASP) [4], propositional satisfiability (SAT) [13], constraint (logic) programming (CSP/CLP) [22,14] are all representatives of directions of research in automated reasoning. The algorithmic techniques developed in subfields of automated reasoning are often suitable for distinct reasoning tasks. For example, answer set programming proved to be an effective tool for formalizing elaborate planning tasks whereas CSP is efficient in solving difficult scheduling problems. Nevertheless, if the task is to solve complex scheduling problems requiring elements of planning then neither ASP nor CSP alone is sufficient. In recent years, researchers attempted to address this problem by developing *hybrid* approaches that combine algorithms and systems from different AI subfields. Research in satisfiability modulo theories (SMT) [21] is a well-known example of this trend.

More recent examples include constraint answer set programming (CASP) [16], which integrates answer set programming with constraint (logic) programming. Constraint answer set programming allows to combine the best of two different automated

reasoning worlds: (1) modeling capabilities of ASP together with advances of its SAT-like technology in solving and (2) constraint processing techniques for effective reasoning over non-boolean constructs. This new area has already demonstrated promising activity, including the development of the CASP solvers ACSOLVER [19], CLINGCON [11], EZCSP [2], and IDP [25]. Related techniques have also been used in the domain of hybrid planning for robotics [23]. CASP opens new horizons for declarative programming applications. Yet the developments in this field pose a number of questions, which also apply to the automated reasoning community as a whole.

The broad attention received by the SMT and CASP paradigms, which aim to integrate and build synergies between diverse constraint technologies, and the success they enjoyed suggest a necessity of a principled and general study of methods to develop such hybrid solvers. [16] provides a study of the relationship between various CASP solvers highlighting the importance of creating unifying approaches to describe such systems. For instance, the CASP systems ACSOLVER, CLINGCON, and EZCSP came to being within two consecutive years. These systems rely on different ASP and CSP technologies, so it is difficult to clearly articulate their similarities and differences. In addition, the CASP solvers adopt different communication schemas among their heterogeneous solving components. The system EZCSP adopts a "*black-box*" architecture, whereas ACSOLVER and CLINGCON advocate tighter integration. The crucial message transpiring from these developments in the CASP community is the ever growing need for standardized techniques to integrate computational methods spanning multiple research areas. Currently such an integration requires nontrivial expertise in multiple areas, for instance, in SAT, ASP and CSP. We argue for undertaking an effort to mitigate difficulties of designing hybrid reasoning systems by identifying general principles for their development and studying the implications of various design choices.

As a step in this direction, in this paper we conduct a case study aiming to explore a crucial aspect in building hybrid systems – the integration schemas of participating solving methods. We study various integration schemas and their performance, using CASP as our test-bed domain. As an exemplary subject for our study we take the CASP system EZCSP. Originally, EZCSP was developed as an inference engine for CASP that allowed a lightweight, *black-box*, integration of ASP and CSP. In order for our analysis to be conclusive we found it important to study the different integration mechanisms using the same technology. Within the course of this work we implemented "*grey-box*" and "*clear-box*" approaches for combining ASP and CSP reasoning within EZCSP. We evaluate these configurations of EZCSP on three domains – Weighted Sequence, Incremental Scheduling, and Reverse Folding – from the *Model and Solve* track of the *Third Answer Set Programming Competition – 2011* (ASPCOMP) [1]. Hybrid paradigms such as CASP allow for mixing language constructs and computational mechanisms stemming from different formalisms. Yet, one may design encodings that favor only single reasoning capabilities of a hybrid system. For this reason, in our study we evaluate different encodings for the proposed benchmarks that we call "pure ASP", "true CASP", and "pure CSP". As a result we expect to draw a comprehensive picture comparing and contrasting various integration schemas on several kinds of encodings possible within hybrid approaches.

We start with a brief review of the CASP formalism. Then we draw a parallel to SMT solving, aimed at showing that it is possible to transfer to SMT the results obtained in this work for CASP solving. In Section 3 we review the integration schemas used in the design of hybrid solvers focusing on the schemas implemented in EZCSP within this project. Section 4 provides a brief introduction to the application domains considered, and discusses the variants of the encodings we compared. Experimental results and their analysis form Section 5.

## 2 Review: the CASP and SMT problems

The review of logic programs with constraint atoms follows the lines of [15]. A *regular program* is a finite set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_l, not\ a_{l+1}, \ldots, not\ a_m,\ not\ not\ a_{m+1}, \ldots, not\ not\ a_n, \qquad (1)$$

where $a_0$ is $\bot$ or an atom, and each $a_i$ ($1 \leq i \leq n$) is an atom. This is a special case of programs with nested expressions [18]. We refer the reader to [18] for details on the definition of an answer set of a logic program. A *choice rule* construct $\{a\}$ [20] of the LPARSE language can be seen as an abbreviation for a rule $a \leftarrow\ not\ not\ a$ [9]. We adopt this abbreviation.

A constraint satisfaction problem (CSP) is defined as a triple $\langle X, D, C\rangle$, where $X$ is a set of variables, $D$ is a domain of values, and $C$ is a set of constraints. Every constraint is a pair $\langle t, R\rangle$, where $t$ is an $n$-tuple of variables and $R$ is an $n$-ary relation on $D$. An *evaluation* of the variables is a function from the set of variables to the domain of values, $\nu : X \to D$. An evaluation $\nu$ *satisfies* a constraint $\langle (x_1, \ldots, x_n), R\rangle$ if $(v(x_1), \ldots, v(x_n)) \in R$. A *solution* is an evaluation that satisfies all constraints.

Consider an alphabet consisting of regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$ respectively. By $\tilde{\mathcal{C}}$, we denote the set of all literals over $\mathcal{C}$. The constraint literals are identified with constraints via a function $\gamma : \tilde{\mathcal{C}} \to C$ so that for any literal $l$, $\gamma(l)$ has a solution if and only if $\gamma(\bar{l})$ does not have one (where $\bar{l}$ denotes a complement of $l$). For a set $Y$ of constraint literals over $\mathcal{C}$, by $\gamma(Y)$ we denote a set of corresponding constraints, i.e., $\{\gamma(c) \mid c \in Y\}$. Furthermore, each variable in $\gamma(Y)$ is associated with a domain. For a set $M$ of literals, by $M^+$ and $M^{\mathcal{C}}$ we denote the set of positive literals in $M$ and the set of constraint literals over $\mathcal{C}$ in $M$, respectively.

A *logic program with constraint atoms* is a regular logic program over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ such that, in rules of the form (1), $a_0$ is $\bot$ or $a_0 \in \mathcal{A}$. Given a logic program with constraint atoms $\Pi$, by $\Pi^{\mathcal{C}}$ we denote $\Pi$ extended with choice rules $\{c\}$ for each constraint atom $c$ occurring in $\Pi$. We say that a consistent and complete set $M$ of literals over atoms of $\Pi$ is an *answer set* of $\Pi$ if

**(a1)** $M^+$ is an answer set of $\Pi^{\mathcal{C}}$ and
**(a2)** $M^{\mathcal{C}}$ has a solution.

The CASP problem is the problem of determining, given a logic program with constraint atoms $\Pi$, whether $\Pi$ has an answer set.

For example, let $\Pi$ be the program

$$\begin{array}{l} am \leftarrow\ X < 12 \\ lightOn \leftarrow\ switch, not\ am \\ \{switch\} \\ \bot \leftarrow not\ lightOn. \end{array} \tag{2}$$

Intuitively, this program states that (a) *light* is *on* only if an action of *switch* occurs during the *pm* hours and (b) *light* is *on* (according to the last rule in the program). Consider a domain of $X$ to be integers from 0 till 24. It is easy to see that a set

$$\{switch, lightOn, \neg\ am, \neg X < 12\}$$

forms the only answer set of program (2).

One may now draw a parallel to satisfiability modulo theories (SMT). To do so we first formally define the SMT problem. A *theory* $T$ is a set of closed first-order formulas. A CNF formula $F$ (a set of clauses) is $T$*-satisfiable* if $F \wedge T$ is satisfiable in the first-order sense. Otherwise, it is called $T$-unsatisfiable. Let $M$ be a set of literals. We sometimes may identify $M$ with a conjunction consisting of all its elements. We say that $M$ is a $T$-model of $F$ if

**(m1)** $M$ is a model of $F$ and
**(m2)** $M$ is $T$*-satisfiable*.

The SMT problem for a theory $T$ is the problem of determining, given a formula $F$, whether $F$ has a $T$-model. It is easy to see that in the CASP problem, $\Pi^C$ in condition (a1) plays the role of $F$ in (m1) in the SMT problem. At the same time, the condition (a2) is similar in nature to the condition (m2).

Given this tight conceptual relation between the SMT and CASP formalisms, it is not surprising that solvers stemming from these different research areas share a lot in common in their design even though these areas have been developing to a large degree independently (CASP being a much younger field). We start next section by reviewing major design principles/methods in crafting SMT solvers. We then discuss how CASP solvers follow one or another method. This discussion allows us to systematize solvers' design patterns present both in SMT and CASP so that their relation becomes clearer. Such transparent view on solvers' architectures immediately translates findings in one area into another. Thus although the case study conducted in this research uses CASP technology only, we expect similar results to hold for SMT, and for the construction of hybrid automated reasoning methods in general.

## 3 SMT/CASP Integration Schemas

Satisfiability modulo theories (SMT) integrates different theories "under one roof". Often it also integrates different computational procedures for processing such hybrid theories. We are interested in these synergic procedures explored by the SMT community over the past decade. We follow [21, Section 3.2] for a review of several integration techniques exploited in SMT.

In every discussed approach, a formula $F$ is treated as a satisfiability formula where each of its atoms is considered as a propositional symbol, *forgetting* about the theory $T$. Such view naturally invites an idea of *lazy* integration: the formula $F$ is given to a SAT solver, if the solver determines that $F$ is unsatisfiable then $F$ is $T$-unsatisfiable as well. Otherwise, a propositional model $M$ of $F$ found by the SAT solver is checked by a specialized $T$-solver which determines whether $M$ is $T$-satisfiable. If so, then it is also a $T$-model of $F$, otherwise $M$ is used to build a clause $C$ that precludes this assignment, i.e., $M \not\models C$ while $F \cup C$ is $T$-satisfiable if and only if $F$ is $T$-satisfiable. The SAT solver is invoked on an augmented formula $F \cup C$. Such process is repeated until the procedure finds a $T$-model or returns unsatisfiable. Note how in this approach two automated reasoning systems – a SAT solver and a specialized $T$-solver – interleave: a SAT solver generates "candidate models" whereas a $T$-solver tests whether these models are in accordance with requirements specified by theory $T$. We find that it is convenient to introduce the following terminology for the future discussion: a *base* solver and a *theory* solver, where a base solver is responsible for generating candidate models and *theory* solver is responsible for any additional testing required for stating whether a candidate model is indeed a solution.

It is easy to see how the lazy integration policy translates into the realm of CASP. Given a program with constraint atoms $\Pi$, an answer set solver serves the role of a base solver by generating answer sets of $\Pi^{\mathcal{C}}$ (that are "candidate answer sets" for $\Pi$) and then uses a CLP/CSP solver as a theory solver to verify whether condition (a2) is satisfied on these candidate answer sets. Constraint answer set solver EZCSP embraces the lazy integration approach in its design.[3] To solve the CASP problem, EZCSP offers a user several options for *base* and *theory* solvers. For instance, it allows for the use of answer set solvers CLASP [10], CMODELS [12], DLV [5] as base solvers and CLP systems SICSTUS PROLOG [24] and BPROLOG [26] as theory solvers. Such variety in possible configurations of EZCSP illustrates how lazy integration provides great flexibility in choosing underlying base and theory solving technology in addressing problems of interest.

The Davis-Putnam-Logemann-Loveland (DPLL) procedure [6] is a backtracking-based search algorithm for deciding the satisfiability of a propositional CNF formula. DPLL-like procedures form the basis for most modern SAT solvers as well as answer set solvers. If a DPLL-like procedure underlies a base solver in the SMT and CASP tasks then it opens a door to several refinements of lazy integration. We now describe these refinements that will also be a focus of the present case study.

In the lazy integration approach a base solver is invoked iteratively. Consider the SMT task: a CNF formula $F_{i+1}$ of the $i+1^{\text{th}}$ iteration to a SAT solver consists of a CNF formula $F_i$ of the $i^{\text{th}}$ iteration and an additional clause (or a set of clauses). Modern DPLL-like solvers commonly implement such technique as *incremental* solving. For instance, incremental SAT-solving allows the user to solve several SAT problems $F_1, \ldots, F_n$ one after another (using single invocation of the solver), if $F_{i+1}$ results from $F_i$ by adding clauses. In turn, the solution to $F_{i+1}$ may benefit from the knowledge obtained during solving $F_1, \ldots, F_i$. Various modern SAT-solvers, including MIN-

[3] [2] refers to lazy integration of EZCSP as *lightweight* integration of ASP and constraint programming.

ISAT [7,8], implement interfaces for incremental SAT solving. Similarly, the answer set solver CMODELS implements an interface that allows the user to solve several ASP problems $\Pi_1, \ldots, \Pi_n$ one after another, if $\Pi_{i+1}$ results from $\Pi_i$ by adding a set of rules whose heads are $\bot$. It is natural to utilize incremental DPLL-like procedures for enhancing the lazy integration protocol: we call this refinement *lazy+* integration. In this approach rather than invoking a base solver from scratch an incremental interface provided by a solver is used to implement the iterative process.

[21] also reviews such integration techniques used in SMT as *on-line SAT solver* and *theory propagation*. In the on-line SAT solver approach, the $T$-satisfiability of the (partial) assignment is checked incrementally, while the assignment is being built by the DPLL-like procedure. This can be done fully eagerly as soon as a change in the partial assignment occurs or at some regular intervals, for instance. Once the inconsistency is detected, a SAT solver is instructed to backtrack. The theory propagation approach extends the on-line SAT solver technique by allowing a theory solver not only to verify that a current partial assignment is $T$-consistent but also to detect literals in a CNF formula that must hold given the current partial assignment. The CASP solver CLINGCON exemplifies the implementation of the theory propagation integration schema in CASP by unifying answer set solver CLASP as a base solver and constraint processing system GECODE. ACSOLVER and IDP systems are other CASP solvers that implement the theory propagation integration schema.

**Three Kinds of** EZCSP**:** To conduct our analysis of various integration schemas and their effect on the performance of the hybrid systems we used the CASP solver EZCSP as a baseline technology. As mentioned earlier, original EZCSP implements the lazy integration schema. In the course of this work we developed enhanced interfaces with answer set solver CMODELS that allowed for the two other integration schemas: lazy+ integration and on-line SAT solver. These implementations rely on API interfaces provided by CMODELS allowing for varying level of integration between the solvers. The development of these API interfaces in CMODELS was greatly facilitated by the API interface provided by MINISAT v. 1.12b supporting non-clausal constraints [8]. In the following we call

- EZCSP implementing lazy integration with CMODELS as a base solver – a *black-box*.
- EZCSP implementing lazy+ integration with CMODELS – a *grey-box*.
- EZCSP implementing on-line SAT solver integration with CMODELS (fully eagerly) – a *clear-box*.

In all these configurations of EZCSP we assume BPROLOG to serve in the role of a theory solver.

## 4 Application Domains

In this work we compare and contrast different integration schemas of hybrid solvers on three application domains that stem from various subareas of computer science. This section provides a brief overview of these applications. All benchmark domains are from the *Third Answer Set Programming Competition – 2011* (ASPCOMP) [1], in

particular, the *Model and Solve* track. We chose these domains for our investigation as they display features that benefit from the synergy of computational methods in ASP and CSP. Each considered problem contains variables ranging over a large integer domain thus making grounding required in pure ASP a bottleneck. On the other hand, the modeling capabilities of ASP and availability of such sophisticated solving technique as learning makes ASP attractive for designing solutions to these domains. As a result, CASP languages and solvers become a natural choice for these benchmarks making them ideal for our investigation.

**Three Kinds of CASP Encodings:** It is easy to note that hybrid languages such as CASP allow for mix-and-match constructs and processing techniques stemming from different formalisms. Yet, any pure ASP encoding of a problem is also a CASP formalization of the same problem. Similarly, it is possible to encode a problem in such a way that only the CSP solving capabilities of the CASP paradigm are employed. In this study for two of the benchmarks we considered three kinds of encodings in the CASP language of EZCSP: *pure-ASP* encoding; *pure-CSP* encoding; and *true-CASP* encoding. In the third benchmark, the use of three distinct encodings was not possible because both ASP and CSP features play a major role in the efficiency of the computation.

Analysis of these varying kinds of encodings in CASP gives us a better perspective on how different integration schemas are effected by the design choices made during the encoding of a problem. At the same time considering the encoding variety allows us to verify our intuition that true-CASP is an appropriate modeling and solving choice for the explored domains.

The **weighted-sequence** (WSEQ) domain is a handcrafted benchmark problem. Its key features are inspired by the important industrial problem of finding an optimal join order by cost-based query optimizers in database systems. [17] provides a complete description of the problem itself as well as the formalization that became "golden standard" in this work, i.e., the formalization named SEQ++.

In the weighted-sequence problem we are given a set of leaves (nodes) and an integer $m$ – maximum cost. Each leaf is a pair *(weight, cardinality)* where *weight* and *cardinality* are integers. Every sequence (permutation) of leaves is such that all leaves but the first are assigned a *color* that, in turn, associates a leaf with a *cost* (via a cost formula). A colored sequence is associated with the *cost* that is a sum of leaves' costs. The task is to find a colored sequence with cost at most $m$. We refer the reader to [17] for the details of pure-ASP encoding SEQ++. The same paper also contains the details on a true-CASP variant of SEQ++ in the language of CLINGCON. We further adapted that encoding to the language of EZCSP by means of simple syntactic transformations. Here we provide a review of details of the SEQ++ formalization that we find most relevant to this presentation. The non-domain predicates of the pure-ASP encoding are $leafPos$, $posColor$, $posCost$. Intuitively, $leafPos$ is responsible for assigning a position to a leaf, $posColor$ is responsible for assigning a color to each position, $posCost$ carries information on costs associated with each leaf. The main difference between the pure-ASP and true-CASP encodings is in the treatment of the cost values of the leaves. We first note that cost predicate $posCost$ in the pure-ASP encoding is "functional". In other words, when this predicate occurs in an answer set its first argument uniquely determines its second argument. Often, such functional predicates in ASP en-

codings can be replaced by constraint atoms in CASP encodings. Indeed, this is the case in the weighted-sequence problem. Thus in the true-CASP encoding, predicate $posCost$ is replaced by constraint atoms, making it possible to evaluate cost values by CSP techniques. This approach is expected to benefit performance especially when the cost values are large. Predicates $leafPos$ and $posColor$ are also functional. The pure-CSP encoding is obtained from the true-CASP encoding by replacing $leafPos$ and $posColor$ predicates by constraint atoms.

The **incremental scheduling** (IS) domain stems from a problem occurring in commercial printing. In this domain, a schedule is maintained up-to-date with respect to jobs being added and equipment going offline. A problem description includes a set of devices, each with predefined number of instances (slots for jobs), and a set of jobs to be produced. The penalty for a job being tardy is computed as $td \cdot imp$, where $td$ is the job's tardiness and $imp$ is a positive integer denoting the job's importance. The total penalty of a schedule is the sum of the penalties of the jobs. The task is to find a schedule whose total penalty is no larger than the value specified in a problem instance. We direct the reader to [3] for a complete description of the domain. The pure-CSP encoding used in our experiments is the official competition encoding submitted to ASPCOMP by the EZCSP team. In this encoding, constraint atoms are used for (i) assigning start times to jobs, (ii) selecting which device instance will perform a job, and (iii) calculating tardiness and penalties. The true-CASP encoding was obtained from the pure-CSP encoding by introducing a new relation $on\_instance(j, i)$, stating that job $j$ runs on device-instance $i$. This relation and ASP constructs of the EZCSP language replaced the constraint atoms responsible for the assignment of device instances in the pure-CSP encoding. The pure-ASP encoding was obtained from the true-CASP encoding by introducing suitable new relations, such as $start(j, s)$ and $penalty(j, p)$, to replace all the remaining constraint atoms.

In the **reverse folding** (RF) domain, one manipulates a sequence of $n$ pairwise connected segments located on a 2D plane in order to take the sequence from an initial configuration to a goal configuration. The sequence is manipulated by pivot moves: rotations of a segment around its starting point by 90 degree in either direction. A pivot move on a segment causes the segments that follow to rotate around the same center. Concurrent pivot moves are prohibited. At the end of each move, the segments in the sequence must not intersect. A problem instance specifies the number of segments, the goal configuration, and required number of moves, $t$. The task is to find a sequence of exactly $t$ pivot moves that produces the goal configuration. The true-CASP encoding used for our experiments is from the official ASPCOMP 2011 submission package of the EZCSP team. In this encoding, relation $pivot(s, i, d)$ states that at step $s$ the $i^{th}$ segment is rotated in direction $d$. The effects of pivot moves are described by constraint atoms, which allow carrying out the corresponding calculations with CSP techniques. The pure-ASP encoding was obtained from the true-CASP encoding by adopting an ASP-based formalization of the effects of pivot moves. This was accomplished by introducing two new relations, $tfoldx(s, i, x)$ and $tfoldy(s, i, y)$, stating that the new start of segment $i$ at step $s$ is $\langle x, y\rangle$. The definition of the relations is provided by suitable ASP rules.

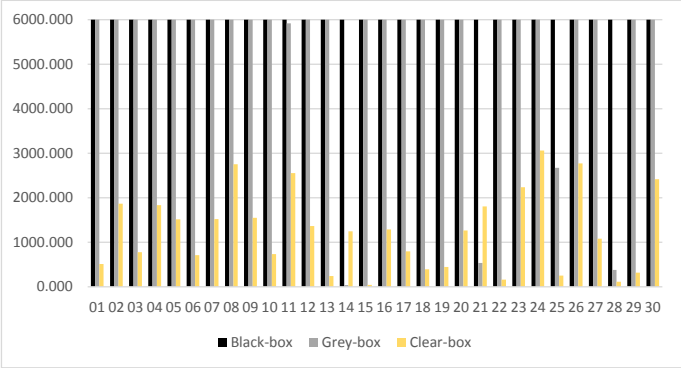



**Figure 1.** Performance on WSEQ domain: true-CASP encoding

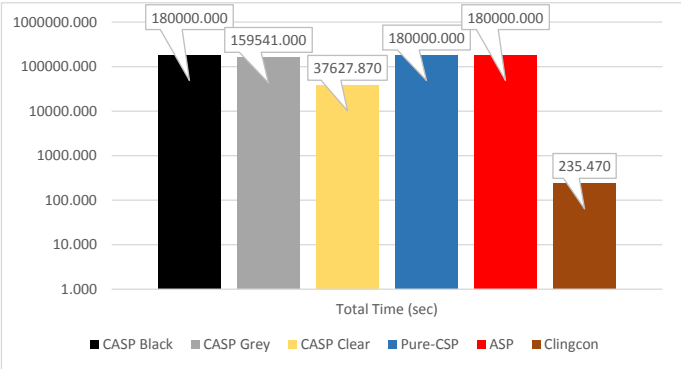



**Figure 2.** Performance on WSEQ domain: total times in logarithmic scale

## 5 Experimental Results

The experimental comparison of the integration schemas was conducted on a computer with an Intel Core i7 processor running at 3GHz. Memory limit for each process and timeout considered were 3 GB RAM and $6,000$ seconds respectively. The version of EZCSP used in the experiments was 1.6.20b49: it incorporated CMODELS version 3.83 as a base solver and BPROLOG 7.4_3 as a theory solver. Answer set solver CMODELS 3.83 was also used for the experiments with the pure-ASP encodings. In order to provide a frame of reference with respect to the state of the art in CASP, the tables for WSEQ and IS include performance information for CLINGCON 2.0.3 on true-CASP encodings adapted to the language of CLINGCON. The ezcsp executable used in the experiments and the encodings can be downloaded from *http://www.mbalduccini.tk/ezcsp/aspocp2013/ezcsp-binaries.tgz* and

*http://www.mbalduccini.tk/ezcsp/aspocp2013/experiments.tgz* respectively. In all figures presented:

- CASP Black, CASP Grey, CASP Clear denote EZCSP implementing respectively *black-box*, *grey-box* and *clear-box*, and running a true-CASP encoding;
- Pure-CSP denotes EZCSP implementing *black-box* running a pure-CSP encoding (note that for pure-CSP encodings there is no difference in performance between the integration schemas);
- ASP denotes CMODELS running a pure-ASP encoding;
- Clingcon denotes CLINGCON running a true-CASP encoding.

We begin our analysis with WSEQ. The instances used in the experiments are the 30 instances available via ASPCOMP. WSEQ proves to be a domain that truly requires the interaction of the ASP and CSP solvers. Answer set solver CMODELS on the pure-ASP encoding runs out of memory on every instance (in the tables, out-of-memory conditions and timeouts are both rendered as out-of-time results). EZCSP on the pure-CSP encoding reaches the timeout limit on every instance. The true-CASP encoding running in *black-box* also times out on every instance. As shown in Figure 1, the true-CASP encoding running in *grey-box* performs slightly better. The true-CASP encoding running in *clear-box* instead performs *substantially* better. Figure 2 reports the total times across all the instances for all solvers/encodings pairs considered. Notably, CASP solver CLINGCON on true-CASP encoding is several orders of magnitude faster than any other configuration. This confirms that for this domain tight integration schemas indeed have an advantage. Recall that CLINGCON implements a tighter integration schema than that of EZCSP *clear-box* that, in addition to the on-line SAT solver schema of *clear-box*, also includes theory propagation. Answer set solver CLASP serves the role of base solver of CLINGCON whereas GECODE is the theory solver.

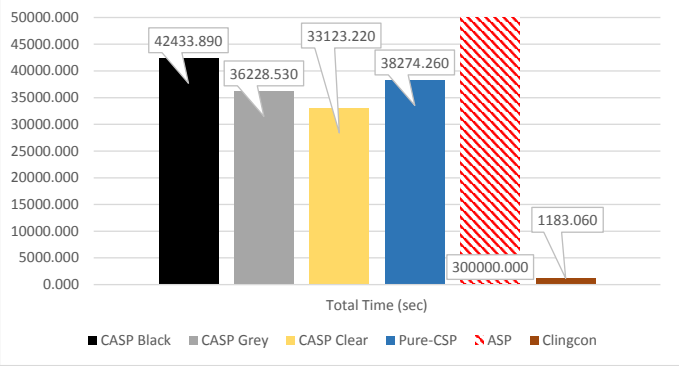



**Figure 3.** Performance on IS domain, easy instances: total times (ASP encoding off-chart)

In case of the IS domain we considered two sets of experiments. In the former we used the 50 official instances from ASPCOMP. We refer to these instances as *easy*.

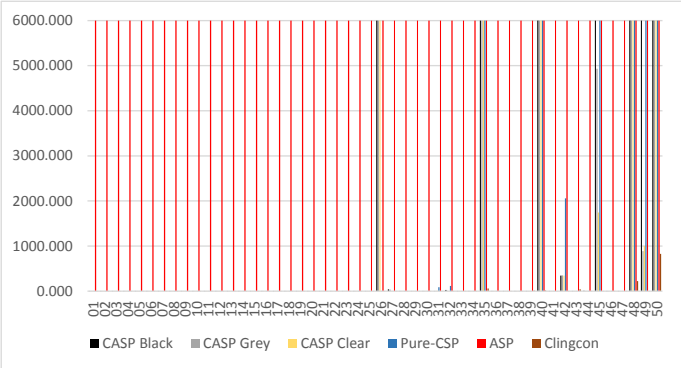



**Figure 4.** Performance on IS domain, easy instances: overall view

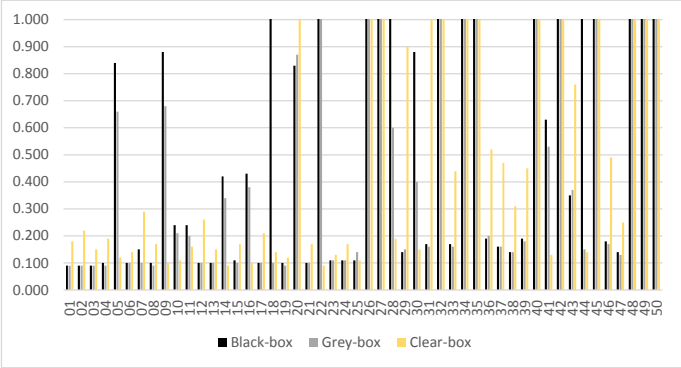



**Figure 5.** Performance on IS domain, easy instances: true-CASP encoding (detail of 0-1sec execution time range)

Figure 4 depicts the overall per-instance performance on the IS-easy domain. It appears that tight integration schemas have an advantage, allowing the true-CASP encoding to outperform the pure-CSP encoding. As one might expect, the best performance for the true-CASP encoding is obtained with the *clear-box* integration schema, as shown in Figure 3 and in Figure 5. Figure 3 provides a comparison of the total times. In this case the early pruning of the search space made possible by the *clear-box* architecture yields substantial benefits. As expected, it is also the case that *grey-box* is faster than *black-box*. As for WSEQ, CLINGCON is the fastest, and CMODELS on the pure-ASP encoding runs out of memory in all the instances.

The next experiment reveals an interesting change in behavior of solver/encodings pairs as the complexity of the instances of the IS domain grows. In this test, we used a set of 30 instances obtained by (1) generating randomly 500 fresh instances; (2) running the true-CASP encoding with the *grey-box* integration schema on them with a timeout

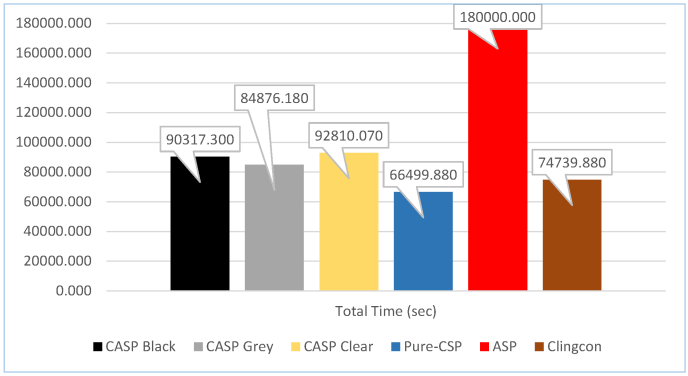



**Figure 6.** Performance on IS domain, hard instances: overall view

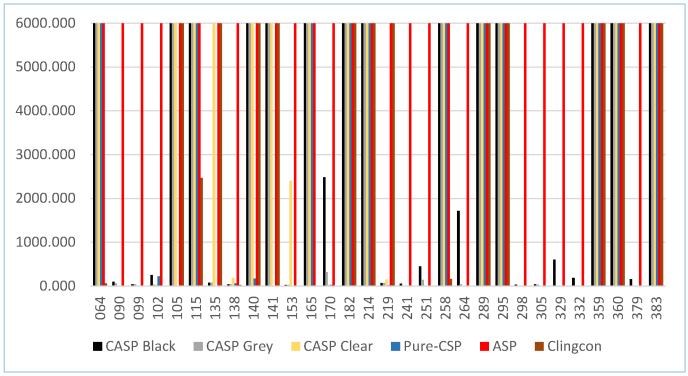



**Figure 7.** Performance on IS domain, hard instances: total times

of 300 seconds; (3) selecting randomly, from those, 15 instances that resulted in time-out and 15 instances that were solved in 25 seconds or more. The numerical parameters used in the process were selected with the purpose of identifying challenging instances. The overall per-instance execution times reported in Figure 7 clearly indicate the level of difficulty of the selected instances. Remarkably, these more difficult instances are solved more efficiently by the pure-CSP encoding that relies only on the CSP solver, as evidenced by Figure 6. In fact, the pure-CSP encoding outperforms every other method of computation, *including* CLINGCON on true-CASP encoding. More specifically, solving the instances with the true-CASP encoding takes between 30% and 50% longer than with the pure-CSP encoding. (Once again, CMODELS runs out of memory.)

The final experiment focuses on the RF domain. We used the 50 official instances from ASPCOMP to conduct the analysis. Figure 9 presented shows that this domain is comparatively easy. Figure 10 illustrates that the *black-box* and *grey-box* integration schemas are several orders of magnitude faster than *clear-box*. This somewhat surpris-

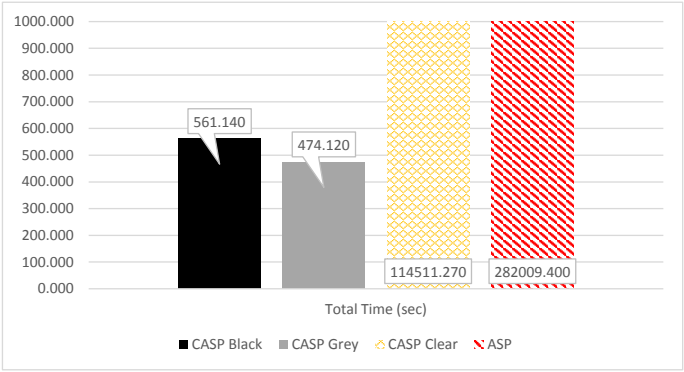



**Figure 8.** Performance on RF domain: total times (detail of 0-1000sec execution time range, ASP and *clear-box* off-chart)

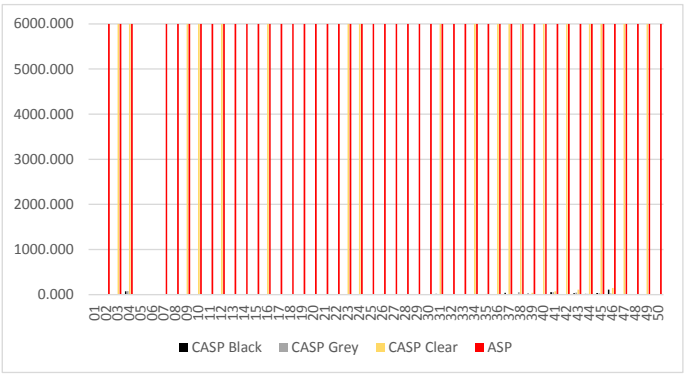



**Figure 9.** Performance on RF domain: overall view

ing result can be explained by the fact that in this domain frequent checks with the theory solver add more overhead rather than being of help in identifying an earlier point to backtrack. CMODELS on the pure-ASP encoding runs out of memory in all but 3 instances. The total execution times are presented in Figure 8.

## 6 Conclusions

The case study conducted in this work clearly illustrates the influence that integration methods have on the behavior of hybrid systems. Each integration schema may be of use and importance for some domain. Thus systematic means ought to be found for facilitating building hybrid systems supporting various coupling mechanisms. Building clear and flexible API interfaces allowing for various types of interactions between the solvers seems a necessary step towards making the development of hybrid solving

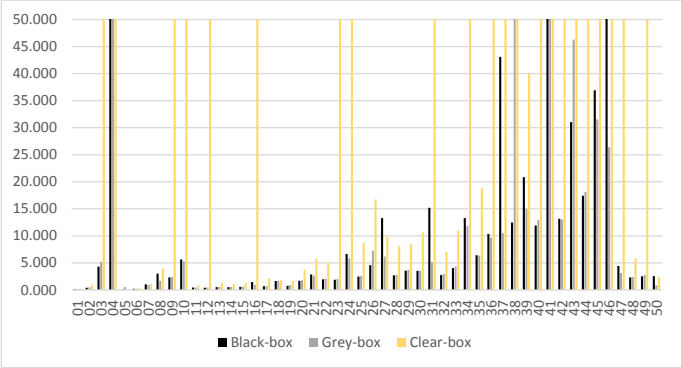



**Figure 10.** Performance on RF domain: true-CASP encoding, detail of 0-0.50sec execution time range

systems effective. This work provides evidence for the need of an effort to this ultimate goal.

## References

1. Third answer set programming competition (2011), https://www.mat.unical.it/aspcomp2011/
2. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In: Proceedings of ICLP'09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09) (2009)
3. Balduccini, M.: Industrial-Size Scheduling with ASP+CP. In: Delgrande, J.P., Faber, W. (eds.) 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR11). Lecture Notes in Artificial Intelligence (LNCS), vol. 6645, pp. 284–296. Springer Verlag, Berlin (2011)
4. Brewka, G., Niemelä, I., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM 54(12), 92–103 (2011)
5. Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The DLV system: Model generator and application frontends. In: Proceedings of Workshop on Logic Programming (WLP97) (1997)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5(7), 394–397 (1962)
7. Een, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. In: SAT (2005)
8. Een, N., Sörensson, N.: An extensible sat-solver. In: SAT (2003)
9. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming 5, 45–74 (2005)
10. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392. MIT Press (2007)
11. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proceedings of 25th International Conference on Logic Programming (ICLP). pp. 235–249. Springer (2009)

12. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36, 345–377 (2006)
13. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability solvers. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 89–134. Elsevier (2008)
14. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19(20), 503–581 (1994)
15. Lierler, Y.: Constraint answer set programming (2012), http://www.cs.utexas.edu/users/ai-lab/pub-view.php?PubID=127221
16. Lierler, Y.: On the relation of constraint answer set programming languages and algorithms. In: Proceedings of the 26th Conference on Artificial Intelligence (AAAI-12). MIT Press (2012)
17. Lierler, Y., Smith, S., Truszczynski, M., Westlund, A.: Weighted-sequence problem: Asp vs casp and declarative vs problem oriented solving. In: Fourteenth International Symposium on Practical Aspects of Declarative Languages (2012), http://www.cs.utexas.edu/users/ai-lab/pub-view.php?PubID=127085
18. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence 25, 369–389 (1999)
19. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence (2008)
20. Niemelä, I., Simons, P.: Extending the Smodels system with cardinality and weight constraints. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 491–521. Kluwer (2000)
21. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
22. Rossi, F., van Beck, P., Walsh, T.: Constraint porgramming. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 181–212. Elsevier (2008)
23. Schuller, P., Patoglu, V., Erdem, E.: A Systematic Analysis of Levels of Integration between Low-Level Reasoning and Task Planning. In: Workshop on Combining Task and Motion Planning at the IEEE International Conference on Robotics and Automation 2013 (2013)
24. SICStus: Sicstus Prolog Web Site (2008), http://www.sics.se/isl/sicstuswww/site/
25. Wittocx, J., Mariën, M., Denecker, M.: The IDP system: a model expansion system for an extension of classical logic. In: LaSh. pp. 153–165 (2008)
26. Zhou, N.F.: The language features and architecture of B-Prolog. Journal of Theory and Practice of Logic Programming (TPLP) 12(1–2), 189–218 (Jan 2012)

# Aspartame: Solving Constraint Satisfaction Problems with Answer Set Programming

M. Banbara[1], M. Gebser[2], K. Inoue[3], T. Schaub*[2], T. Soh[1], N. Tamura[1], and M. Weise[2]

[1] University of Kobe
[2] University of Potsdam
[3] National Institute of Informatics Tokyo

**Abstract.** Encoding finite linear CSPs as Boolean formulas and solving them by using modern SAT solvers has proven to be highly effective, as exemplified by the award-winning *sugar* system. We here develop an alternative approach based on ASP. This allows us to use first-order encodings providing us with a high degree of flexibility for easy experimentation with different implementations. The resulting system *aspartame* re-uses parts of *sugar* for parsing and normalizing CSPs. The obtained set of facts is then combined with an ASP encoding that can be grounded and solved by off-the-shelf ASP systems. We establish the competitiveness of our approach by empirically contrasting *aspartame* and *sugar*.

## 1 Introduction

Encoding finite linear Constraint Satisfaction Problems (CSPs; [1, 2]) as propositional formulas and solving them by using modern solvers for Satisfiability Testing (SAT; [3]) has proven to be a highly effective approach, as demonstrated by the award-winning *sugar*[4] system. The CSP solver *sugar* reads a CSP instance and transforms it into a propositional formula in Conjunctive Normal Form (CNF). The translation relies on the order encoding [4, 5], and the resulting CNF formula can be solved by an off-the-shelf SAT solver.

In what follows, we elaborate upon an alternative approach based on Answer Set Programming (ASP; [6]) and present the resulting CSP solver *aspartame*[5]. The major difference between *sugar* and *aspartame* rests upon the implementation of the translation of CSPs into Boolean constraint problems. While *sugar* implements a translation into CNF in the imperative programming language JAVA, *aspartame* starts with a translation into a set of facts.[6] In turn, these facts are combined with a general-purpose ASP encoding for CSP solving (also based on the order encoding), which is subsequently instantiated by an off-the-shelf ASP grounder. The resulting propositional logic program is then solved by an off-the-shelf ASP solver.

---

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

[4] `http://bach.istc.kobe-u.ac.jp/sugar`

[5] `http://www.cs.uni-potsdam.de/wv/aspartame`

[6] In practice, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing CSPs.

The high-level approach of ASP has obvious advantages. First, instantiation is done by general-purpose ASP grounders rather than dedicated implementations. Second, the elaboration tolerance of ASP allows for easy maintenance and modifications of encodings. And finally, it is easy to experiment with novel or heterogeneous encodings. However, the intruding question is whether the high-level approach of *aspartame* matches the performance of the more dedicated *sugar* system. We empirically address this question by contrasting the performance of both CSP solvers, while fixing the back-end solver to *clasp*, used as both a SAT and an ASP solver.

From an ASP perspective, we gain insights into advanced modeling techniques for solving CSPs. The ASP encoding implementing CSP solving with *aspartame* has the following features:

- usage of function terms to abbreviate structural subsums
- avoidance of (artificial) intermediate Integer variables (to break sum expressions)
- order encoding applied to structural subsum variables (as well as input variables)
- encoding-wise filtering of relevant threshold values (no blind usage of domains)
- customizable "pigeon-hole constraint" encoding for alldifferent constraints
- "smart" encoding of table constraints, tracing admissible tuples along arguments

In the sequel, we assume some familiarity with ASP, its semantics as well as its basic language constructs. A comprehensive treatment of ASP can be found in [6], one oriented towards ASP solving is given in [7]. Our encodings are given in the language of *gringo* 3 [8]. Although we provide essential definitions of CSPs in the next section, we refer the reader to the literature [1, 2] for a broader perspective.

## 2 Background

A *Constraint Satisfaction Problem* (CSP) is given by a pair $(\mathcal{V}, \mathcal{C})$ consisting of a set $\mathcal{V}$ of *variables* and a set $\mathcal{C}$ of *constraint clauses*. Every variable $x \in \mathcal{V}$ has an associated finite *domain* $D(x)$ such that either $D(x) = \{\top, \bot\}$ or $\emptyset \subset D(x) \subseteq \mathbb{Z}$; $x$ is a *Boolean variable* if $D(x) = \{\top, \bot\}$, and an *Integer variable* otherwise. We denote the set of Boolean variables in $\mathcal{V}$ by $\mathcal{B}(\mathcal{V})$ and the set of Integer variables in $\mathcal{V}$ by $\mathcal{I}(\mathcal{V})$. A constraint clause $C \in \mathcal{C}$ is a set of literals over Boolean variables in $\mathcal{B}(\mathcal{V})$ as well as linear inequalities or global constraints on Integer variables in $\mathcal{I}(\mathcal{V})$. Any *literal* in $C$ is of the form $e$ or $\overline{e}$, where $e$ is either a Boolean variable in $\mathcal{B}(\mathcal{V})$, a linear inequality, or a global constraint. A *linear inequality* is an expression $\sum_{1 \leq i \leq n} a_i x_i \leq m$ in which $m$ as well as all $a_i$ for $1 \leq i \leq n$ are Integer constants and $x_1, \ldots, x_n$ are Integer variables in $\mathcal{I}(\mathcal{V})$. A *global constraint* (cf. [9]) is an arbitrary relation over Integer variables in $\mathcal{I}(\mathcal{V})$; we here restrict ourselves to table and alldifferent constraints over subsets $\{x_1, \ldots, x_n\}$ of the Integer variables in $\mathcal{I}(\mathcal{V})$, where a *table constraint* specifies tuples $(d_1, \ldots, d_n) \in D(x_1) \times \cdots \times D(x_n)$ of admitted value combinations and *alldifferent* applies if $x_1, \ldots, x_n$ are assigned to distinct values in their respective domains.[7]

[7] Linear inequalities relying on further comparison operators, such as $<$, $>$, $\geq$, $=$, and $\neq$, can be converted into the considered format via appropriate replacements [5]. Moreover, note that we here limit the consideration of global constraints to the ones that are directly, i.e., without normalization by *sugar*, supported in our prototypical ASP encodings shipped with *aspartame*.

Given a CSP $(\mathcal{V}, \mathcal{C})$, a *variable assignment* $v$ is a (total) mapping $v : \mathcal{V} \to \bigcup_{x \in \mathcal{V}} D(x)$ such that $v(x) \in D(x)$ for every $x \in \mathcal{V}$. A Boolean variable $x \in \mathcal{B}(\mathcal{V})$ is *satisfied* w.r.t. $v$ if $v(x) = \top$. Likewise, a linear inequality $\sum_{1 \leq i \leq n} a_i x_i \leq m$ is *satisfied* w.r.t. $v$ if $\sum_{1 \leq i \leq n} a_i v(x_i) \leq m$. Table constraints $e \subseteq D(x_1) \times \cdots \times D(x_n)$ and alldifferent constraints over subsets $\{x_1, \ldots, x_n\}$ of $\mathcal{I}(\mathcal{V})$ are *satisfied* w.r.t. $v$ if $(v(x_1), \ldots, v(x_n)) \in e$ or $v(x_i) \neq v(x_j)$ for all $1 \leq i < j \leq n$, respectively. Any Boolean variable, linear inequality, or global constraint that is not satisfied w.r.t. $v$ is *unsatisfied* w.r.t. $v$. A constraint clause $C \in \mathcal{C}$ is *satisfied* w.r.t. $v$ if there is some literal $e \in C$ (or $\overline{e} \in C$) such that $e$ is satisfied (or unsatisfied) w.r.t. $v$. The assignment $v$ is a *solution* for $(\mathcal{V}, \mathcal{C})$ if every $C \in \mathcal{C}$ is satisfied w.r.t. $v$.

*Example 1.* Consider a CSP $(\mathcal{V}, \mathcal{C})$ with Boolean and Integer variables $\mathcal{B}(\mathcal{V}) = \{b\}$ and $\mathcal{I}(\mathcal{V}) = \{x, y, z\}$, where $D(x) = D(y) = D(z) = \{1, 2, 3\}$, and constraint clauses $\mathcal{C} = \{C_1, C_2, C_3\}$ as follows:

$$C_1 = \{\mathit{alldifferent}(x, y, z)\} \tag{1}$$

$$C_2 = \{b, 4x - 3y + z \leq 0\} \tag{2}$$

$$C_3 = \left\{\overline{b}, (x, y) \in \{(1, 3), (2, 2), (3, 1)\}\right\} \tag{3}$$

The alldifferent constraint in $C_1$ requires values assigned to $x$, $y$, and $z$ to be mutually distinct. Respective assignments $v$ satisfying the linear inequality $4x - 3y + z \leq 0$ in $C_2$ include $v(x) = 2$, $v(y) = 3$, and $v(z) = 1$ or $v(x) = 1$, $v(y) = 3$, and $v(z) = 2$, while the table constraint in $C_3$ is satisfied w.r.t. assignments $v$ containing $v(x) = 1$, $v(y) = 3$, and $v(z) = 2$ or $v(x) = 3$, $v(y) = 1$, and $v(z) = 2$. In view of the Boolean variable $b$, whose value allows for "switching" between the linear inequality in $C_2$ and the table constraint in $C_3$, we obtain the following solutions $v_1, \ldots, v_4$ for $(\mathcal{V}, \mathcal{C})$:

| | $b$ | $x$ | $y$ | $z$ |
|---|---|---|---|---|
| $v_1$ | $\bot$ | 2 | 3 | 1 |
| $v_2$ | $\bot$ | 1 | 3 | 2 |
| $v_3$ | $\top$ | 1 | 3 | 2 |
| $v_4$ | $\top$ | 3 | 1 | 2 |

## 3 Approach

The *aspartame* tool extends the SAT-based solver *sugar* by an output component to represent a CSP in terms of ASP facts. The generated facts can then, as usual, be combined with a first-order encoding processable with off-the-shelf ASP systems. In what follows, we describe the format of facts generated by *aspartame*, and we present a dedicated ASP encoding utilizing function terms to capture substructures in CSP instances.

### 3.1 Fact Format

Facts express the variables and constraints of a CSP instance in the syntax of ASP grounders like *gringo* [8]. Their format is easiest explained on the CSP from Example 1, whose fact representation is shown in Listing 1. While facts of the predicate `var`/2

```
1  var(bool,b).  var(int,x;y;z,range(1,3)).

3  constraint(1,global(alldifferent,arg(x,arg(y,arg(z,nil))))).
4  constraint(2,b).
5  constraint(2,op(le,op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),0)).
6  constraint(3,op(neg,b)).
7  constraint(3,rel(r,arg(x,arg(y,nil)))).

9  rel(r,2,3,supports).
10 tuple(r,1,1,1).  tuple(r,1,2,3).
11 tuple(r,2,1,2).  tuple(r,2,2,2).
12 tuple(r,3,1,3).  tuple(r,3,2,1).
```

**Listing 1.** Facts representing the CSP from Example 1.

provide labels of Boolean variables like $b$, the predicate `var`/3 includes a third argument for declaring the domains of Integer variables like $x$, $y$, and $z$. Domain declarations rely on function terms `range(`$l$`,`$u$`)`, standing for continuous Integer intervals $[l, u]$. While one term, `range(1,3)`, suffices for the common domain $\{1, 2, 3\}$ of $x$, $y$, and $z$, in general, several intervals can be specified (via separate facts) to form non-continuous domains. Note that the interval format for Integer domains offers a compact fact representation of (continuous) domains; e.g., the single term `range(1,10000)` captures a domain with 10000 elements. Furthermore, the usage of meaningful function terms avoids any need for artificial labels to refer to domains or parts thereof.

The literals of constraint clauses are also represented by means of function terms. In fact, the second argument of `constraint`/2 in Line 3 of Listing 1 stands for $\mathit{alldifferent}(x, y, z)$ from the constraint clause $C_1$ in (1), which is identified via the first argument of `constraint`/2. Since every fact of the predicate `constraint`/2 is supposed to describe a single literal only, constraint clause identifiers establish the connection between individual literals of a clause. This can be observed on the facts in Line 4–7, specifying literals belonging to the binary constraint clauses $C_2$ and $C_3$ in (2) and (3). Here, the terms `b` and `op(neg,b)` refer to the literals $b$ and $\overline{b}$ over Boolean variable $b$, where `op(neg,`$e$`)` is the general notation of $\overline{e}$ for all (supported) constraint expressions $e$. The more complex term of the form `op(le,`$\Sigma$`,`$m$`)` in Line 5 stands for a linear inequality $\Sigma \leq m$. In particular, the inequality $4x - 3y + z \leq 0$ from $C_2$ is represented by nested `op(add,`$\Sigma$`,`$ax$`)` terms whose last argument $ax$ and deepest $\Sigma$ part are of the form `op(mul,`$a$`,`$x$`)`; such nesting corresponds to the precedence $(((4{*}x){+}(-3{*}y)){+}(1{*}z)) \leq 0$. The representation by function terms captures linear inequalities of arbitrary arity and, as with Integer intervals, associates (sub)sums with canonical labels. Currently, the order of arguments $ax$ is by variable labels $x$, while more "clever" orders may be established in the future.

The function terms expressing table and alldifferent constraints both include an argument list of the form `arg(`$x_1$`,arg(`$\ldots$`,arg(`$x_n$`,nil)`$\ldots$`))`, in which $x_1, \ldots, x_n$ refer to Integer variables. In Line 3 of Listing 1, an alldifferent constraint over arguments $\boldsymbol{x}$ is declared via `global(alldifferent,`$\boldsymbol{x}$`)`; at present, `alldifferent` is a fixed keyword in facts generated by *aspartame*, but support for other kinds of global constraints can be added in the future. Beyond an argument list $\boldsymbol{x}$, function terms of the form `rel(`$r$`,`$\boldsymbol{x}$`)` also include an identifier $r$ referring to a collection of table constraint tuples. For instance, the corresponding argument `r` in Line 7 ad-

dresses the tuples specified by the facts in Line 9–12. Here, `rel(r,2,3,supports)` declares that `r` is of arity 2 and includes 3 tuples, provided as white list entries via facts of the form `tuple(r,`$t$`,`$i$`,`$d$`)`. The latter include tuple and argument identifiers $t$ and $i$ along with a value $d$. Accordingly, the facts in Line 10, 11, and 12 specify the pairs $(1,3)$, $(2,2)$, and $(3,1)$ of values, which are the combinations admitted by the table constraint from $C_3$ in (3). The application of the table constraint to variables $x$ and $y$ is expressed by the argument list in Line 7, so that tuple declarations can be re-used for other variables subject to a similar table constraint.

### 3.2 First-Order Encoding

In addition to an output component extending *sugar* for generating ASP facts, *aspartame* comes along with alternative first-order ASP encodings of solutions for CSP instances. In the following, we sketch a dedicated encoding that, for one, relies on function terms to capture recurrences of similar structures and, for another, lifts the order encoding approach to structural subsum entities.

**Static Extraction of Relevant Values** To begin with, Listing 2 shows (relevant) instances of domain predicates, evaluated upon grounding, for the CSP from Example 1. While derived facts in Line 1 merely provide a projection of the predicate `var`/3 omitting associated domains, the instances of `look`/2 in Line 2 express that all values in the common domain $\{1,2,3\}$ of $x$, $y$, and $z$ shall be considered. In fact, domain predicates extract variable values that can be relevant for the satisfiability of a CSP instance, while discarding the rest. The respective static analysis consists of three stages: (i) isolation of threshold values relevant to linear inequalities; (ii) addition of missing values for variables occurring in alldifferent constraints; (iii) addition of white/black list values for table constraints.

In the first stage, we consider the domains of Integer variables $x$ in terms of corresponding (non-overlapping) intervals $I(x) = \{[l_1,u_1],\ldots,[l_k,u_k]\}$. These are extended to multiplications by Integer constants $a$ according to the following scheme:

$$I(ax) = \begin{cases} \{[a*l_1, a*u_1],\ldots,[a*l_k, a*u_k]\} & \text{if } 0 \leq a \\ \{[a*u_k, a*l_k],\ldots,[a*u_1, a*l_1]\} & \text{if } a < 0 \end{cases}$$

For $4x - 3y + z \leq 0$ from $C_2$ in (2), we get $I(4x) = \{[4,12]\}$, $I(-3y) = \{[-9,-3]\}$, and $I(1z) = \{[1,3]\}$. Such intervals are used to retrieve bounds for (sub)sums:

$$\begin{aligned} \overrightarrow{l}(ax) &= \min\{l \mid [l,u] \in I(ax)\} \\ \overrightarrow{u}(ax) &= \max\{u \mid [l,u] \in I(ax)\} \\ \overrightarrow{l}(a_1x_1 + a_2x_2) &= \overrightarrow{l}(a_1x_1) + \overrightarrow{l}(a_2x_2) \\ \overrightarrow{u}(a_1x_1 + a_2x_2) &= \overrightarrow{u}(a_1x_1) + \overrightarrow{u}(a_2x_2) \end{aligned}$$

Given $\overrightarrow{l}(4x) = 4$, $\overrightarrow{u}(4x) = 12$, $\overrightarrow{l}(-3y) = -9$, $\overrightarrow{u}(-3y) = -3$, $\overrightarrow{l}(1z) = 1$, and $\overrightarrow{u}(1z) = 3$, we derive $\overrightarrow{l}(4x - 3y) = -5$, $\overrightarrow{u}(4x - 3y) = 9$, $\overrightarrow{l}(4x - 3y + z) = -4$, and $\overrightarrow{u}(4x - 3y + z) = 12$.

In view of the comparison with 0 in $4x-3y+z \leq 0$, we can now "push in" relevant thresholds via:

$$\overleftarrow{l}(\sum_{1\leq i\leq n} a_i x_i) = \max\{m, \overrightarrow{l}(\sum_{1\leq i\leq n} a_i x_i)\} \qquad \text{for } \sum_{1\leq i\leq n} a_i x_i \leq m$$
$$\overleftarrow{u}(\sum_{1\leq i\leq n} a_i x_i) = \min\{m, \overrightarrow{u}(\sum_{1\leq i\leq n} a_i x_i)\} \qquad \text{for } \sum_{1\leq i\leq n} a_i x_i \leq m$$
$$\overleftarrow{l}(\sum_{1\leq i\leq n-1} a_i x_i) = \max\{\overleftarrow{l}(\sum_{1\leq i\leq n} a_i x_i) - \overrightarrow{u}(a_n x_n), \overrightarrow{l}(\sum_{1\leq i\leq n-1} a_i x_i)\}$$
$$\overleftarrow{u}(\sum_{1\leq i\leq n-1} a_i x_i) = \min\{\overleftarrow{u}(\sum_{1\leq i\leq n} a_i x_i) - \overrightarrow{l}(a_n x_n), \overrightarrow{u}(\sum_{1\leq i\leq n-1} a_i x_i)\}$$

Such threshold analysis leads to $\overleftarrow{l}(4x-3y+z) = \overleftarrow{u}(4x-3y+z) = 0$, $\overleftarrow{l}(4x-3y) = -3$, $\overleftarrow{u}(4x-3y) = -1$, $\overleftarrow{l}(4x) = 4$, and $\overleftarrow{u}(4x) = 8$, telling us that subsums relevant for checking whether $4x-3y+z \leq 0$ satisfy $-3 \leq 4x-3y \leq -1$ and $4 \leq 4x \leq 8$. Note that maxima (or minima) used to construct $\overleftarrow{l}(\sum_{1\leq i\leq n} a_i x_i)$ (or $\overleftarrow{u}(\sum_{1\leq i\leq n} a_i x_i)$) serve two purposes. For one, they correct infeasible arithmetical thresholds to domain values; e.g., $\overleftarrow{l}(4x-3y) - \overrightarrow{u}(-3y) = -3+3 = 0$ tells us that 0 would be the greatest lower bound to consider for $4x$ (since $4x - 3y + z \leq 0$ were necessarily satisfied when $4x \leq 0$), while the smallest possible value $\overrightarrow{l}(4x) = 4$ exceeds 0. For another, dominating values like $4x = 12$ are discarded, given that $4x - 3y + z \leq 0$ cannot hold when $4x > \overleftarrow{u}(4x-3y) - \overrightarrow{l}(-3y) = -1 + 9 = 8$.

Letting $ub(0) = \{0\}$, the upper bounds for $\sum_{1\leq i\leq n} a_i x_i$ that deserve further consideration are then obtained as follows:

$$ub(\sum_{1\leq i\leq n} a_i x_i) = \{\max\{j + a_n{*}k, \overleftarrow{l}(\sum_{1\leq i\leq n} a_i x_i)\} \mid j \in ub(\sum_{1\leq i\leq n-1} a_i x_i),$$
$$k \in \mathbb{Z}, [l,u] \in I(a_n x_n), l \leq a_n{*}k \leq \min\{u, \overleftarrow{u}(\sum_{1\leq i\leq n} a_i x_i) - j\}\}$$

Starting from the above thresholds, $ub(4x) = \{4, 8\}$, $ub(4x-3y) = \{-3, -2, -1\}$, and $ub(4x-3y+z) = \{0\}$ indicate upper bounds for subsums that are of interest in evaluating $4x - 3y + z \leq 0$. Upper bounds in $ub(\sum_{1\leq i\leq n} a_i x_i)$ can in turn be related to "maximal" pairs of addends:

$$S(\sum_{1\leq i\leq n} a_i x_i) = \{(j, \max\{a_n{*}k \mid [l,u] \in I(a_n x_n), l \leq a_n{*}k \leq \min\{u, ub - j\},$$
$$k \in \mathbb{Z}\}) \mid j \in ub(\sum_{1\leq i\leq n-1} a_i x_i), ub \in ub(\sum_{1\leq i\leq n} a_i x_i), \overrightarrow{l}(a_n x_n) \leq ub - j\}$$

In our example, we get $S(4x) = \{(0,4), (0,8)\}$, $S(4x-3y) = \{(4,-9), (4,-6), (8,-9)\}$, and $S(4x-3y+z) = \{(-3,3), (-2,2), (-1,1)\}$.

Finally, we associate each pair $(j, a_n{*}k) \in S(\sum_{1\leq i\leq n} a_i x_i)$ of addends with the upper bound $s(j, a_n{*}k) = \min\{ub \in ub(\sum_{1\leq i\leq n} a_i x_i) \mid j + a_n{*}k \leq ub\}$, thus obtaining $s(0,4) = 4$, $s(0,8) = 8$, $s(4,-9) = -3$, $s(4,-6) = -2$, $s(8,-9) = -1$, and $s(-3,3) = s(-2,2) = s(-1,1) = 0$.

The described analysis of thresholds for subsums is implemented via deterministic domain predicates in our ASP encoding. Variables' domain values underlying relevant addends are provided by the derived facts in Line 10–12 of Listing 2. Note that value 3 for $x$ as well as 1 for $y$ are ignored here, given that $4x = 12$ and $-3y = -3$ do not admit $4x - 3y + z \leq 0$ to hold. The mapping of relevant addends to their associated

```
1  var(int,x;y;z).
2  look(x;y;z,1;2;3).

4  order(x;y;z,3,2).
5  order(x;y;z,2,1).

7  order(op(add,op(mul,4,x),op(mul,-3,y)),-1,-2).
8  order(op(add,op(mul,4,x),op(mul,-3,y)),-2,-3).

10 look(op(mul,4,x),1;2,1).
11 look(op(mul,-3,y),2;3,-1).
12 look(op(mul,1,z),1;2;3,1).

14 look(op(add,op(mul,4,x),op(mul,-3,y)),4,-6,-2).
15 look(op(add,op(mul,4,x),op(mul,-3,y)),4,-9,-3).
16 look(op(add,op(mul,4,x),op(mul,-3,y)),8,-9,-1).

18 look(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),-1,1,0).
19 look(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),-2,2,0).
20 look(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),-3,3,0).

22 bound(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),12).

24 difind(arg(x,arg(y,arg(z,nil))),x,1).
25 difind(arg(x,arg(y,arg(z,nil))),y,2).
26 difind(arg(x,arg(y,arg(z,nil))),z,3).
27 difmax(arg(x,arg(y,arg(z,nil))),3,1;2;3).
28 difall(arg(x,arg(y,arg(z,nil)))).

30 relind(r,arg(x,arg(y,nil)),x,1).
31 relind(r,arg(x,arg(y,nil)),y,2).
```

**Listing 2.** Domain predicates derived via stratified rules (not shown) from facts in Listing 1.

upper bound can be observed in Line 14–20 for the (sub)sums $4x - 3y$ and $(4x - 3y) + z$. The respective facts describe patterns for mapping assigned domain values to their multiplication results and then to upper bounds for subsums, which are eventually subject to a (non-trivial) comparison in some linear inequality. (Trivial comparisons are performed via the total upper bound for an addition result, as given in Line 22.) Notably, the static threshold analysis is implemented on terms representing the domains of variables, and outcomes are then mapped back to original variables. Thus, linear inequalities over different variables with the same domains are analyzed only once. The final function terms, however, mention the variables whose values are evaluated, where recurring substructures may share a common term with which all relevant threshold values are associated.

Although the analysis of the linear inequality $4x - 3y + z \leq 0$ identifies the values 3 for $x$ and 1 for $y$ as redundant, the presence of $\mathit{alldifferent}(x, y, z)$ leads to their "release" as relevant candidates for $x$ and $y$. Accordingly, all values in the common domain $\{1, 2, 3\}$ of $x$, $y$, and $z$ are put into (decreasing) order, given by the derived facts in Line 4–5. Beyond that, the order among relevant upper bounds in $ub(4x - 3y) = \{-3, -2, -1\}$ is reflected in Line 7–8; this is used to apply the order encoding to structural subsum variables (in addition to the input variables $x$, $y$, and $z$). The residual derived facts in Line 24–31 serve convenience by associating indexes to the arguments of $\mathit{alldifferent}(x, y, z)$ as well as to $x$ and $y$ considered in $(x, y) \in \{(1, 3), (2, 2), (3, 1)\}$. Furthermore, the fact in Line 27 indicates the index 3 of variable $z$ in $\mathit{alldifferent}(x, y, z)$ as the final position at which either of the

```
1  % generate variable assignment
2
3  { less(V,E) : order(V,E,_) } :- var(int,V).
4   :- order(V,E1,E2), less(V,E2), not less(V,E1).
5  value(V,E) :- look(V,E), not less(V,E), less(V,EE) : order(V,EE,E).
6
7  { value(V,true) } :- var(bool,V).
8  value(V,false) :- var(bool,V), not value(V,true).
9
10 % evaluate linear inequalities
11
12 leq(op(mul,F,V),F*E) :- look(op(mul,F,V),E,1), less(V,EE) : order(V,EE,E).
13 leq(op(mul,F,V),F*E) :- look(op(mul,F,V),E,-1), not less(V,E).
14 leq(op(add,S1,S2),E) :- look(op(add,S1,S2),E1,E2,E), leq(S1,E1;;S2,E2).
15 leq(op(add,S1,S2),E) :- order(op(add,S1,S2),E,EE), leq(op(add,S1,S2),EE).
16
17 % evaluate alldifferent expressions
18
19 seen(A,I,E) :- difind(A,V,I), value(V,E).
20 seen(A,I,E) :- difind(A,_,I), seen(A,I-1,E), not difmax(A,I-1,E).
21
22 redo(A) :- difind(A,V,I), seen(A,I-1,E), value(V,E).
23 redo(A) :- difall(A), difmax(A,I,E), not seen(A,I,E).
24
25 % evaluate table expressions
26
27 rela(R,A,T,2)   :- relind(R,A,V,1), tuple(R,T,1,E), value(V,E).
28 rela(R,A,T,I+1) :- relind(R,A,V,I), tuple(R,T,I,E), value(V,E), rela(R,A,T,I).
29
30 rela(R,A,U) :- rela(R,A,_,I+1), rel(R,I,_,U).
31
32 % check constraint clauses
33
34 hold(C) :- constraint(C,V), value(V,true).
35 hold(C) :- constraint(C,op(neg,V)), value(V,false).
36
37 hold(C) :- constraint(C,op(le,S,E)), bound(S,U), leq(S,E) : E < U.
38 hold(C) :- constraint(C,op(neg,op(le,S,E))), bound(S,U), E < U, not leq(S,E).
39
40 hold(C) :- constraint(C,global(alldifferent,A)), not redo(A).
41 hold(C) :- constraint(C,op(neg,global(alldifferent,A))), redo(A).
42
43 hold(C) :- constraint(C,rel(R,A)), not rela(R,A,conflicts),
44            rela(R,A,supports) : rel(R,_,_,supports).
45 hold(C) :- constraint(C,op(neg,rel(R,A))), not rela(R,A,supports),
46            rela(R,A,conflicts) : rel(R,_,_,conflicts).
47
48 constraint(C) :- constraint(C,_).
49  :- constraint(C), not hold(C).
50
51 % display variable assignment
52
53 #hide.
54 #show value/2.
```

**Listing 3.** First-order encoding of solutions for finite linear CSPs.

values 1, 2, or 3 can possibly be assigned, and the fact in Line 28 expresses that all three values in $D(x) \cup D(y) \cup D(z) = \{1, 2, 3\}$ must be assigned in order to satisfy $\mathit{alldifferent}(x, y, z)$.

**Non-deterministic Encoding Part** With the described domain predicates at hand, the encoding part in Listing 3 implements the non-deterministic guessing of a variable as-

signment along with the evaluation of constraint clauses. Following the idea of order encodings in SAT [4, 5], the choice rule in Line 3 permits guessing `less(V,E)` for all but the smallest (relevant) value `E` in the domain of an Integer variable `V`, thus indicating that `V` is assigned to some smaller value than `E`. The consistency among guessed atoms is established by the integrity constraint in Line 4, requiring `less(V,E1)` to hold if `less(V,E2)` is true for the (immediate) predecessor value `E2` of `E1`. The actual value assigned to `V`, given by the greatest `E` for which `less(V,E)` is false, is extracted in Line 5. For Boolean variables, the value `true` can be guessed unconditionally via the choice rule in Line 7, and `false` is derived otherwise via the rule in Line 8.

The dedicated extension of the order encoding idea to subsums of linear inequalities is implemented by means of the rules in Line 12–15 of Listing 3. To this end, upper bounds for singular multiplication results indicated as relevant by instances of `look(op(mul,F,V),E,G)` are directly derived from `less`/2. Thereby, the flag $\texttt{G} = \texttt{F}/|\texttt{F}|$ provides the polarity of the actual coefficient `F`.[8] If `F` is positive, i.e., $\texttt{G} = 1$, the upper bound $\texttt{F}\star\texttt{E}$ is established as soon as `less(V,EE)` holds for the immediate successor value `EE` of `E` (or if `E` is the greatest relevant value in the domain of `V`). On the other hand, if $\texttt{G} = -1$ indicates that `F` is negative, the upper bound $\texttt{F}\star\texttt{E}$ is derived from `not less(V,E)`, which means that the value assigned to `V` is greater than or equal to `E`. Relevant upper bounds `E` for subsums rely on maximal pairs `(E1,E2)` of addends, identified via static threshold analysis and readily provided by instances of `look(op(add,S1,S2),E1,E2,E)`. In fact, the rule in Line 14 derives `leq(op(add,S1,S2),E)`, indicating that $\texttt{S1} + \texttt{S2} \leq \texttt{E}$, from `leq(S1,E1)` and `leq(S2,E2)`. Although an established upper bound inherently implies any greater (relevant) upper bound to hold as well w.r.t. a total variable assignment, ASP (and SAT) solvers are not committed to guessing "input variables" first. Rather, structural variables like the instances of `leq(op(add,S1,S2),E)` may be fixed upon solving, possibly in view of recorded conflict clauses, before a total assignment has been determined. In view of this, the additional rule in Line 15 makes sure that an established upper bound `EE` propagates to its immediate successor `E` (if there is any). For instance, (simplified) ground instances of the rule stemming from $ub(4x - 3y) = \{-3, -2, -1\}$ include the following:

```
leq(op(add,op(mul,4,x),op(mul,-3,y)),-1) :-
    leq(op(add,op(mul,4,x),op(mul,-3,y)),-2).
leq(op(add,op(mul,4,x),op(mul,-3,y)),-2) :-
    leq(op(add,op(mul,4,x),op(mul,-3,y)),-3).
```

Unlike with the domains of Integer variables, we rely on a rule, rather than an integrity constraint, to establish consistency among the bounds for structural subsums. The reason for this is that upper bounds for addends `S1` and `S2`, contributing left and right justifications, may include divergent gaps, so that consistent value orderings for them are, in general, not guaranteed to immediately produce all relevant upper bounds for $\texttt{S1} + \texttt{S2}$. Encoding variants resolving this issue and using integrity constraints like the one in Line 4 are a subject to future investigation.

While linear inequalities can be evaluated by means of boundaries derived more or less directly from instances of `less(V,E)`, the evaluation of alldifferent and table

[8] Coefficients given in facts generated by *aspartame* are distinct from 0.

constraints in Line 19–23 and Line 27–30 of Listing 3 relies on particular instances of `value(V,E)`. The basic idea of checking whether an alldifferent constraint holds is to propagate assigned values along the indexes of participating variables. Then, a recurrence is detected when the value assigned to a variable with index `I` has been marked as already assigned, as determined from `seen(A,I-1,E)` in Line 22. Moreover, whenever `difall(A)` indicates that all domain values for the variables in argument list `A` must be assigned, the rule in Line 23 additionally derives a recurrence from some gap (a value that has not been assigned to the variable at the last possible index). Our full encoding further features so-called "pigeon-hole constraints" (cf. [10, 11]) to check that the smallest or greatest $1, \ldots, n-1$ domain values for an alldifferent constraint with $n$ variables are not populated by more than $i$ variables for $1 \leq i \leq n-1$. Such conditions can again be checked based on instances of `less(V,E)`, and both counter-based (cf. [12]) as well as aggregate-based (cf. [13]) implementations are applicable in view of the native support of aggregates by ASP solvers like *clasp* (cf. [14]). In fact, the usage of rules to express redundant constraints, like the one in Line 23 or those for pigeon-hole constraints, as well as their ASP formulation provide various degrees of freedom, where comprehensive evaluation and configuration methods are subjects to future work.

The strategy for evaluating table constraints is closely related to the one for detecting value recurrences in alldifferent constraints. Based on the indexes of variables in a table constraint, tuples that are (still) admissible are forwarded via the rules in Line 27–28. The inclusion of a full tuple in an assignment is detected by the rule in Line 30, checking whether the arity `I` of a table constraint has been reached for some tuple, where a value `supports` or `conflicts` for `U` additionally indicates whether the included tuple belongs to a white or black list, respectively. Note that this strategy avoids explicit references to variables whose values are responsible for the exclusion of tuples, given that lack of inclusion is detected from incomplete tuple traversals.

Finally, the rules in Line 34–49 explore the values assigned to Boolean variables and the outcomes of evaluating particular kinds of constraints to derive `hold(C)` if and only if some positive or negative literal in `C` is satisfied or unsatisfied, respectively, w.r.t. the variable assignment represented by instances of `value(V,E)`. Without going into details, let us still note that our full encoding also features linear inequalities relying on the comparison operators $\geq$, $=$, and $\neq$, for which additional rules are included to derive `hold(C)`, yet sticking to the principle of upper bound evaluation via `leq`/2. In fact, the general possibility of complemented constraint expressions as well as of disjunctions potentially admits unsatisfied constraint expressions w.r.t. solutions, and our encoding reflects this by separating the evaluation of particular constraint expressions in Line 12–30 from further literal and clause evaluation in Line 34–49.

## 4 The *aspartame* System

The architecture of the *aspartame* system is given in Figure 1. As mentioned, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing CSPs. Hence, it accepts the same input formats, viz. XCSP[9] and *sugar*'s native CSP format[10]. We then implemented an

[9] `http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf`

[10] `http://bach.istc.kobe-u.ac.jp/sugar/package/current/docs/syntax.html`

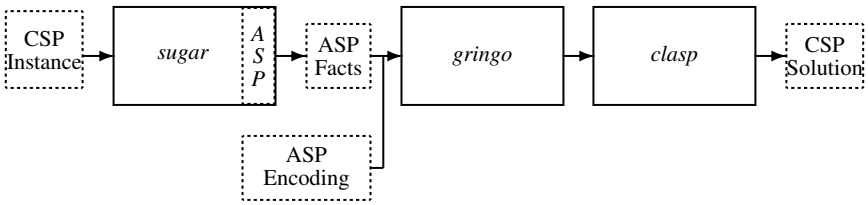



**Fig. 1.** Architecture of *aspartame*.

output hook for *sugar* that provides us with the resulting CSP instance in the fact format described in Section 3.1. These facts are then used for grounding the (full version of the) dedicated ASP encoding in Listing 3 or an alternative one (discussed below). This is done by the ASP grounder *gringo*. In turn, the resulting propositional logic program is passed to the ASP solver *clasp* that returns an assignment, representing a solution to the original CSP instance.

We empirically access the performance of *aspartame* relative to two ASP encodings, the dedicated one described in Section 3.2 as well as a more direct encoding inspired by the original CNF construction of *sugar* [5], and additionally consider the SAT-based reference solver *sugar* (2.0.0). In either case, we use the combined ASP and SAT solver *clasp* (2.1.0), and ASP-based approaches further rely on *gringo* (3.0.5) for grounding ASP encodings on facts generated by *aspartame*. We selected 60 representative CSP instances (that are neither too easy nor too hard), consisting of intensional and global constraints, from the benchmarks of the 2009 CSP Competition[11] for running systematic experiments on a cluster of Linux machines equipped with dual Xeon E5520 quad-core 2.26 GHz processors and 48 GB RAM. To get some first insights into suitable search options, we ran *clasp* with its default (berkmin-like) and the popular "vsids" decision heuristic; while SAT-based preprocessing (cf. [15]) is performed by default on CNF inputs, we optionally enabled it for (ground) ASP instances, leading to four combinations of *clasp* settings for ASP-based approaches and two for SAT-based *sugar*.

Table 1 reports runtime results in seconds, separated into conversion time of *aspartame* from CSP instances to ASP facts (first "convert" column) and of *sugar* from CSP instances to CNF, *gringo* times for grounding ASP encodings relative to facts, and finally columns for the search times of *clasp* with the aforementioned options. Each computational phase was restricted to 600 seconds, and timeouts counted in the last row of Table 1 are taken as 600 seconds in the second last row providing average runtimes. Looking at these summary rows, we observe that our two ASP encodings are solved most effectively when vsids decision heuristic and SAT preprocessing are both enabled; unlike this, neither decision heuristic dominates the other on CNF input. Apparently, *clasp* on CNFs generated by *sugar* still has a significant edge on facts by *aspartame* combined with either ASP encoding. In particular, we observe drastic performance discrepancies on some instance families (especially "fischer" and "queensKnights"), where *clasp* performs stable on CNFs from *sugar* but runs into trouble on corresponding ASP instances. Given that *aspartame* and its ASP encodings are prototypes, such behavior does not disprove the basic approach, but rather motivates future

[11] `http://www.cril.univ-artois.fr/CPAI09`

**Table 1.** Experiments comparing ASP encoding variants and the SAT-based solver *sugar*.

| Benchmark | | *ASP Encoding 1 (dedicated)* | | | | | *ASP Encoding 2 (SAT-inspired)* | | | | | *sugar* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | convert | ground | default | vsids | sat-pre | vsids/ sat-pre | ground | default | vsids | sat-pre | vsids/ sat-pre | convert | default | vsids |
| 1-fullins-5-5 | 2.02 | 1.41 | 13.96 | 11.28 | 5.15 | 3.66 | 0.91 | 10.72 | 12.15 | 6.72 | 7.18 | 1.73 | 2.40 | 2.19 |
| 3-fullins-5-6 | 3.55 | 32.39 | 17.07 | 14.90 | 21.75 | 6.52 | 11.26 | 13.08 | 16.52 | 11.53 | 14.02 | 5.36 | 1.91 | 1.50 |
| 4-fullins-4-7 | 2.20 | 4.45 | 28.35 | 45.59 | 22.02 | 28.38 | 2.29 | 20.03 | 39.92 | 28.60 | 42.91 | 2.80 | 2.19 | 4.81 |
| abb313GPIA-7 | 5.02 | 46.97 | 0.67 | 1.85 | 0.71 | 2.01 | 20.33 | 31.75 | 2.27 | 41.61 | 1.63 | 7.18 | 6.05 | 0.05 |
| abb313GPIA-8 | 5.14 | 51.23 | 411.17 | 460.56 | 521.11 | TO | 21.97 | 173.06 | 451.02 | 433.31 | 180.33 | 7.23 | 131.41 | 70.57 |
| abb313GPIA-9 | 5.96 | 52.79 | TO | TO | TO | TO | 24.68 | 243.72 | TO | 289.54 | TO | 8.49 | 0.59 | 5.73 |
| bibd-8-98-49-4-21_glb | 3.68 | 28.30 | TO | 20.70 | 4.47 | 1.42 | 28.59 | 10.52 | 18.11 | 5.36 | 8.41 | 11.28 | 1.65 | 1.49 |
| bibd-10-120-36-3-8_glb | 5.20 | 69.94 | 46.68 | 11.62 | 8.21 | 1.13 | 76.65 | 15.84 | 4.55 | 2.47 | 7.93 | 13.63 | 7.28 | 0.85 |
| bibd-25-25-9-9-3_glb | 7.34 | 84.68 | TO | 434.23 | 3.50 | 1.13 | 17.53 | 235.11 | 177.27 | TO | 59.41 | 8.90 | 28.86 | 41.12 |
| bibd-31-31-6-6-1_glb | 9.74 | 254.13 | 159.32 | 3.34 | 156.71 | 14.64 | 39.84 | 12.24 | 0.18 | 0.17 | 0.11 | 17.39 | 77.95 | 0.20 |
| C2-3-15 | 0.88 | 3.30 | 0.21 | 0.13 | 0.15 | 0.08 | 9.74 | 6.03 | 4.88 | 6.63 | 8.24 | 1.64 | 7.37 | 1.05 |
| C4-1-61 | 1.44 | 24.44 | 4.66 | 2.86 | 4.90 | 2.80 | 148.33 | TO | TO | TO | TO | 3.40 | 3.06 | 9.87 |
| C4-2-61 | 2.96 | 26.04 | 6.26 | 1.98 | 6.16 | 2.04 | 152.33 | 222.52 | 69.47 | 227.54 | 165.80 | 3.22 | 7.05 | 3.21 |
| C5-3-91 | 2.73 | 71.55 | 31.42 | 10.32 | 31.87 | 11.91 | TO | TO | TO | TO | TO | 7.89 | 261.39 | TO |
| chnl-10-11 | 0.37 | 0.18 | 0.64 | 1.20 | 4.36 | 11.44 | 0.14 | 1.35 | 2.65 | 12.40 | 13.75 | 0.92 | 19.58 | 55.26 |
| chnl-10-15 | 0.70 | 0.17 | 9.15 | 3.08 | 6.95 | 10.61 | 0.19 | 4.04 | 20.14 | 10.82 | 18.01 | 0.51 | 32.07 | 24.68 |
| chnl-10-20 | 0.56 | 0.24 | 19.14 | 6.91 | 6.51 | 5.22 | 0.33 | 41.96 | 38.34 | 6.81 | 11.98 | 1.11 | 7.77 | 6.30 |
| chnl10-15-pb-cnf-cr | 0.42 | 0.42 | 9.28 | 3.05 | 6.93 | 10.74 | 0.20 | 4.00 | 20.05 | 10.90 | 16.43 | 1.12 | 31.85 | 24.87 |
| costasArray-14 | 0.46 | 0.82 | 3.54 | 2.93 | 3.65 | 4.87 | 0.62 | 18.63 | 18.19 | 24.79 | 6.52 | 2.03 | 0.25 | 0.07 |
| costasArray-15 | 0.52 | 0.94 | 17.96 | 60.17 | 75.50 | 17.95 | 0.78 | 62.10 | 136.40 | 4.96 | 8.83 | 1.59 | 18.41 | 8.16 |
| costasArray-16 | 0.52 | 1.20 | 15.13 | 30.18 | 69.69 | 0.99 | 1.03 | 130.75 | 203.94 | 245.02 | 36.31 | 1.71 | 32.65 | 33.25 |
| costasArray-17 | 0.56 | 1.50 | TO | TO | TO | 232.64 | 1.28 | TO | TO | TO | TO | 2.08 | 148.55 | 553.54 |
| fischer-1-2-fair | 1.44 | 304.21 | 11.21 | 92.81 | 11.42 | 18.79 | 28.50 | 23.81 | 7.81 | TO | TO | 3.42 | 0.97 | 0.02 |
| fischer-2-3-fair | 4.18 | 348.84 | 0.85 | 21.11 | 0.79 | 6.50 | 68.42 | 23.19 | 3.69 | 23.71 | 3.54 | 7.19 | 0.13 | 0.04 |
| fischer-3-8-fair | 30.18 | 515.69 | TO | 458.58 | TO | 60.32 | 376.33 | TO | TO | TO | TO | 35.16 | 22.05 | 37.27 |
| fischer-4-6-fair | 35.12 | 535.93 | 282.84 | 242.05 | 386.37 | 274.27 | 384.19 | TO | TO | TO | TO | 39.15 | 17.16 | 8.79 |
| fischer-6-1-fair | 4.40 | 457.55 | 0.91 | 12.16 | 0.90 | 14.68 | 150.78 | TO | 10.29 | TO | 5.46 | 13.20 | 1.79 | 2.69 |
| magicSquare-6_glb | 0.29 | 2.72 | 46.20 | 1.23 | 4.27 | 1.49 | 4.20 | 9.50 | 2.42 | 18.39 | 1.77 | 2.26 | 0.79 | 0.46 |
| magicSquare-7_glb | 0.28 | 8.04 | 7.54 | 78.17 | 1.15 | 30.18 | 15.63 | 51.08 | 9.59 | 51.00 | 7.41 | 3.00 | 4.10 | 2.20 |
| magicSquare-8_glb | 0.42 | 21.75 | TO | 318.69 | 477.11 | 47.22 | 53.61 | 331.09 | 19.14 | 124.79 | 36.60 | 6.29 | 5.50 | 3.72 |
| mps-mzzv42z | 4.84 | 166.01 | 2.74 | 1.54 | 2.61 | 1.02 | 395.20 | 8.56 | 5.44 | 8.75 | 5.20 | 24.84 | 5.44 | 3.28 |
| mps-p2756 | 1.74 | 278.02 | 4.98 | 11.89 | 5.04 | 5.29 | TO | TO | TO | TO | TO | 161.39 | 1.32 | 1.20 |
| mps-red-air06 | 11.59 | 272.57 | 139.86 | 381.61 | 34.78 | 25.27 | TO | TO | TO | TO | TO | 27.73 | 0.63 | TO |
| mps-red-fiber | 1.28 | 92.78 | 5.07 | 3.78 | 5.34 | 5.39 | 388.43 | 33.55 | TO | 33.43 | TO | 38.74 | 2.05 | 6.76 |
| queensKnights-50-5-add | 1.81 | 16.86 | 18.62 | 24.71 | 44.44 | 18.79 | 28.12 | 558.69 | 85.48 | 537.34 | 44.58 | 3.25 | 12.15 | 0.66 |
| queensKnights-50-5-mul | 3.50 | 17.31 | 40.78 | 48.54 | 38.57 | 20.64 | 30.33 | 221.08 | 67.39 | TO | 50.76 | 3.29 | 2.10 | 0.63 |
| queensKnights-80-5-mul | 2.47 | 77.52 | 181.70 | 354.74 | 172.33 | 383.73 | 126.32 | TO | 426.53 | TO | 414.97 | 5.81 | 7.61 | 2.51 |
| queensKnights-100-5-add | 3.49 | 163.50 | TO | TO | TO | TO | 243.06 | TO | TO | TO | TO | 6.92 | 18.45 | 4.39 |
| ramsey-16-3 | 1.07 | 0.51 | 1.17 | 1.16 | 0.18 | 0.10 | 0.39 | 6.74 | 0.53 | 0.78 | 1.04 | 2.19 | 1.99 | 112.63 |
| ramsey-30-4 | 2.13 | 5.91 | 150.64 | 180.79 | 28.56 | 26.47 | 2.75 | 198.83 | 98.97 | 82.69 | 32.39 | 2.35 | 8.09 | 8.67 |
| ramsey-33-4 | 2.53 | 7.58 | 405.05 | 279.25 | 93.36 | 66.78 | 4.05 | TO | 193.49 | 173.67 | 109.45 | 3.13 | 32.02 | 39.82 |
| ramsey-34-4 | 2.34 | 8.46 | TO | TO | 366.41 | 109.45 | 4.14 | TO | TO | TO | 317.08 | 2.78 | 67.41 | 38.71 |
| ruler-34-9-a4 | 1.15 | 13.01 | 21.70 | 29.62 | 19.05 | 19.16 | 3.08 | 35.29 | 39.71 | 38.59 | 39.19 | 4.30 | 41.98 | 41.16 |
| ruler-44-10-a4 | 1.56 | 37.60 | 367.84 | 233.65 | 302.54 | 280.79 | 8.39 | 567.20 | 542.78 | TO | 483.85 | 9.91 | 405.96 | 446.66 |
| ruler-44-9-a4 | 1.04 | 23.41 | 182.48 | 124.14 | 172.79 | 118.37 | 6.18 | 351.14 | 13.41 | 102.36 | 34.25 | 6.61 | TO | 352.28 |
| ruler-55-10-a3 | 1.08 | 5.89 | TO | 500.84 | TO | TO | 3.19 | TO | TO | TO | TO | 1.73 | 43.23 | 70.22 |
| super-jobShop-e0ddr1-8 | 1.21 | 6.42 | 555.34 | 19.21 | 14.32 | 3.67 | 3.66 | 6.10 | 0.11 | 6.92 | 1.05 | 1.16 | 1.21 | 0.49 |
| super-jobShop-e0ddr2-1 | 1.08 | 7.59 | TO | 32.94 | 8.38 | 12.14 | 3.81 | 16.08 | 0.84 | 2.56 | 3.87 | 1.51 | 5.41 | 0.76 |
| super-jobShop-enddr2-3 | 1.00 | 8.60 | TO | 116.06 | 76.55 | 12.93 | 3.95 | 13.21 | 2.59 | 5.30 | 6.33 | 2.15 | 1.19 | 0.72 |
| super-os-taillard-7-4 | 1.02 | 35.84 | TO | 493.02 | TO | 454.02 | 15.35 | 36.02 | 36.35 | 35.06 | 32.14 | 2.08 | 1.14 | 0.89 |
| super-os-taillard-7-6 | 1.01 | 35.51 | 115.70 | TO | 112.53 | TO | 14.49 | 169.99 | 30.25 | 144.26 | 25.89 | 2.14 | 4.20 | 0.76 |
| super-os-taillard-7-7 | 0.84 | 32.73 | 155.31 | 270.49 | 132.71 | 311.67 | 13.53 | 23.55 | 17.77 | 25.74 | 15.78 | 1.95 | 0.80 | 0.78 |
| super-os-taillard-7-8 | 0.95 | 33.23 | 431.15 | 279.09 | 475.51 | 328.50 | 14.46 | 46.95 | 25.64 | 40.83 | 24.06 | 2.01 | 1.19 | 0.96 |
| zeroin-i-1-10 | 1.97 | 2.41 | 20.92 | 27.16 | 14.97 | 16.18 | 1.74 | 22.09 | 28.81 | 26.62 | 18.56 | 2.17 | 3.48 | 2.57 |
| zeroin-i-3-10 | 1.76 | 1.92 | 7.24 | 33.14 | 11.43 | 16.93 | 1.52 | 13.45 | 29.43 | 19.72 | 31.67 | 2.22 | 4.57 | 4.11 |
| ii-32c4 | 3.73 | 21.92 | 0.24 | 0.02 | 0.03 | 0.03 | 56.69 | 0.02 | 0.01 | 0.03 | 0.04 | 21.67 | 8.77 | 0.76 |
| ii-32d3 | 2.80 | 11.87 | 4.58 | 13.70 | 0.39 | 0.26 | 25.12 | 3.54 | 0.31 | 0.54 | 1.35 | 12.33 | 71.17 | 0.20 |
| p2756 | 2.06 | 269.82 | 3.87 | 12.55 | 3.99 | 6.13 | TO | TO | TO | TO | TO | 162.72 | 4.26 | 2.41 |
| ooo-burch-dill-3-accl-ucl | 2.46 | 14.06 | 13.80 | 18.60 | 6.13 | 5.95 | 5.82 | 30.05 | 24.18 | 12.73 | 4.61 | 2.77 | 1.28 | 1.00 |
| ooo-tag14 | 6.93 | 191.71 | 145.03 | 144.09 | 158.56 | 158.99 | 44.99 | 109.54 | 310.78 | 20.82 | 25.11 | 10.67 | 6.54 | 6.07 |
| Average Time | 3.51 | 80.21 | 188.67 | 149.80 | 129.06 | 103.87 | 91.49 | 209.46 | 184.50 | 218.43 | 169.76 | 12.56 | 37.47 | 54.27 |
| Timeouts | 0 | 0 | 12 | 5 | 6 | 5 | 4 | 14 | 13 | 17 | 13 | 0 | 1 | 2 |

investigations of the reasons for performance discrepancies. For one, we conjecture that normalizations of global constraints that are not yet supported by *aspartame* are primarily responsible for large instance sizes and long search times on some instance families. For another, we suppose that both of our ASP encodings are still quite naive compared to years of expertise manifested in *sugar*'s CNF construction. However, the observation that our dedicated ASP encoding has on edge the SAT-inspired one and yields significant performance improvements on some instance families ("C2-3-15"–"C5-3-91" and "mps") clearly encourages further investigations into ASP encodings of CSP instances.

## 5 Related Work

Unlike approaches to constraint answer set solving, e.g., [10, 16–18], which aim at integrating CSP and ASP solving (engines), the focus of *aspartame* lies on pure CSP solving. In fact, *aspartame*'s approach can be regarded as a first-order alternative to SAT-based systems like *sugar* [5], where the performance of the underlying SAT solver is crucial. However, it is now becoming recognized that the SAT encoding to be used also plays an important role [19]. There have been several proposals of encoding constraints to SAT: direct encoding [20, 21], support encoding [22, 23], log encoding [24, 25], log support encoding [26], regular encoding [27], order encoding [4, 5], and compact order encoding [28].

The order encoding, where Boolean variables represent whether $x \leq i$ holds for variables $x$ and values $i$, showed good performance for a wide range of CSPs [4, 11, 27, 29–34]. Especially, the SAT-based constraint solver *sugar* became a winner in global constraint categories at the 2008 and 2009 CSP solver competitions [35]. Moreover, the SAT-based CSP solver BEE [36] and the CLP system B-Prolog [37] utilize the order encoding. In fact, the order encoding provides a compact translation of arithmetic constraints, while also maintaining bounds consistency by unit propagation. Interestingly, it has been shown that the order encoding is the only existing SAT encoding that can reduce tractable CSP to tractable SAT [38].

## 6 Conclusion

We presented an alternative approach to solving finite linear CSPs based on ASP. The resulting system *aspartame* relies on high-level ASP encodings and delegates both the grounding and solving tasks to general-purpose ASP systems. We have contrasted *aspartame* with its SAT-based ancestor *sugar*, which delegates only the solving task to off-the-shelf SAT solvers, while using dedicated algorithms for constraint preprocessing. Although *aspartame* does not fully match the performance of *sugar* from a global perspective, the picture is fragmented and leaves room for further improvements. This is to say that different performances are observed on distinct classes of CSPs, comprising different types of constraints. Thus, it is an interesting topic of future research to devise more appropriate ASP encodings for such settings. Despite all this, *aspartame* demonstrates that ASP's general-purpose technology allows to compete with state-of-the-art constraint solving techniques, not to mention that *aspartame*'s intelligence is driven by an ASP encoding of less than 100 code lines (for non-deterministic predicates subject

to search). In fact, the high-level approach of ASP facilitates extensions and variations of first-order encodings for dealing with particular types of constraints. In the future, we thus aim at more exhaustive investigations of encoding variants, e.g., regarding alldifferent constraints, as well as support for additional kinds of global constraints.

*Acknowledgments* This work was partially funded by the Japan Society for the Promotion of Science (JSPS) under grant KAKENHI 24300007 as well as the German Science Foundation (DFG) under grant SCHA 550/8-3 and SCHA 550/9-1. We are grateful to the anonymous reviewers for many helpful comments.

## References

1. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
2. Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier Science (2006)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS Press (2009)
4. Crawford, J., Baker, A.: Experimental results on the application of satisfiability algorithms to scheduling problems. In Hayes-Roth, B., and Korf, R., eds.: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94), AAAI Press (1994) 1092–1097
5. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints **14**(2) (2009) 254–272
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
8. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.[12]
9. Beldiceanu, N., Simonis, H.: A constraint seeker: Finding and ranking global constraints from examples. In Lee, J., ed.: Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11), Springer-Verlag (2011) 12–26
10. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. Theory and Practice of Logic Programming **10**(4-6) (2010) 465–480
11. Metodi, A., Codish, M., Stuckey, P.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. Journal of Artificial Intelligence Research **46** (2013) 303–341
12. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In van Beek, P., ed.: Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05), Springer-Verlag (2005) 827–831
13. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
14. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. [39] 250–264
15. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05), Springer-Verlag (2005) 61–75

[12] `http://potassco.sourceforge.net`

16. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. [39] 235–249
17. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In Faber, W., Lee, J., eds.: Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09), (2009) 16–30
18. Ostrowski, M., Schaub, T.: ASP modulo CSP: The clingcon system. Theory and Practice of Logic Programming **12**(4-5) (2012) 485–503
19. Prestwich, S.: CNF encodings. [3] 75–97
20. de Kleer, J.: A comparison of ATMS and CSP techniques. In Sridharan, N., ed.: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89), Morgan Kaufmann Publishers (1989) 290–296
21. Walsh, T.: SAT v CSP. In Dechter, R., ed.: Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'00), Springer-Verlag (2000) 441–456
22. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence **45**(3) (1990) 275–286
23. Gent, I.: Arc consistency in SAT. In van Harmelen, F., ed.: Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI'02), IOS Press (2002) 121–125
24. Iwama, K., Miyazaki, S.: SAT-variable complexity of hard combinatorial problems. In Pehrson, B., Simon, I., eds.: Proceedings of the Thirteenth IFIP World Computer Congress (WCC'94), North-Holland (1994) 253–258
25. Van Gelder, A.: Another look at graph coloring via propositional satisfiability. Discrete Applied Mathematics **156**(2) (2008) 230–243
26. Gavanelli, M.: The log-support encoding of CSP into SAT. In Bessiere, C., ed.: Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07), Springer-Verlag (2007) 815–822
27. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables into problems with Boolean variables. In Hoos, H., Mitchell, D., eds.: Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04), Springer-Verlag (2004) 1–15
28. Tanjo, T., Tamura, N., Banbara, M.: Azucar: A SAT-based CSP solver using compact order encoding (tool presentation). In Cimatti, A., Sebastiani, R., eds.: Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12), Springer-Verlag (2012) 456–462
29. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of Boolean cardinality constraints. In Rossi, F., ed.: Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), Springer-Verlag (2003) 108–122
30. Gent, I., Nightingale, P.: A new encoding of AllDifferent into SAT. In Frisch, A., Miguel, I., eds.: Proceedings of the Third International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (ModRef'04), (2004) 95–110
31. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154**(16) (2006) 2291–2306
32. Soh, T., Inoue, K., Tamura, N., Banbara, M., Nabeshima, H.: A SAT-based method for solving the two-dimensional strip packing problem. Fundamenta Informaticae **102**(3-4) (2010) 467–487
33. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3) (2009) 357–391
34. Banbara, M., Matsunaka, H., Tamura, N., Inoue, K.: Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In Fermüller, C., Voronkov, A., eds.: Proceedings of the Seventeenth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10), Springer-Verlag (2010) 112–126

35. Lecoutre, C., Roussel, O., van Dongen, M.: Promoting robust black-box solvers through competitions. Constraints **15**(3) (2010) 317–326
36. Metodi, A., Codish, M.: Compiling finite domain constraints to SAT with BEE. Theory and Practice of Logic Programming **12**(4-5) (2012) 465–483
37. Zhou, N.: The SAT compiler in B-prolog. The Association for Logic Programming Newsletter, March 2013 (2013)[13]
38. Petke, J., Jeavons, P.: The order encoding: From tractable CSP to tractable SAT. In Sakallah, K., Simon, L., eds.: Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11), Springer-Verlag (2011) 371–372
39. Hill, P., Warren, D., eds.: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), Springer-Verlag (2009)

[13] `http://www.cs.nmsu.edu/ALP/2013/03/the-sat-compiler-in-b-prolog`

# A Functional View of Strong Negation in Answer Set Programming

Michael Bartholomew and Joohyung Lee

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, USA

**Abstract.** The distinction between strong negation and default negation has been useful in answer set programming. We present an alternative account of strong negation, which lets us view strong negation in terms of the functional stable model semantics by Bartholomew and Lee. More specifically, we show that, under complete interpretations, minimizing both positive and negative literals in the traditional answer set semantics is essentially the same as ensuring the uniqueness of Boolean function values under the functional stable model semantics. The same account lets us view Lifschitz's two-valued logic programs as a special case of the functional stable model semantics. In addition, we show how non-Boolean intensional functions can be eliminated in favor of Boolean intensional functions, and furthermore can be represented using strong negation, which provides a way to compute the functional stable model semantics using existing ASP solvers. We also note that similar results hold with the functional stable model semantics by Cabalar.

## 1 Introduction

The distinction between default negation and strong negation has been useful in answer set programming. In particular, it yields an elegant solution to the frame problem. The fact that block $b$ stays at the same location $l$ by inertia can be described by the rule

$$\mathit{On}(b,l,t+1) \ \leftarrow \ \mathit{On}(b,l,t),\ \mathit{not}\ \sim\mathit{On}(b,l,t+1) \tag{1}$$

along with the rule that describes the uniqueness of location values [Lifschitz, 2002],

$$\sim\mathit{On}(b,l_1,t) \ \leftarrow \ \mathit{On}(b,l,t),\ l \neq l_1\ . \tag{2}$$

Here '$\sim$' is the symbol for strong negation that represents explicit falsity while '*not*' is the symbol for default negation (negation as failure). Rule (1) asserts that without explicit evidence to the contrary, block $b$ remains at location $l$. If we are given explicit conflicting information about the location of $b$ at time $t+1$ then this conclusion will be defeated by rule (2), which asserts the uniqueness of location values.

An alternative representation of inertia, which uses choice rules instead of strong negation, was recently presented by Bartholomew and Lee [2012]. Instead of rule (1), they use the choice rule

$$\{\mathit{On}(b,l,t+1)\} \ \leftarrow \ \mathit{On}(b,l,t)\ , \tag{3}$$

which states that "if $b$ is at $l$ at time $t$, then decide arbitrarily whether to assert that $b$ is at $l$ at time $t+1$." Instead of rule (2), they write weaker rules for describing the functional

property of *On*:

$$\leftarrow \ \{\mathit{On}(b,l,t) : \mathit{Location}(l)\}0 \qquad \text{(existence of location)} \tag{4}$$

$$\leftarrow \ 2\{\mathit{On}(b,l,t) : \mathit{Location}(l)\} \qquad \text{(uniqueness of location),} \tag{5}$$

which can be also combined into one rule: $\leftarrow \ \mathit{not}\ 1\{\mathit{On}(b,l,t) : \mathit{Location}(l)\}1$ . In the absence of additional information about the location of block $b$ at time $t{+}1$, asserting $\mathit{On}(b,l,t{+}1)$ is the only option, in view of the existence of location constraint (4). But if we are given conflicting information about the location of $b$ at time $t{+}1$ then not asserting $\mathit{On}(b,l,t{+}1)$ is the only option, in view of the uniqueness of location constraint (5).

Rules (3), (4), and (5) together can be more succinctly represented in the language of [Bartholomew and Lee, 2012] by means of intensional functions. That is, the three rules can be replaced by one rule

$$\{\mathit{Loc}(b,t{+}1) = l\} \ \leftarrow \ \mathit{Loc}(b,t) = l \ , \tag{6}$$

where *Loc* is an intensional function constant (the rule reads, "if block $b$ is at location $l$ at time $t$, by default, the block is at $l$ at time $t{+}1$"). In fact, Corollary 2 of [Bartholomew and Lee, 2012] tells us how to eliminate intensional functions in favor of intensional predicates, justifying the equivalence between (6) and the set of rules (3), (4), and (5). The translation allows us to compute the language of [Bartholomew and Lee, 2012] using existing ASP solvers, such as SMODELS and GRINGO. However, DLV cannot be used because it does not accept choice rules. On the other hand, all these solvers accept rules (1) and (2), which contain strong negation.

The two representations of inertia involving intensional predicate *On* do not result in the same answer sets. In the first representation, which uses strong negation, each answer set contains only one atom of the form $\mathit{On}(b,l,t)$ for each block $b$ and each time $t$; for all other locations $l'$, negative literals $\sim\mathit{On}(b,l',t)$ belong to the answer set. On the other hand, such negative literals do not occur in the answer sets of a program that follows the second representation, which yields fewer ground atoms. This difference can be well explained by the difference between the symmetric and the asymmetric views of predicates that Lifschitz described in his message to Texas Action Group, titled "Choice Rules and the Belief-Based View of ASP": [1]

> The way I see it, in ASP programs we use predicates of two kinds, let's call them "symmetric" and "asymmetric." The fact that an object $a$ does not have a property $p$ is reflected by the presence of $\sim p(a)$ in the answer set if $p$ is "symmetric," and by the absence of $p(a)$ if $p$ is "asymmetric." In the second case, the strong negation of $p$ is not used in the program at all.

According to these terminologies, predicate *On* is symmetric in the first representation, and asymmetric in the second representation.

This paper presents several technical results that help us understand the relationship between these two views. In this regard, it helps us to understand strong negation as a way of expressing intensional Boolean functions.

[1] http://www.cs.utexas.edu/users/vl/tag/choice_discussion

– Our first result provides an alternative account of strong negation in terms of Boolean intensional functions. For instance, (1) can be identified with

$$On(b,l,t+1)=\text{TRUE} \leftarrow On(b,l,t)=\text{TRUE} \wedge \neg(On(b,l,t+1)=\text{FALSE}) \ ,$$

and (2) can be identified with

$$On(b,l_1,t)=\text{FALSE} \leftarrow On(b,l,t)=\text{TRUE} \wedge l \neq l_1 \ .$$

Under complete interpretations, we show that minimizing both positive and negative literals in the traditional answer set semantics is essentially the same as ensuring the uniqueness of Boolean function values under the functional stable model semantics. In this sense, strong negation can be viewed as a mere disguise of Boolean functions.[2]
– We show how non-Boolean intensional functions can be eliminated in favor of Boolean functions. Combined with the result in the first bullet, this tells us a new way of turning the language of [Bartholomew and Lee, 2012] into traditional answer set programs with strong negation, so that system DLV, as well as SMODELS and GRINGO, can be used for computing the language of [Bartholomew and Lee, 2012]. As an example, it tells us how to turn (6) into the set of rules (1) and (2).
– Lifschitz [2012] recently proposed "two-valued logic programs," which modifies the traditional stable model semantics to represent complete information without distinguishing between strong negation and default negation. Using our result that views strong negation in terms of Boolean functions, we show that two-valued logic programs are in fact a special case of the functional stable model semantics in which every function is Boolean.

While the main results are stated for the language of [Bartholomew and Lee, 2012], similar results hold with the language of [Cabalar, 2011] based on the relationship between the two languages studied in [Bartholomew and Lee, 2013]. Furthermore, we note that the complete interpretation assumption in the first bullet can be dropped if we instead refer to the language of [Cabalar, 2011], at the price of introducing partial interpretations.

The paper is organized as follows. In Section 2 we review the two versions of the stable model semantics, one that allows strong negation, but is limited to express intensional predicates only, and the other that allows both intensional predicates and intensional functions. As a special case of the latter we also present multi-valued propositional formulas under the stable model semantics. Section 3 shows how strong negation can be viewed in terms of Boolean functions. Section 4 shows how non-Boolean functions can be eliminated in favor of Boolean functions. Section 5 shows how Lifschitz's two-valued logic programs can be viewed as a special case of the functional stable model semantics. Section 6 shows how strong negation can be represented in the language of [Cabalar, 2011].

## 2 Preliminaries

### 2.1 Review: First-Order Stable Model Semantics and Strong Negation

This review follows [Ferraris *et al.*, 2011]. A *signature* is defined as in first-order logic, consisting of *function constants* and *predicate constants*. Function constants of arity 0

[2] It is also well-known that strong negation can be also viewed in terms of auxiliary predicate constants [Gelfond and Lifschitz, 1991].

are also called *object constants*. We assume the following set of primitive propositional connectives and quantifiers: $\bot$ (falsity), $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$. The syntax of a formula is defined as in first-order logic. We understand $\neg F$ as an abbreviation of $F \rightarrow \bot$.

The stable models of a sentence $F$ relative to a list of predicates $\mathbf{p} = (p_1, \ldots, p_n)$ are defined via the *stable model operator with the intensional predicates* $\mathbf{p}$, denoted by $\mathrm{SM}[F; \mathbf{p}]$. Let $\mathbf{u}$ be a list of distinct predicate variables $u_1, \ldots, u_n$ of the same length as $\mathbf{p}$. By $\mathbf{u} = \mathbf{p}$ we denote the conjunction of the formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \leftrightarrow p_i(\mathbf{x}))$, where $\mathbf{x}$ is a list of distinct object variables of the same length as the arity of $p_i$, for all $i = 1, \ldots, n$. By $\mathbf{u} \leq \mathbf{p}$ we denote the conjunction of the formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \rightarrow p_i(\mathbf{x}))$ for all $i = 1, \ldots, n$, and $\mathbf{u} < \mathbf{p}$ stands for $(\mathbf{u} \leq \mathbf{p}) \wedge \neg(\mathbf{u} = \mathbf{p})$. For any first-order sentence $F$, expression $\mathrm{SM}[F; \mathbf{p}]$ stands for the second-order sentence

$$F \wedge \neg \exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})),$$

where $F^*(\mathbf{u})$ is defined recursively:

- $p_i(\mathbf{t})^* = u_i(\mathbf{t})$ for any list $\mathbf{t}$ of terms;
- $F^* = F$ for any atomic formula $F$ (including $\bot$ and equality) that does not contain members of $\mathbf{p}$;
- $(F \wedge G)^* = F^* \wedge G^*$; $\quad (F \vee G)^* = F^* \vee G^*$;
- $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$;
- $(\forall x F)^* = \forall x F^*$; $\quad (\exists x F)^* = \exists x F^*$.

A model of a sentence $F$ (in the sense of first-order logic) is called $\mathbf{p}$-*stable* if it satisfies $\mathrm{SM}[F; \mathbf{p}]$.

The traditional stable models of a logic program $\Pi$ are identical to the Herbrand stable models of the *FOL-representation* of $\Pi$ (i.e., the conjunction of the universal closures of implications corresponding to the rules).

Ferraris *et al.* [2011] incorporate strong negation into the stable model semantics by distinguishing between intensional predicates of two kinds, *positive* and *negative*. Each negative intensional predicate has the form $\sim p$, where $p$ is a positive intensional predicate and '$\sim$' is a symbol for strong negation. In this sense, syntactically $\sim$ is not a logical connective, as it can appear only as a part of a predicate constant. An interpretation of the underlying signature is *coherent* if it satisfies the formula $\neg \exists \mathbf{x}(p(\mathbf{x}) \wedge \sim p(\mathbf{x}))$, where $\mathbf{x}$ is a list of distinct object variables, for each negative predicate $\sim p$. We consider coherent interpretations only.

**Example 1** *The following is a representation of the Blocks World in the syntax of logic programs:*

$$\begin{array}{rcll}
\bot & \leftarrow & \mathit{On}(b_1, b, t), \mathit{On}(b_2, b, t) & (b_1 \neq b_2) \\
\mathit{On}(b, l, t+1) & \leftarrow & \mathit{Move}(b, l, t) & \\
\bot & \leftarrow & \mathit{Move}(b, l, t), \mathit{On}(b_1, b, t) & \\
\bot & \leftarrow & \mathit{Move}(b, b_1, t), \mathit{Move}(b_1, l, t) & \\
\mathit{On}(b, l, 0) & \leftarrow & \mathit{not}\ \sim \mathit{On}(b, l, 0) & \\
\sim \mathit{On}(b, l, 0) & \leftarrow & \mathit{not}\ \mathit{On}(b, l, 0) & \\
\mathit{Move}(b, l, t) & \leftarrow & \mathit{not}\ \sim \mathit{Move}(b, l, t) & \\
\sim \mathit{Move}(b, l, t) & \leftarrow & \mathit{not}\ \mathit{Move}(b, l, t) & \\
\mathit{On}(b, l, t+1) & \leftarrow & \mathit{On}(b, l, t), \mathit{not}\ \sim \mathit{On}(b, l, t+1) & \\
\sim \mathit{On}(b, l, t) & \leftarrow & \mathit{On}(b, l_1, t) & (l \neq l_1)\ .
\end{array} \tag{7}$$

*Here On and Move are intensional predicate constants, $b$, $b_1$, $b_2$ are variables ranging over the blocks, $l$, $l_1$ are variables ranging over the locations (blocks and the table), and $t$ is a variable ranging over the timepoints. The first rule asserts that at most one block can be on another block. The next three rules describe the effect and preconditions of action Move. The next four rules describe that fluent On is initially exogenous, and action Move is exogenous at each time. The next rule describes inertia, and the last rule asserts that a block can be at most at one location.*

### 2.2 Review: The Functional Stable Model Semantics

The functional stable model semantics is defined by modifying the semantics in the previous section to allow "intensional" functions [Bartholomew and Lee, 2012]. For predicate symbols (constants or variables) $u$ and $c$, we define $u \leq c$ as $\forall \mathbf{x}(u(\mathbf{x}) \rightarrow c(\mathbf{x}))$. We define $u = c$ as $\forall \mathbf{x}(u(\mathbf{x}) \leftrightarrow c(\mathbf{x}))$ if $u$ and $c$ are predicate symbols, and $\forall \mathbf{x}(u(\mathbf{x}) = c(\mathbf{x}))$ if they are function symbols.

Let $\mathbf{c}$ be a list of distinct predicate and function constants and let $\widehat{\mathbf{c}}$ be a list of distinct predicate and function variables corresponding to $\mathbf{c}$. We call members of $\mathbf{c}$ *intensional* constants. By $\mathbf{c}^{pred}$ we mean the list of the predicate constants in $\mathbf{c}$, and by $\widehat{\mathbf{c}}^{pred}$ the list of the corresponding predicate variables in $\widehat{\mathbf{c}}$. We define $\widehat{\mathbf{c}} < \mathbf{c}$ as $(\widehat{\mathbf{c}}^{pred} \leq \mathbf{c}^{pred}) \wedge \neg(\widehat{\mathbf{c}} = \mathbf{c})$ and SM$[F;\ \mathbf{c}]$ as

$$F \wedge \neg\exists\widehat{\mathbf{c}}(\widehat{\mathbf{c}} < \mathbf{c} \wedge F^*(\widehat{\mathbf{c}})),$$

where $F^*(\widehat{\mathbf{c}})$ is defined the same as the one in Section 2.1 except for the base case:

– When $F$ is an atomic formula, $F^*$ is $F' \wedge F$, where $F'$ is obtained from $F$ by replacing all intensional (function and predicate) constants in it with the corresponding (function and predicate) variables.

If $\mathbf{c}$ contains predicate constants only, this definition of a stable model reduces to the one in [Ferraris *et al.*, 2011], also reviewed in Section 2.1.

According to [Bartholomew and Lee, 2012], a *choice formula* $\{F\}$ is an abbreviation of the formula $F \vee \neg F$, which is also strongly equivalent to $\neg\neg F \rightarrow F$. A formula $\{\mathbf{t} = \mathbf{t}'\}$, where $\mathbf{t}$ contains an intensional function constant and $\mathbf{t}'$ does not, represents that $\mathbf{t}$ takes the value $\mathbf{t}'$ by default, as the following example demonstrates.

**Example 2** *Let $F_1$ be $\{f = 1\}$, which stands for $(f = 1) \vee \neg(f = 1)$, and $I_1$ be an interpretation such that $I_1(f) = 1$. Let's assume that we consider only interpretations that map numbers to themselves. $I_1$ is an $f$-stable model of $F_1$: $F_1^*(\widehat{f})$ is equivalent to $((\widehat{f}=1)\wedge(f=1))\vee\neg(f=1)$,[3] which is further equivalent to $(\widehat{f}=1)$ under the assumption $I_1$. It is not possible to satisfy this formula by assigning $\widehat{f}$ a different value from $I_1(f)$. On the other hand, $I_2$ such that $I_2(f) = 2$ is not $f$-stable since $F_1^*(\widehat{f})$ is equivalent to $\top$ under $I_2$, so that it is possible to satisfy this formula by assigning $\widehat{f}$ a different value from $I_2(f)$. If we let $F_2$ be $\{f = 1\} \wedge (f = 2)$, then $I_2$ is a $f$-stable of $F_2$, but $I_1$ is not: $F_2^*(\widehat{f})$ is equivalent to $\widehat{f}=2$ under $I_2$, so that $\widehat{f}$ has to map to 2 as well. This example illustrates the nonmonotonicity of the semantics.*

[3] It holds that $(\neg F)^*$ is equivalent to $\neg F$.

**Example 3** *The Blocks World can be described in this language as follows. For readability, we write in a logic program like syntax:*

$$\begin{aligned}
\bot &\leftarrow \mathit{Loc}(b_1,t)=b \wedge \mathit{Loc}(b_2,t)=b \wedge (b_1 \neq b_2)\\
\mathit{Loc}(b,t+1)=l &\leftarrow \mathit{Move}(b,l,t)\\
\bot &\leftarrow \mathit{Move}(b,l,t) \wedge \mathit{Loc}(b_1,t)=b\\
\bot &\leftarrow \mathit{Move}(b,b_1,t) \wedge \mathit{Move}(b_1,l,t)\\
\{\mathit{Loc}(b,0)=l\} &\\
\{\mathit{Move}(b,l,t)\} &\\
\{\mathit{Loc}(b,t+1)=l\} &\leftarrow \mathit{Loc}(b,t)=l\ .
\end{aligned}$$

*Here Loc is a function constant. The last rule is a default formula that describes the commonsense law of inertia. The stable models of this program are the models of* $\mathrm{SM}[F;\ \mathit{Loc}, \mathit{Move}]$*, where* $F$ *is the FOL-representation of the program.*

### 2.3 Review: Stable Models of Multi-Valued Propositional Formulas

The following is a review of the stable model semantics of multi-valued propositional formulas from [Bartholomew and Lee, 2012], which can be viewed as a special case of the functional stable model semantics in the previous section.

The syntax of multi-valued propositional formulas is given in [Ferraris *et al.*, 2011]. A *multi-valued propositional signature* is a set $\sigma$ of symbols called *constants*, along with a nonempty finite set $\mathit{Dom}(c)$ of symbols, disjoint from $\sigma$, assigned to each constant $c$. We call $\mathit{Dom}(c)$ the *domain* of $c$. A *Boolean* constant is one whose domain is the set $\{\text{TRUE}, \text{FALSE}\}$. An *atom* of a signature $\sigma$ is an expression of the form $c=v$ ("the value of $c$ is $v$") where $c \in \sigma$ and $v \in \mathit{Dom}(c)$. A *(multi-valued propositional) formula* of $\sigma$ is a propositional combination of atoms.

A *(multi-valued propositional) interpretation* of $\sigma$ is a function that maps every element of $\sigma$ to an element of its domain. An interpretation $I$ *satisfies* an atom $c=v$ (symbolically, $I \models c=v$) if $I(c) = v$. The satisfaction relation is extended from atoms to arbitrary formulas according to the usual truth tables for the propositional connectives. $I$ is a *model* of a formula if it satisfies the formula.

The reduct $F^I$ of a multi-valued propositional formula $F$ relative to a multi-valued propositional interpretation $I$ is the formula obtained from $F$ by replacing each maximal subformula that is not satisfied by $I$ with $\bot$. Interpretation $I$ is a *stable model* of $F$ if $I$ is the only interpretation satisfying $F^I$.

**Example 4** *Similar to Example 2, consider the signature* $\sigma = \{f\}$ *such that* $\mathit{Dom}(c) = \{1,2,3\}$*. Let* $I_1$ *be an interpretation such that* $I_1(c) = 1$*, and* $I_2$ *be such that* $I_2(c) = 2$*. Recall that* $\{f=1\}$ *is shorthand for* $(f=1) \vee \neg(f=1)$*. The reduct of this formula relative to* $I_1$ *is* $(f=1) \vee \bot$*, and* $I_1$ *is the only model of the reduct. On the other hand, the reduct of* $\{f=1\}$ *relative to* $I_2$ *is* $(\bot \vee \neg\bot)$ *and* $I_2$ *is not its unique model. Also, the reduct of* $\{f=1\} \wedge (f=2)$ *relative to* $I_1$ *is* $(\bot \vee \neg\bot) \wedge \bot$ *and* $I_1$ *is not a model. The reduct of* $\{f=1\} \wedge (f=2)$ *relative to* $I_2$ *is* $(\bot \vee \neg\bot) \wedge (f=2)$*, and* $I_2$ *is the only model of the reduct.*

## 3 Relating Strong Negation to Boolean Functions

### 3.1 Representing Strong Negation in Multi-Valued Propositional Formulas

Given a traditional propositional logic program $\Pi$ of a signature $\sigma$ [Gelfond and Lifschitz, 1991], we identify $\sigma$ with the multi-valued propositional signature whose constants are the

same symbols from $\sigma$ and every constant is Boolean. By $\Pi^{mv}$ we mean the multi-valued propositional formula that is obtained from $\Pi$ by replacing negative literals of the form $\sim p$ with $p = \text{FALSE}$ and positive literals of the form $p$ with $p = \text{TRUE}$.

We say that a set $X$ of literals from $\sigma$ is *complete* if, for each atom $a \in \sigma$, either $a$ or $\sim a$ is in $X$. We identify a complete set of literals from $\sigma$ with the corresponding multi-valued propositional interpretation.

**Theorem 1** *A complete set of literals is an answer set of $\Pi$ in the sense of [Gelfond and Lifschitz, 1991] iff it is a stable model of $\Pi^{mv}$ in the sense of [Bartholomew and Lee, 2012].*

The theorem tells us that checking the minimality of positive and negative literals under the traditional stable model semantics is essentially the same as checking the uniqueness of corresponding function values under the stable model semantics from [Bartholomew and Lee, 2012].

**Example 5** *Consider the program that describes a simple transition system consisting of two states depending on whether fluent $p$ is true or false, and an action that makes $p$ true (subscripts $0$ and $1$ represent time stamps).*

$$\begin{array}{rcl@{\qquad\qquad}rcl} p_0 & \leftarrow & \mathit{not}\ \sim p_0 & p_1 & \leftarrow & a \\ \sim p_0 & \leftarrow & \mathit{not}\ p_0 & & & \\ & & & p_1 & \leftarrow & p_0, \mathit{not}\ \sim p_1 \\ a & \leftarrow & \mathit{not}\ \sim a & \sim p_1 & \leftarrow & \sim p_0, \mathit{not}\ p_1\ . \\ \sim a & \leftarrow & \mathit{not}\ a & & & \end{array} \tag{8}$$

*The program has four answer sets, each of which corresponds to one of the four edges of the transition system. For instance, $\{\sim p_0, a, p_1\}$ is an answer set. This program can be encoded in the input languages of* GRINGO *and* DLV*. In the input language of* DLV*, which allows disjunctions in the head of a rule, the four rules in the first column can be succinctly replaced by*

$$p_0 \vee \sim p_0 \qquad\qquad a \vee \sim a\ .$$

*According to Theorem 1, the stable models of this program are the same as the stable models of the following multi-valued propositional formula (written in a logic program style syntax; '$\neg$' represents default negation):*

$$\begin{array}{rcl@{\qquad\qquad}rcl} p_0 = \text{TRUE} & \leftarrow & \neg(p_0 = \text{FALSE}) & p_1 = \text{TRUE} & \leftarrow & a = \text{TRUE} \\ p_0 = \text{FALSE} & \leftarrow & \neg(p_0 = \text{TRUE}) & & & \\ & & & p_1 = \text{TRUE} & \leftarrow & p_0 = \text{TRUE} \wedge \neg(p_1 = \text{FALSE}) \\ a = \text{TRUE} & \leftarrow & \neg(a = \text{FALSE}) & p_1 = \text{FALSE} & \leftarrow & p_0 = \text{FALSE} \wedge \neg(p_1 = \text{TRUE})\ . \\ a = \text{FALSE} & \leftarrow & \neg(a = \text{TRUE}) & & & \end{array}$$

### 3.2 Relation among Strong Negation, Default Negation, Choice Rules and Boolean Functions

In certain cases, strong negation can be replaced by default negation, and furthermore the expression can be rewritten in terms of choice rules, which often yields a succinct representation.

The following theorem, which extends the *Theorem on Double Negation* from [Ferraris *et al.*, 2009] to allow intensional functions, presents a condition under which equivalent transformations in classical logic preserve stable models.

**Theorem 2** *Let $F$ be a sentence, let $\mathbf{c}$ be a list of predicate and function constants, and let $I$ be a (coherent) interpretation. Let $F'$ be the sentence obtained from $F$ by replacing a subformula $\neg H$ with $\neg H'$ such that $I \models \widetilde{\forall}(H \leftrightarrow H')$. Then*

$$I \models \mathrm{SM}[F;\ \mathbf{c}] \textit{ iff } I \models \mathrm{SM}[F';\ \mathbf{c}]\ .$$

We say that an interpretation is *complete* on a predicate $p$ if it satisfies $\forall \mathbf{x}(p(\mathbf{x}) \vee \sim p(\mathbf{x}))$. It is clear that, for any complete interpretation $I$, we have $I \models \sim p(\mathbf{t})$ iff $I \models \neg p(\mathbf{t})$. This fact allows us to use Theorem 2 to replace strong negation occurring in $H$ with default negation.

**Example 5 continued** *Each answer set of the first program in Example 5 is complete. In view of Theorem 2, the first two rules can be rewritten as $p_0 \ \leftarrow\ $ not not $p_0$ and $\sim p_0 \ \leftarrow\ $ not not $\sim p_0$, which can be further abbreviated as choice rules $\{p_0\}$ and $\{\sim p_0\}$. Consequently, the whole program can be rewritten using choice rules as*

$$\begin{array}{cc} \{p_0\} & p_1 \ \leftarrow\ a \\ \{\sim p_0\} & \\ & \{p_1\} \ \leftarrow\ p_0 \\ \{a\} & \{\sim p_1\} \ \leftarrow \sim p_0\ . \\ \{\sim a\} & \end{array}$$

*Similarly, since $I \models (p_0 = \mathrm{FALSE})$ iff $I \models \neg(p_0 = \mathrm{TRUE})$, in view of Theorem 2, the first rule of the second program in Example 5 can be rewritten as $p_0 = \mathrm{TRUE} \ \leftarrow\ \neg\neg(p_0 = \mathrm{TRUE})$ and further as $\{p_0 = \mathrm{TRUE}\}$. This transformation allows us to rewrite the whole program as*

$$\begin{array}{cc} \{p_0 = B\} & p_1 = \mathrm{TRUE} \ \leftarrow\ a = \mathrm{TRUE} \\ \{a = B\} & \{p_1 = B\} \ \leftarrow\ p_0 = B\ , \end{array}$$

*where $B$ ranges over $\{\mathrm{TRUE}, \mathrm{FALSE}\}$. This program represents the transition system more succinctly than program* (8)*.*

### 3.3 Representing Strong Negation by Boolean Functions in the First-Order Case

Theorem 1 can be extended to the first-order case as follows.

Let $f$ be a function constant. A first-order formula is called *$f$-plain* if each atomic formula

- does not contain $f$, or
- is of the form $f(\mathbf{t}) = u$ where $\mathbf{t}$ is a tuple of terms not containing $f$, and $u$ is a term not containing $f$.

For example, $f = 1$ is $f$-plain, but each of $p(f)$, $g(f) = 1$, and $1 = f$ is not $f$-plain.

For a list $\mathbf{c}$ of predicate and function constants, we say that a first-order formula $F$ is $\mathbf{c}$-plain if $F$ is $f$-plain for each function constant $f$ in $\mathbf{c}$. Roughly speaking, $\mathbf{c}$-plain formulas do not allow the functions in $\mathbf{c}$ to be nested in another predicate or function, and at most one function in $\mathbf{c}$ is allowed in each atomic formula. For example, $f = g$ is not $(f, g)$-plain, and neither is $f(g) = 1 \rightarrow g = 1$.

Let $F$ be a formula whose signature contains both positive and negative predicate constants $p$ and $\sim p$. Formula $F^{(p,\sim p)}_{b}$ is obtained from $F$ as follows:

– in the signature of $F$, replace $p$ and $\sim p$ with a new intensional function constant $b$ of arity $n$, where $n$ is the arity of $p$ (or $\sim p$), and add two non-intensional object constants TRUE and FALSE;
– replace every occurrence of $\sim p(\mathbf{t})$, where $\mathbf{t}$ is a list of terms, with $b(\mathbf{t}) = \text{FALSE}$, and then replace every occurrence of $p(\mathbf{t})$ with $b(\mathbf{t}) = \text{TRUE}$.

By *BC*$_b$ ("Boolean Constraint on $b$") we denote the conjunction of the following formulas, which asserts that $b$ is a Boolean function:

$$\text{TRUE} \neq \text{FALSE}\ , \tag{9}$$

$$\neg\neg\forall\mathbf{x}(b(\mathbf{x}) = \text{TRUE} \vee b(\mathbf{x}) = \text{FALSE})\ ,$$

where $\mathbf{x}$ is a list of distinct object variables.

**Theorem 3** *Let* $\mathbf{c}$ *be a set of predicate and function constants, and let* $F$ *be a* $\mathbf{c}$*-plain formula. Formulas*

$$\forall\mathbf{x}((p(\mathbf{x}) \leftrightarrow b(\mathbf{x}){=}\text{TRUE}) \wedge (\sim p(\mathbf{x}) \leftrightarrow b(\mathbf{x}){=}\text{FALSE})), \tag{10}$$

*and BC*$_b$ *entail*

$$\text{SM}[F;\ p, \sim p, \mathbf{c}] \leftrightarrow \text{SM}[F^{(p,\sim p)}_{b};\ b, \mathbf{c}]\ .$$

If we drop the requirement that $F$ be $\mathbf{c}$-plain, the statement does not hold as in the following example demonstrates.

**Example 6** *Take* $\mathbf{c}$ *as* $(f, g)$ *and let* $F$ *be* $p(f) \wedge\ \sim p(g)$. $F^{(p,\sim p)}_{b}$ *is* $b(f) = \text{TRUE} \wedge b(g) = \text{FALSE}$. *Consider the interpretation* $I$ *whose universe is* $\{1, 2\}$ *such that* $I$ *contains* $p(1), \sim p(2)$ *and with the mappings* $b^I(1) = \text{TRUE}, b^I(2) = \text{FALSE}, f^I = 1, g^I = 2$. $I$ *certainly satisfies BC*$_b$ *and (10).* $I$ *also satisfies* $\text{SM}[F;\ p, \sim p, f, g]$ *but does not satisfy* $\text{SM}[F^{(p,\sim p)}_{b};\ b, f, g]$*: we can let* $I$ *be* $\widehat{b}^I(1) = \text{FALSE}, \widehat{b}^I(2) = \text{TRUE}, \widehat{f}^I = 2, \widehat{g}^I = 1$ *to satisfy both* $(\widehat{b}, \widehat{f}, \widehat{g}) < (b, f, g)$ *and* $(F^{(p,\sim p)}_{b})^*(\widehat{b}, \widehat{f}, \widehat{g})$*, which is*

$$b(f) = \text{TRUE} \wedge \widehat{b}(\widehat{f}) = \text{TRUE} \wedge b(g) = \text{FALSE} \wedge \widehat{b}(\widehat{g}) = \text{FALSE}.$$

Note that any interpretation that satisfies both (10) and *BC*$_b$ is complete on $p$. Theorem 3 tells us that, for any interpretation $I$ that is complete on $p$, minimizing the extents of both $p$ and $\sim p$ has the same effect as ensuring that the corresponding Boolean function $b$ have a unique value.

The following corollary shows that there is a 1–1 correspondence between the stable models of $F$ and the stable models of $F^{(p,\sim p)}_{b}$. For any interpretation $I$ of the signature of $F$ that is complete on $p$, by $I^{(p,\sim p)}_{b}$ we denote the interpretation of the signature of $F^{(p,\sim p)}_{b}$ obtained from $I$ by replacing the relation $p^I$ with function $b^I$ such that

$$b^I(\xi_1, \ldots, \xi_n) = \text{TRUE}^I \ \text{ if } \ p^I(\xi_1, \ldots, \xi_n) = \text{TRUE};$$
$$b^I(\xi_1, \ldots, \xi_n) = \text{FALSE}^I \ \text{ if } \ (\sim p)^I(\xi_1, \ldots, \xi_n) = \text{TRUE}\ .$$

(Notice that we overloaded the symbols TRUE and FALSE: object constants on one hand, and truth values on the other hand.) Since $I$ is complete on $p$ and coherent, $b^I$ is well-defined. We also require that $I^{(p,\sim p)}_{b}$ satisfy (9). Consequently, $I^{(p,\sim p)}_{b}$ satisfies *BC*$_b$.

**Corollary 1** *Let* $\mathbf{c}$ *be a set of predicate and function constants, and let* $F$ *be a* $\mathbf{c}$*-plain sentence.* (I) *An interpretation* $I$ *of the signature of* $F$ *that is complete on* $p$ *is a model of* $\text{SM}[F;\ p, \sim p, \mathbf{c}]$ *iff* $I^{(p,\sim p)}_{b}$ *is a model of* $\text{SM}[F^{(p,\sim p)}_{b};\ b, \mathbf{c}]$. (II) *An interpretation* $J$ *of the signature of* $F^{(p,\sim p)}_{b}$ *is a model of* $\text{SM}[F^{(p,\sim p)}_{b} \wedge BC_b;\ b, \mathbf{c}]$ *iff* $J = I^{(p,\sim p)}_{b}$ *for some model* $I$ *of* $\text{SM}[F;\ p, \sim p, \mathbf{c}]$.

The other direction, eliminating Boolean intensional functions in favor of symmetric predicates, is similar as we show in the following.

Let $F$ be a $(b, \mathbf{c})$-plain formula such that every atomic formula containing $b$ has the form $b(\mathbf{t}) = \text{TRUE}$ or $b(\mathbf{t}) = \text{FALSE}$, where $\mathbf{t}$ is any list of terms (not containing members from $(b, \mathbf{c})$). Formula $F^{\,b}_{(p,\sim p)}$ is obtained from $F$ as follows:

- in the signature of $F$, replace $b$ with predicate constants $p$ and $\sim p$, whose arities are the same as that of $b$;
- replace every occurrence of $b(\mathbf{t}) = \text{TRUE}$, where $\mathbf{t}$ is any list of terms, with $p(\mathbf{t})$, and $b(\mathbf{t}) = \text{FALSE}$ with $\sim p(\mathbf{t})$.

**Theorem 4** *Let* $\mathbf{c}$ *be a set of predicate and function constants, let* $b$ *be a function constant, and let* $F$ *be a* $(b, \mathbf{c})$*-plain formula such that every atomic formula containing* $b$ *has the form* $b(\mathbf{t}) = \text{TRUE}$ *or* $b(\mathbf{t}) = \text{FALSE}$. *Formulas (10) and* $BC_b$ *entail*

$$\text{SM}[F;\ b, \mathbf{c}] \leftrightarrow \text{SM}[F^{\,b}_{(p,\sim p)};\ p, \sim p, \mathbf{c}]\ .$$

The following corollary shows that there is a 1–1 correspondence between the stable models of $F$ and the stable models of $F^{\,b}_{(p,\sim p)}$. For any interpretation $I$ of the signature of $F$ that satisfies $BC_b$, by $I^{\,b}_{(p,\sim p)}$ we denote the interpretation of the signature of $F^{\,b}_{(p,\sim p)}$ obtained from $I$ by replacing the function $b^I$ with predicate $p^I$ such that

$$p^I(\xi_1, \ldots, \xi_n) = \text{TRUE} \ \text{ iff } \ b^I(\xi_1, \ldots, \xi_n) = \text{TRUE}^I\ ;$$
$$(\sim p)^I(\xi_1, \ldots, \xi_n) = \text{TRUE} \ \text{ iff } \ b^I(\xi_1, \ldots, \xi_n) = \text{FALSE}^I\ .$$

**Corollary 2** *Let* $\mathbf{c}$ *be a set of predicate and function constants, let* $b$ *be a function constant, and let* $F$ *be a* $(b, \mathbf{c})$*-plain sentence such that every atomic formula containing* $b$ *has the form* $b(\mathbf{t}) = \text{TRUE}$ *or* $b(\mathbf{t}) = \text{FALSE}$. (I) *An interpretation* $I$ *of the signature of* $F$ *is a model of* $\text{SM}[F \wedge BC_b;\ b, \mathbf{c}]$ *iff* $I^{\,b}_{(p,\sim p)}$ *is a model of* $\text{SM}[F^{\,b}_{(p,\sim p)};\ p, \sim p, \mathbf{c}]$. (II) *An interpretation* $J$ *of the signature of* $F^{\,b}_{(p,\sim p)}$ *is a model of* $\text{SM}[F^{\,b}_{(p,\sim p)};\ p, \sim p, \mathbf{c}]$ *iff* $J = I^{\,b}_{(p,\sim p)}$ *for some model* $I$ *of* $\text{SM}[F \wedge BC_b;\ b, \mathbf{c}]$.

An example of this corollary is shown in the next section.

## 4 Representing Non-Boolean Functions Using Strong Negation

In this section, we show how to eliminate non-Boolean intensional functions in favor of Boolean intensional functions. Combined with the method in the previous section, it gives us a systematic method of representing non-Boolean intensional functions using strong negation.

### 4.1 Eliminating non-Boolean Functions in Favor of Boolean Functions

Let $F$ be an $f$-plain formula. Formula $F_b^f$ is obtained from $F$ as follows:

- in the signature of $F$, replace $f$ with a new boolean intensional function $b$ of arity $n+1$ where $n$ is the arity of $f$;
- replace each subformula $f(\mathbf{t}) = c$ with $b(\mathbf{t}, c) = \text{TRUE}$.

By $UE_b$, we denote the following formulas that preserve the functional property:

$$\forall \mathbf{x}yz(y \neq z \wedge b(\mathbf{x}, y) = \text{TRUE} \rightarrow b(\mathbf{x}, z) = \text{FALSE}),$$

$$\neg\neg\forall \mathbf{x}\exists y(b(\mathbf{x}, y) = \text{TRUE}),$$

where $\mathbf{x}$ is a $n$-tuple of variables and all variables in $\mathbf{x}$, $y$, and $z$ are pairwise distinct.

**Theorem 5** *For any $f$-plain formula $F$,*

$$\forall \mathbf{x}y\big((f(\mathbf{x}) = y \leftrightarrow b(\mathbf{x}, y) = \text{TRUE}) \wedge (f(\mathbf{x}) \neq y \leftrightarrow b(\mathbf{x}, y) = \text{FALSE})\big)$$

*and $\exists xy(x \neq y)$ entail*

$$\text{SM}[F;\ f, \mathbf{c}] \ \leftrightarrow\ \text{SM}[F_b^f \wedge UE_b;\ b, \mathbf{c}]\ .$$

By $I_b^f$, we denote the interpretation of the signature of $F_b^f$ obtained from $I$ by replacing the mapping $f^I$ with the mapping $b^I$ such that

$$\begin{array}{ll} b^I(\xi_1, \ldots, \xi_n, \xi_{n+1}) = \text{TRUE}^I & \text{if } f^I(\xi_1, \ldots, \xi_n) = \xi_{n+1} \\ b^I(\xi_1, \ldots, \xi_n, \xi_{n+1}) = \text{FALSE}^I & \text{otherwise.} \end{array}$$

**Corollary 3** *Let $F$ be an $f$-plain sentence.* (I) *An interpretation $I$ of the signature of $F$ that satisfies $\exists xy(x \neq y)$ is a model of* SM$[F;\ f, \mathbf{c}]$ *iff $I_b^f$ is a model of* SM$[F_b^f \wedge UE_b;\ b, \mathbf{c}]$. (II) *An interpretation $J$ of the signature of $F_b^f$ that satisfies $\exists xy(x \neq y)$ is a model of* SM$[F_b^f \wedge UE_b;\ b, \mathbf{c}]$ *iff $J = I_b^f$ for some model $I$ of* SM$[F;\ f, \mathbf{c}]$.

**Example 3 continued** *In the program in Example 3, we eliminate non-Boolean function Loc in favor of Boolean function On as follows. The last two rules are $UE_{On}$.*

$$\begin{array}{rcl} \bot & \leftarrow & \mathit{On}(b_1, b, t) = \text{TRUE} \wedge \mathit{On}(b_2, b, t) = \text{TRUE} \wedge b_1 \neq b_2 \\ \mathit{On}(b, l, t+1) = \text{TRUE} & \leftarrow & \mathit{Move}(b, l, t) \\ \bot & \leftarrow & \mathit{Move}(b, l, t) \wedge \mathit{On}(b_1, b, t) = \text{TRUE} \\ \bot & \leftarrow & \mathit{Move}(b, b_1, t) \wedge \mathit{Move}(b_1, l, t) \\ \{\mathit{On}(b, l, 0) = \text{TRUE}\} & & \\ \{\mathit{Move}(b, l, t)\} & & \\ \{\mathit{On}(b, l, t+1) = \text{TRUE}\} & \leftarrow & \mathit{On}(b, l, t) = \text{TRUE} \\ \mathit{On}(b, l, t) = \text{FALSE} & \leftarrow & \mathit{On}(b, l_1, t) = \text{TRUE} \wedge l \neq l_1 \\ \bot & \leftarrow & \mathit{not}\ \exists l(\mathit{On}(b, l, t) = \text{TRUE})\ . \end{array}$$

*For this program, it is not difficult to check that the last rule is redundant. Indeed, since the second to the last rule is the only rule that has $\mathit{On}(b, l, t) = \text{FALSE}$ in the head, one can check that any model that does not satisfy $\exists l(\mathit{On}(b, l, t) = \text{TRUE})$ is not stable even if we drop the last rule.*

*Corollary 2 tells us that this program can be represented by an answer set program containing strong negation (with the redundant rule dropped).*

$$\begin{array}{rcll}
\bot & \leftarrow & \mathit{On}(b_1,b,t), \mathit{On}(b_2,b,t) & (b_1 \neq b_2) \\
\mathit{On}(b,l,t+1) & \leftarrow & \mathit{Move}(b,l,t) & \\
\bot & \leftarrow & \mathit{Move}(b,l,t), \mathit{On}(b_1,b,t) & \\
\bot & \leftarrow & \mathit{Move}(b,b_1,t), \mathit{Move}(b_1,l,t) & \\
\{\mathit{On}(b,l,0)\} & & & \\
\{\mathit{Move}(b,l,t)\} & & & \\
\{\mathit{On}(b,l,t+1)\} & \leftarrow & \mathit{On}(b,l,t) & \\
\sim\mathit{On}(b,l,t) & \leftarrow & \mathit{On}(b,l_1,t) & (l \neq l_1)\ .
\end{array} \tag{11}$$

*Let us compare this program with program* (7)*. Similar to the explanation in Example 5 (continued), the 5th and the 7th rules of* (7) *can be represented using choice rules, which are the same as the 5th and the 6th rules of* (11)*. The 6th and the 8th rules of* (7) *represent the closed world assumption. We can check that adding these rules to* (11) *extends the answer sets of* (7) *in a conservative way with the definition of the negative literals. This tells us that the answer sets of the two programs are in a 1-1 correspondence.*

As the example explains, non-Boolean functions can be represented using strong negation by composing the two translations, first eliminating non-Boolean functions in favor of Boolean functions as in Corollary 3 and then eliminating Boolean functions in favor of predicates as in Corollary 2. In the following we state this composition.

Let $F$ be an $f$-plain formula where $f$ is an intensional function constant. Formula $F_p^f$ is obtained from $F$ as follows:

- in the signature of $F$, replace $f$ with two new intensional predicates $p$ and $\sim p$ of arity $n+1$ where $n$ is the arity of $f$;
- replace each subformula $f(\mathbf{t}) = c$ with $p(\mathbf{t}, c)$.

By $UE_p$, we denote the following formulas that preserve the functional property:

$$\forall \mathbf{x}yz(y \neq z \wedge p(\mathbf{x},y) \rightarrow \sim p(\mathbf{x},z))\ ,$$
$$\neg\neg\forall \mathbf{x}\exists y\ p(\mathbf{x},y)\ ,$$

where $\mathbf{x}$ is an $n$-tuple of variables and all variables in $\mathbf{x}, y, z$ are pairwise distinct.

**Theorem 6** *For any $(f,\mathbf{c})$-plain formula $F$, formulas*

$$\forall \mathbf{x}y(f(\mathbf{x}) = y \leftrightarrow p(\mathbf{x},y)),\ \ \forall \mathbf{x}y(f(\mathbf{x}) \neq y \leftrightarrow \sim p(\mathbf{x},y)),\ \ \exists xy(x \neq y)$$

*entail*

$$\mathrm{SM}[F;\ f,\mathbf{c}] \leftrightarrow \mathrm{SM}[F_p^f \wedge UE_p;\ p, \sim p, \mathbf{c}]\ .$$

By $I^{\,f}_{(p,\sim p)}$, we denote the interpretation of the signature of $F^{\,f}_{(p,\sim p)}$ obtained from $I$ by replacing the function $f^I$ with the relation $p^I$ that consists of the tuples $\langle \xi_1, \ldots, \xi_n, f^I(\xi_1,\ldots,\xi_n)\rangle$ for all $\xi_1,\ldots,\xi_n$ from the universe of $I$. We then also add the set $(\sim p)^I$ that consists of the tuples $\langle \xi_1,\ldots,\xi_n,\xi_{n+1}\rangle$ for all $\xi_1,\ldots,\xi_n,\xi_{n+1}$ from the universe of $I$ that do not occur in the set $p^I$.

**Corollary 4** *Let $F$ be an $(f, \mathbf{c})$-plain sentence.* (I) *An interpretation $I$ of the signature of $F$ that satisfies $\exists xy(x \neq y)$ is a model of* SM$[F;\ f, \mathbf{c}]$ *iff $I^f_{(p,\sim p)}$ is a model of* SM$[F^f_p \wedge UE_p;\ p, \sim p, \mathbf{c}]$. (II) *An interpretation $J$ of the signature of $F^f_p$ that satisfies $\exists xy(x \neq y)$ is a model of* SM$[F^f_p \wedge UE_p;\ p, \sim p, \mathbf{c}]$ *iff $J = I^f_{(p,\sim p)}$ for some model $I$ of* SM$[F;\ f, \mathbf{c}]$.

Theorem 6 and Corollary 4 are similar to Theorem 8 and Corollary 2 from [Bartholomew and Lee, 2012]. The main difference is that the latter statements refer to the constraint called $UEC_p$ that is weaker than $UE_p$. For instance, the elimination method from [Bartholomew and Lee, 2012] turns the Blocks World in Example 3 into almost the same program as (11) except that the last rule is turned into the constraint $UEC_{On}$:

$$\leftarrow\ On(b, l, t) \wedge On(b, l_1, t) \wedge l \neq l_1\ . \tag{12}$$

It is clear that the stable models of $F^f_p \wedge UE_p$ are under the symmetric view, and the stable models of $F^f_p \wedge UEC_p$ are under the asymmetric view. To see how replacing $UE_{On}$ by $UEC_{On}$ turns the symmetric view to the asymmetric view, first observe that adding (12) to program (11) does not affect the stable models of the program. Let's call this program $\Pi$. It is easy to see that $\Pi$ is a conservative extension of the program that is obtained from $\Pi$ by deleting the rule with $\sim On(b, l, t)$ in the head.

## 5 Relating to Lifschitz's Two-Valued Logic Programs

Lifschitz [2012] presented a high level definition of a logic program that does not contain explicit default negation, but can handle nonmonotonic reasoning in a similar style as in Reiter's default logic. In this section we show how his formalism can be viewed as a special case of multi-valued propositional formulas under the stable model semantics in which every function is Boolean.

### 5.1 Review: Two-Valued Logic Programs

Let $\sigma$ be a signature in propositional logic. A *two-valued rule* is an expression of the form

$$L_0\ \leftarrow\ L_1, \ldots, L_n : F \tag{13}$$

where $L_0, \ldots, L_n$ are propositional literals formed from $\sigma$ and $F$ is a propositional formula of signature $\sigma$.

A *two-valued program $\Pi$* is a set of two-valued rules. An interpretation $I$ is a function from $\sigma$ to $\{\text{TRUE}, \text{FALSE}\}$. The *reduct* of a program $\Pi$ relative to an interpretation $I$, denoted $\Pi^I$, is the set of rules $L_0\ \leftarrow\ L_1, \ldots, L_n$ corresponding to the rules (13) of $\Pi$ for which $I \models F$. Interpretation $I$ is a stable model of $\Pi$ if it is a minimal model of $\Pi^I$.

**Example 7**

$$a\ \leftarrow\ :\ a, \qquad \neg a\ \leftarrow :\ \neg a, \qquad b\ \leftarrow\ a\ :\ \top \tag{14}$$

*The reduct of this program relative to $\{a, b\}$ consists of rules $a$ and $b\ \leftarrow\ a$. Interpretation $\{a, b\}$ is the minimal model of the reduct, so that it is a stable model of the program.*

As described in [Lifschitz, 2012], if $F$ in every rule (13) has the form of conjunctions of literals, then the two-valued logic program can be turned into a traditional answer set

program containing strong negation when we consider complete answer sets only. For instance, program (14) can be turned into

$$a \leftarrow \mathit{not}\ {\sim}a, \qquad {\sim}a \leftarrow \mathit{not}\ a, \qquad b \leftarrow a\ .$$

This program has two answer sets, $\{a, b\}$ and ${\sim}a$, and only the complete answer set $\{a, b\}$ corresponds to the stable model found in Example 7.

### 5.2 Translation into SM with Boolean Functions

Given a two-valued logic program $\Pi$ of a signature $\sigma$, we identify $\sigma$ with the multi-valued propositional signature whose constants are from $\sigma$ and the domain of every constant is Boolean values $\{\text{TRUE}, \text{FALSE}\}$. For any propositional formula $G$, $Tr(G)$ is obtained from $G$ by replacing every negative literal ${\sim}A$ with $A = \text{FALSE}$ and every positive literal $A$ with $A = \text{TRUE}$. By *tv2sm*$(\Pi)$ we denote the multi-valued propositional formula which is defined as the conjunction of

$$\neg\neg Tr(F) \wedge Tr(L_1) \wedge \cdots \wedge Tr(L_n) \rightarrow Tr(L_0)$$

for each rule (13) in $\Pi$.

For any interpretation $I$ of $\sigma$, we obtain the multi-valued interpretation $I'$ from $I$ as follows. For each atom $A$ in $\sigma$,

$$I'(A) = \begin{cases} \text{TRUE} & \text{if } I \models A \\ \text{FALSE} & \text{if } I \models \neg A \end{cases}$$

**Theorem 7** *For any two-valued logic program $\Pi$, an interpretation $I$ is a stable model of $\Pi$ in the sense of [Lifschitz, 2012] iff $I'$ is a stable model of tv2sm$(\Pi)$ in the sense of [Bartholomew and Lee, 2012].*

**Example 7 continued** *For the program $\Pi$ in Example 7, tv2sm$(\Pi)$ is the following multi-valued propositional formula:*

$$\big(\neg\neg(a = \text{TRUE}) \rightarrow a = \text{TRUE}\big) \wedge\ \big(\neg\neg(a = \text{FALSE}) \rightarrow a = \text{FALSE}\big) \wedge\ \big(a = \text{TRUE} \rightarrow b = \text{TRUE}\big).$$

*According to [Bartholomew and Lee, 2012], this too has only one stable model in which $a$ and $b$ are both mapped to* TRUE*, corresponding to the only stable model of $\Pi$ according to Lifschitz.*

Consider extending the rules (13) to contain variables. It is not difficult to see that the translation *tv2sm*$(\Pi)$ can be straightforwardly extended to non-ground programs. This accounts for providing the semantics of the first-order extension of two-valued logic programs.

## 6 Strong Negation and the Cabalar Semantics

There are other stable model semantics of intensional functions. Theorem 5 from [Bartholomew and Lee, 2013] states that the semantics by Bartholomew and Lee [2013] coincides with the semantics by Cabalar [2011] on c-plain formulas. Thus several theorems in this note stated for the Bartholomew-Lee semantics hold also under the Cabalar semantics.

A further result holds with the Cabalar semantics since it allows functions to be partial. This provides extensions of Theorem 3 and Corollary 1, which do not require the interpretations to be complete. Below we state this result. Due to lack of space, we refer the reader

to [Bartholomew and Lee, 2013] for the definition of CBL, which is the second-order expression used to define the Cabalar semantics.

Similar to $BC_b$ in Section 3.3, by $BC'_b$ we denote the conjunction of the following formulas:

$$\text{TRUE} \neq \text{FALSE}, \tag{15}$$

$$\neg\neg\forall\mathbf{x}(b(\mathbf{x}) = \text{TRUE} \vee b(\mathbf{x}) = \text{FALSE} \vee b(\mathbf{x}) \neq b(\mathbf{x})),$$

where $\mathbf{x}$ is a list of distinct object variables.[4]

**Theorem 8** *Let* $\mathbf{c}$ *be a set of predicate constants, and let* $F$ *be a formula. Formulas*

$$\forall\mathbf{x}((p(\mathbf{x}) \leftrightarrow b(\mathbf{x}) = \text{TRUE}) \wedge (\sim p(\mathbf{x}) \leftrightarrow b(\mathbf{x}) = \text{FALSE}) \wedge (\neg p(\mathbf{x}) \wedge \neg \sim p(\mathbf{x}) \leftrightarrow b(\mathbf{x}) \neq b(\mathbf{x})),$$

*and* $BC'_b$ *entail* [5]

$$\text{SM}[F;\ p, \sim p, \mathbf{c}] \leftrightarrow \text{CBL}[F^{(p,\sim p)}_b;\ b, \mathbf{c}]\ .$$

The following corollary shows that there is a 1–1 correspondence between the stable models of $F$ and the stable models of $F^{(p,\sim p)}_b$.[6] For any interpretation $I$ of the signature of $F$, by $I^{(p,\sim p)}_b$ we denote the interpretation of the signature of $F^{(p,\sim p)}_b$ obtained from $I$ by replacing the relation $p^I$ with function $b^I$ such that

$$\begin{array}{l} b^I(\boldsymbol{\xi}) = \text{TRUE}^I \ \text{ if } \ p^I(\boldsymbol{\xi}) = \text{TRUE}\ ; \\ b^I(\boldsymbol{\xi}) = \text{FALSE}^I \ \text{ if } \ (\sim p)^I(\boldsymbol{\xi}) = \text{TRUE}\ ; \\ b^I(\boldsymbol{\xi}) = u \ \text{ if } \ p^I(\boldsymbol{\xi}) = (\sim p)^I(\boldsymbol{\xi}) = \text{FALSE}\ . \end{array}$$

Since $I$ is coherent, $b^I$ is well-defined. We also require that $I^{(p,\sim p)}_b$ satisfy (15). Consequently, $I^{(p,\sim p)}_b$ satisfies $BC'_b$.

**Corollary 5** *Let* $F$ *be a sentence, and let* $\mathbf{c}$ *be a set of predicate constants.* (I) *An interpretation* $I$ *of the signature of* $F$ *is a model of* $\text{SM}[F;\ p, \sim p, \mathbf{c}]$ *iff* $I^{(p,\sim p)}_b$ *is a model of* $\text{CBL}[F^{(p,\sim p)}_b;\ b, \mathbf{c}]$. (II) *An interpretation* $J$ *of the signature of* $F^{(p,\sim p)}_b$ *is a model of* $\text{CBL}[F^{(p,\sim p)}_b \wedge BC'_b;\ b, \mathbf{c}]$ *iff* $J = I^{(p,\sim p)}_b$ *for some model* $I$ *of* $\text{SM}[F; p, \sim p, \mathbf{c}]$.

## 7 Conclusion

In this note, we showed that, under complete interpretations, symmetric predicates using strong negation can be alternatively expressed in terms of Boolean intensional functions in the language of [Bartholomew and Lee, 2012]. They can also be expressed in terms of Boolean intensional functions in the language of [Cabalar, 2011], but without requiring the complete interpretation assumption, at the price of relying on the notion of partial interpretations.

System CPLUS2ASP [Casolary and Lee, 2011; Babb and Lee, 2013] turns action language $\mathcal{C}$+ into answer set programs containing asymmetric predicates. The translation in

[4] Under partial interpretations, $b(\mathbf{t}) \neq b(\mathbf{t})$ is true if $b(\mathbf{t})$ is undefined. See [Cabalar, 2011; Bartholomew and Lee, 2013] for more details.

[5] The entailment is under partial interpretations and satisfaction.

[6] Recall the notation defined in Section 3.3.

this paper that eliminates intensional functions in favor of symmetric predicates provides an alternative method of computing $\mathcal{C}+$ using ASP solvers.

**Acknowledgements:** We are grateful to Vladimir Lifschitz for bringing attention to this subject, to Gregory Gelfond for useful discussions related to this paper, and to anonymous referees for 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

[Babb and Lee, 2013] Joseph Babb and Joohyung Lee. CPLUS2ASP: Computing action language C+ in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2013. To appear.

[Bartholomew and Lee, 2012] Michael Bartholomew and Joohyung Lee. Stable models of formulas with intensional functions. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 2–12, 2012.

[Bartholomew and Lee, 2013] Michael Bartholomew and Joohyung Lee. On the stable model semantics for intensional functions. *TPLP*, 2013. To appear.

[Cabalar, 2011] Pedro Cabalar. Functional answer set programming. *TPLP*, 11(2-3):203–233, 2011.

[Casolary and Lee, 2011] Michael Casolary and Joohyung Lee. Representing the language of the causal calculator in answer set programming. In *ICLP (Technical Communications)*, pages 51–61, 2011.

[Ferraris *et al.*, 2009] Paolo Ferraris, Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Symmetric splitting in the general theory of stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 797–803. AAAI Press, 2009.

[Ferraris *et al.*, 2011] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.

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

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

[Lifschitz, 2012] Vladimir Lifschitz. Two-valued logic programs. In *ICLP (Technical Communications)*, pages 259–266, 2012.

# An Algebra of Causal Chains*

Pedro Cabalar and Jorge Fandinno

Department of Computer Science
University of Corunna, SPAIN
{cabalar, jorge.fandino}@udc.es

**Abstract.** In this work we propose a multi-valued extension of logic programs under the stable models semantics where each true atom in a model is associated with a set of justifications, in a similar spirit than a set of proof trees. The main contribution of this paper is that we capture justifications into an algebra of truth values with three internal operations: an addition '$+$' representing alternative justifications for a formula, a commutative product '$*$' representing joint interaction of causes and a non-commutative product '$\cdot$' acting as a concatenation or proof constructor. Using this multi-valued semantics, we obtain a one-to-one correspondence between the syntactic proof tree of a standard (non-causal) logic program and the interpretation of each true atom in a model. Furthermore, thanks to this algebraic characterization we can detect semantic properties like redundancy and relevance of the obtained justifications. We also identify a lattice-based characterization of this algebra, defining a direct consequences operator, proving its continuity and that its least fix point can be computed after a finite number of iterations. Finally, we define the concept of *causal stable model* by introducing an analogous transformation to Gelfond and Lifschitz's program reduct.

## 1 Introduction

A frequent informal way of explaining the effect of default negation in an introductory class on semantics in logic programming (LP) is that a literal of the form '*not* $p$' should be read as "there is no way to derive $p$." Although this idea seems quite intuitive, it is actually using a concept outside the discourse of any of the existing LP semantics: the *ways to derive* $p$. To explore this idea, [1] introduced the so-called *causal logic programs*. The semantics was an extension of stable models [2] relying on the idea of "justification" or "proof". Any true atom, in a standard (non-causal) stable model needs to be justified. In a *causal stable model*, the truth value of each true atom captures these possible justifications, called *causes*. Let us see an example to illustrate this.

*Example 1.* Suppose we have a row boat with two rowers, one at each side of the boat, port and starboard. The boat moves forward $fwd$ if both rowers strike at a time. On the other hand, if we have a following wind, the boat moves forward anyway. □

* This research was partially supported by Spanish MEC project TIN2009-14562-C05-04 and Xunta program INCITE 2011.

Suppose now that we have indeed that both rowers stroke at a time when we additionally had a following wind. A possible encoding for this example could be the set of rules $\Pi_1$:

$$p : port \quad s : starb \quad w : fwind$$
$$fwd \leftarrow port \wedge starb \qquad fwd \leftarrow fwind$$

In the only causal stable model of this program, atom $fwd$ was justified by two alternative and independent causes. On the one hand, cause $\{p, s\}$ representing the joint interaction of $port$ and $starb$. On the other hand, cause $\{w\}$ inherited from $fwind$. We label rules (in the above program only atoms) that we want to be reflected in causes. Unlabelled $fwd$ rules are just ignored when reflecting causal information. For instance, if we decide to keep track of the application of these rules, we would handle instead a program $\Pi_2$ obtained just by labelling these two rules in $\Pi_1$ as follows:

$$a : fwd \leftarrow port \wedge starb \tag{1}$$
$$b : fwd \leftarrow fwind \tag{2}$$

The two alternative justifications for atom $fwd$ become the pair of causes $\{p, s\} \cdot a$ and $\{w\} \cdot b$. The informal reading of $\{p, s\} \cdot a$ is that "the joint interaction of $\{p\}$ and $\{s\}$, the cause $\{p, s\}$, is necessary to apply rule $a$." From a graphical point of view, we can represent *causes* as proof trees.

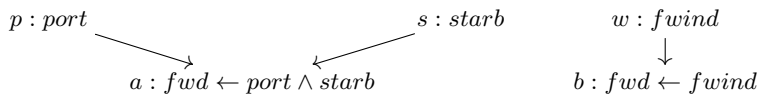



**Fig. 1.** Proof trees justifying atom $fwd$ in the program $\Pi_2$

In this paper, we show that causes can be embedded in an algebra with three internal operations: an addition '$+$' representing alternative justifications for a formula, a commutative product '$*$' representing joint interaction of causes (in a similar spirit to the '$+$' in [3]) and a non-commutative product '$\cdot$' acting as a concatenation or rule application. Using these operations, we can see that justification for $fwd$ would correspond now to the value $((p * s) \cdot a) + (w \cdot b)$ which means that $fwd$ is justified by the two alternative *causes*, $(p * s) \cdot a$ and $(w \cdot b)$. The former refers to the application of rule $a$ to the join interaction of $p$ and $s$. Similarly, the later refers to the application of rule $b$ to $w$. From a graphical point of view, each cause corresponds to one of proof trees in the Figure 1, the right hand side operator of application corresponds to the head whereas the left hand side operator corresponds to the product of its children.

The rest of the paper is organised as follows. Section 2 describes the algebra with these three operations and a quite natural ordering relation on causes. The next section studies the semantics for positive logic programs and shows the correspondence between the syntactic proof tree of a standard (non-causal) logic program and the interpretation of each atom in a *causal model*. Section 4 introduces default negation and stable models. Finally, Section 5 concludes the paper.

## 2 Algebra of causal values

As we have introduced, our set of *causal values* will constitute an algebra with three internal operations: addition '$+$' representing alternative causes, product '$*$' representing joint interaction between causes and rule application '$\cdot$'. We define now *causal terms*, the syntactic counterpart of (causal) values, just as combinations of these three operations over labels (events).

**Definition 1 (Causal term).** *A* causal term*, $t$, over a set of labels $Lb$ is recursively defined as one of the following expressions:*

$$t ::= l \mid \prod_{t_i \in S} t_i \mid \sum_{t_i \in S} t_i \mid t_1 \cdot t_2$$

*where $l$ is a label $l \in Lb$, $t_1, t_2$ are in their turn causal terms and $S$ is a (possibly empty or possibly infinite) set of causal terms. The set of causal terms over $Lb$ is denoted by $\mathcal{T}_{Lb}$.* □

As we can see, infinite products and sums are allowed whereas a term may only contain a finite number of concatenation applications. Constants $0$ and $1$ will be shorthands for the empty sum $\sum_{t \in \emptyset} t$ and the empty product $\prod_{t \in \emptyset} t$, respectively.

We adopt the following notation. To avoid an excessive use of parentheses, we assume that '$\cdot$' has the highest priority, followed by '$*$' and '$+$' as usual, and we further note that the three operations will be associative. When clear from the context, we will sometimes remove '$\cdot$' so that, for instance, the term $l_1 l_2$ stands for $l_1 \cdot l_2$. As we will see, two (syntactically) different causal terms may correspond to the same causal value. However, we will impose Unique Names Assumption (UNA) for labels, that is, $l \neq l'$ for any two (syntactically) different labels $l, l' \in Lb$, and similarly $l \neq 0$ and $l \neq 1$ for any label $l$.

To fix properties of our algebra we recall that addition '$+$' represents a set of alternative causes and product '$*$' a set of causes that are jointly used. Thus, since both represent sets, they are associative, commutative and idempotent. Contrary, although associative, application '$\cdot$' is not commutative. Note that the right hand side operator represents the applied rule and left hand one represents a cause that is necessary to apply it, therefore they are clearly not interchangeable. We can note another interesting property: application '$\cdot$' distributes over both addition '$+$' and product '$*$'. To illustrate this idea, consider the following variation of our example. Suppose now that the boat also leaves a wake behind when it moves forward. Let $\Pi_3$ be the set of rules $\Pi_1$ plus the rule $k : wake \leftarrow fwd$ reflecting this new assumption. As we saw, $fwd$ is justified by $p * s + w$ and thus $wake$ will be justified by applying rule $k : wake \leftarrow fwd$ to it, i.e.the value $(p * s + w) \cdot k$. We can also see that there are two alternative causes justifying $wake$, graphically represented in the Figure 2. The term that corresponds which this graphical representation is $(p * s) \cdot k + w \cdot k = (p * s + w) \cdot k$. Moreover, application '$\cdot$' also distributes over product '$*$' and $(p * s) \cdot k + w \cdot k$ is equivalent to $(p \cdot k) * (s \cdot k) + (w \cdot k)$. Intuitively, if the joint iteration of $p$ and $s$ is necessary to apply $k$ then both $p$ and $s$ are also necessary to apply it, and conversely. Note that each chain of applications , $(p \cdot k)$, $(s \cdot k)$ and $(w \cdot k)$ corresponds to a path in one of the trees in the Figure 2. Causes can be seen as sets (products) of paths (causal chains).

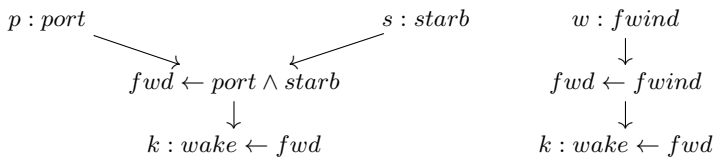



**Fig. 2.** Proof trees pontificating atom $fwd$ in the program $\Pi_3$

**Definition 2 (Causal Chain).** *A* causal chain $x$ *over a set of labels* $Lb$ *is a sequence* $x = l_1 \cdot l_2 \cdot \ldots \cdot l_n$*, or simply* $l_1 l_2 \ldots l_n$*, with length* $|x| = n > 0$ *and* $l_i \in Lb$*.* □

We denote $\mathcal{X}_{Lb}$ to stand for the set of causal chains over $Lb$ and will use letters $x, y, z$ to denote elements from that set. It suffices to have a non-empty set of labels, say $Lb = \{a\}$, to get an infinite set of chains $\mathcal{X}_{Lb} = \{a, aa, aaa, \ldots\}$, although all of them have a finite length. It is easy to see that, by an exhaustive application of distributivity, we can "shift" inside all occurrences of the application operator so that it only occurs in the scope of other application operators. A causal term obtained in this way is a *normal causal term*.

**Definition 3 (Normal causal term).** *A* causal term*,* $t$*, over a set of labels* $Lb$ *is recursively defined as one of the following expressions:*

$$t ::= x \mid \prod_{t_i \in S} t_i \mid \sum_{t_i \in S} t_i$$

*where* $x \in \mathcal{X}_{Lb}$ *is a causal chain over* $Lb$ *and* $S$ *is a (possibly empty or possibly infinite) set of normal causal terms. The set of causal terms over* $Lb$ *is denoted by* $\mathcal{U}_{Lb}$*.* □

**Proposition 1.** *Every causal term* $t$ *can be normalized, i.e. written as an equivalent normal causal term* $u$*.* □

In the same way as application '$\cdot$' distributes over addition '$+$' and product '$*$', the latter, in their turn, also distributes over addition '$+$'. Consider a new variation of our example to illustrate this fact. Suppose that we have now two port rowers that can strike, encoded as the set of rules $\Pi_4$:

$$p_1 : port_1 \quad p_2 : port_2 \quad s : starb$$
$$port \leftarrow port_1 \qquad port \leftarrow port_2 \qquad fwd \leftarrow port \wedge starb$$

We can see that, in the only causal stable model of this program, atom $port$ was justified by two alternative, and independent causes, $p_1$ and $p_2$, and after applying unlabelled rules to them, the resulting value assigned to $fwd$ is $(p_1 + p_2) * s$. It is also clear that there are two alternative causes justifying $fwd$: the result from combining the starboard rower strike with each of the port rower strikes, $p_1 * s$ and $p_2 * s$. That is, causal terms $(p_1 + p_2) * s$ and $p_1 * s + p_2 * s$ are equivalent.

Furthermore, as we introduce above, causes can be ordered by a notion of "strength" of justification. For instance, in our example, $fwd$ is justified by two independent

causes, $p * s + w$ while $fwind$ is only justified by $w$. If we consider the program $\Pi_5$ obtained by removing the fact $w : fwind$ from $\Pi_1$ then $fwd$ keeps being justified by $p * s$ but $fwind$ becomes false. That is, $fwd$ is "more strongly justified" than $fwind$ in $\Pi_1$, written $w \leq p * s + w$. Similarly, $p * s \leq p * s + w$. Note also that, in this program $\Pi_5$, $fwd$ needs the joint interaction of $p$ and $s$ to be justified but $port$ and $starb$ only need $p$ and $s$, respectively. That is, $p$ is "more strongly justified" than $p * s$, written $p * s \leq p$. Similarly, $p * s \leq s$. We can also see that in program $\Pi_2$ which labels rules for $fwd$, one of the alternative causes for $fwd$ is $w \cdot b$ and this is "less strongly justified" than $w$, i.e. $w \cdot b \leq w$ since, from a similar reasoning, $w \cdot b$ needs the application of $b$ to $w$ when $w$ only requires itself. In general, we will see that $a \cdot b \leq a * b \leq X \leq a + b$ where $X$ can be either $a$ or $b$. We formalize this order relation starting for causal chains. Notice that a causal chain $x = l_1 l_2 \dots l_n$ can be alternatively characterized as a partial function from naturals to labels $x : \mathbb{N} \longrightarrow Lb$ where $x(i) = l_i$ for all $i \leq n$ and undefined for $i > n$. Using this characterisation, we can define the following partial order among causal chains:

**Definition 4 (Chain subsumption).** *Given two causal chains $x$ and $y \in \mathcal{X}_{Lb}$, we say that $y$* subsumes *$x$, written $x \leq y$, if and only if there exists a strictly increasing function $\delta : \mathbb{N} \longrightarrow \mathbb{N}$ such that for each $i \in \mathbb{N}$ with $y(i)$ defined, $x\big(\delta(i)\big) = y(i)$.* □

**Proposition 2.** *Given two finite causal chains $x, y \in \mathcal{X}_{Lb}$, they are equivalent (i.e. both $x \leq y$ and $y \leq x$) if and only if they are syntactically identical.* □

Informally speaking, $y$ subsumes $x$, when we can embed $y$ into $x$, or alternatively when we can form $y$ by removing (or skipping) some labels from $x$. For instance, take the causal chains $x = abcde$ and $y = ac$. Clearly we can form $y = ac = a \cdot \not{b} \cdot c \cdot \not{d} \cdot \not{e}$ by removing $b$, $d$ and $e$ from $x$. Formally, $x \leq y$ because we can take some strictly increasing function with $\delta(1) = 1$ and $\delta(2) = 3$ so that $y(1) = x(\delta(1)) = x(1) = a$ and $y(2) = x(\delta(2)) = x(3) = c$.

Although, at a first sight, it may seem counterintuitive the fact that $x \leq y$ implies $|x| \geq |y|$, as we mentioned, a fact or formula is "more strongly justified" when we need to apply less rules to derive it (and so, causal chains contain less labels) respecting their ordering. In this way, chain $ac$ is a "more strongly justification" than $abcde$.

As we saw above, a cause can be seen as a product of causal chains, that from a graphical point of view correspond to the set of paths in a proof tree. We notice now an interesting property relating causes and the "more strongly justified" order relation: a joint interaction of comparable causal chains should collapse to the weakest among them. Take, for instance, a set of rules $\Pi_6$:

$$a : p \qquad b : q \leftarrow p \qquad r \leftarrow p \wedge q$$

where, in the unique causal stable model, $r$ corresponds to the value $a * a \cdot b$. Informally we can read this as "we need $a$ and apply rule $b$ to rule $a$ to prove $r$". Clearly, we are repeating that we need $a$. Term $a$ is redundant and then $a * a \cdot b$ is simply equivalent to $a \cdot b$. This idea is quite related to the definition of *order filter* in order theory. An *order filter* $F$ of a poset $P$ is a special subset $F \subseteq P$ satisfying[1] that for any $x \in F$ and $y \in P$, $x \leq y$

[1] *Order filter* is a weaker notion than *filter* which further satisfies that any pair $x, y \in F$ has a lower bound in $F$ too.

implies $y \in F$. An order filter $F$ is furthermore *generated* by an element $x \in P$ iff $x \leq y$ for all elements $y \in F$, the order filter generated by $x$ is written $||x||$. Considering causes as the union of filters generated by their causal chains, the join interaction of causes just correspond to their union. For instance, if we consider the set of labels $Lb = \{a, b\}$ and its corresponding set of causal chains $\mathcal{X}_{Lb} = \{a, b, ab, ba, \dots\}$, then $||ab||$ and $||a||$ respectively correspond to the set of all chains grater than $ab$ and $a$ in the poset $P = \langle \mathcal{X}_{Lb}, \leq \rangle$. Those are, $||ab|| = \{ab, a, b\}$ and $||a|| = \{a\}$. The term $a * ab$ corresponds just to the union of both sets $||a|| \cup ||ab|| = ||ab||$. We define a cause as follows:

**Definition 5 (Cause).** *A* cause *for a set of labels $Lb$ is any order filter for the poset of chains $\langle \mathcal{X}_{Lb}, \leq \rangle$. We will write $\mathcal{C}_{Lb}$ (or simply $\mathcal{C}$ when there is no ambiguity) to denote the set of all causes for $Lb$.* □

This definition captures the notion of cause, or syntactically a product of causal chains. To capture possible alternative causes, that is, additions of products of causal chains, we notice that addition obeys a similar behaviour with respect to redundant causes. Take, for instance, a set of rules $\Pi_7$:

$$a : p \qquad b : p \leftarrow p$$

It is clear, that the cause $a$ is sufficient to justify $p$, but there are also infinitely many other alternative and redundant causes $a \cdot b$, $a \cdot b \cdot b$, ... that justify $p$, that is $a + a \cdot b + a \cdot b \cdot b + \dots$. To capture a set of alternative causes we define the idea of causal value, in its turn, as a filter of causes.

**Definition 6 (Causal Value).** *Given a set of labels $Lb$, a* causal value *is any order filter for the poset $\langle \mathcal{C}_{Lb}, \subseteq \rangle$.* □

The causal value $|||a|||$, the filter generated by the cause $||a||$, is the set containing $||a|| = \{a, a + b\}$ and all its supersets. That is, $|||a||| = \{||a||, ||a * b||, ||a \cdot b||, \dots\}$. Futhermore, as we will se later, addition can be interpreted as the union of causal values for its respective operands. Thus, $a + a \cdot b + a \cdot b \cdot b + \dots$ just corresponds to the union of the causal values generated by their addend causes, $|||a||| \cup |||a \cdot b||| \cup |||a \cdot b \cdot b||| + \dots = |||a|||$.

The set of possible causal values formed with labels $Lb$ is denoted as $\mathcal{V}_{Lb}$. An element from $\mathcal{V}_{Lb}$ has the form of a set of sets of causal chains that, intuitively, corresponds to a set of alternative causes (*sum of products of chains*). From a graphical point of view, it corresponds to a set of alternative proof trees represented as their respective sets of paths. We define now the correspondence between syntactical causal terms and their semantic counterpart, causal values.

**Definition 7 (Valuation of normal terms).** *The* valuation *of a normal term is a mapping $\epsilon : \mathcal{U}_{Lb} \longrightarrow \mathcal{V}_{Lb}$ defined as:*

$$\epsilon(x) \stackrel{\text{def}}{=} |||x||| \text{ with } x \in \mathcal{X}_{Lb}, \;\; \epsilon\Big(\sum_{t \in S} t\Big) \stackrel{\text{def}}{=} \bigcup_{t \in S} \epsilon(t), \;\; \epsilon\Big(\prod_{t \in S} t\Big) \stackrel{\text{def}}{=} \bigcap_{t \in S} \epsilon(t) \qquad \square$$

Note that any causal term can be normalized and then this definition trivially extends to any causal term. Furthermore, a causal chain $x$ is mapped just to the causal value generated by the cause, in their turn, generated by $x$, i.e. the set containing all causes which contain $x$. The aggregate union of an empty set of sets (causal values) corresponds to $\emptyset$. Therefore $\epsilon(0) = \bigcup_{t\in\emptyset} \epsilon(t) = \emptyset$, i.e. 0 just corresponds to the absence of justification. Similarly, as causal values range over parts of $\mathcal{C}$, the aggregate intersection of an empty set of causal values corresponds to $\mathcal{C}$, and thus $\epsilon(1) = \bigcap_{t\in\emptyset} \epsilon(t) = \mathcal{C}$, i.e. 1 just corresponds to the "maximal" justification.

**Theorem 1 (From [4]).** *$\langle \mathcal{V}_{Lb}, \cup, \cap\rangle$ is the free completely distributive lattice generated by $\langle \mathcal{X}_{Lb}, \leq\rangle$, and the restriction of $\epsilon$ to $\mathcal{X}_{Lb}$ is an injective homomorphism (or embedding).* □

The above theorem means that causal terms form a complete lattice. The order relation $\leq$ between causal terms just corresponds to set inclusion between their corresponding causal values, i.e. $x \leq y$ iff $\epsilon(x) \subseteq \epsilon(y)$. Furthermore, addition '$+$' and product '$*$' just respectively correspond to the least upper bound and the greater lower bound of the associated lattice $\langle \mathcal{T}_{Lb}, \leq\rangle$ or $\langle \mathcal{T}_{Lb}, +, *\rangle$ where:

$$t \leq u \stackrel{\text{def}}{=} \epsilon(t) \subseteq \epsilon(u) \qquad (\Leftrightarrow \;\; t * u = t \;\; \Leftrightarrow \;\; t + u = u\,)$$

for any normal term $t$ and $u$. For instance, in our example $\Pi_2$, $fwd$ was associated with the causal term $p \cdot a * s \cdot a + w \cdot b$. Thus, the causal value associated with it corresponds to

$$\epsilon(p \cdot a * s \cdot a + w \cdot b) = \{\!|\!|\!|p \cdot a|\!|\!|\} \cap \{\!|\!|\!|s \cdot a|\!|\!|\} \cup \{\!|\!|\!|w \cdot b|\!|\!|\}$$

Causal values are, in general, infinite sets. For instance, as we saw before, simply with $Lb = \{a\}$ we have the chains $\mathcal{X}_{Lb} = \{a, aa, aaa, \dots\}$ and $\epsilon(a)$ contains all possible causes in $\mathcal{C}$ that are supersets of $\{a\}$, that is, $\epsilon(a) = \{\{a\}, \{aa, a\}, \{aaa, aa, a\}, \dots\}$. Obviously, writing causal values in this way is infeasible – it is more convenient to use a representative causal term instead. For this purpose, we define a function $\gamma$ that acts as a right inverse morphism for $\epsilon$ selecting minimal causes, i.e., given a causal value $V$, it defines a normal term $\gamma(V) = t$ such that $\epsilon(t) = V$ and $\gamma(V)$ does not have redundant subterms. The function $\gamma$ is defined as a mapping $\gamma : \mathcal{V}_{Lb} \longrightarrow \mathcal{U}_{Lb}$ such that for any causal value $V \in \mathcal{V}_{Lb}$, $\gamma(V) \stackrel{\text{def}}{=} \sum_{C\in\underline{V}} \prod_{x\in\underline{C}} x$ where $\underline{V} = \{C \in V \mid \nexists D \in V, D \subset C\}$ and $\underline{C} = \{x \in C \mid \nexists y \in C, y < x\}$ respectively stand for $\subseteq$-minimal causes of $V$ and $\leq$-minimal chains of $C$. We will use $\gamma(V)$ to represent $V$.

**Proposition 3.** *The mapping $\gamma$ is a right inverse morphism of $\epsilon$.* □

Given a term $t$ we define its *canonical form* as $\gamma(\epsilon(t))$. Canonical terms are of the form of sums of products of causal chains. As it can be imagined, not any term in that form is a canonical term. For instance, going back, we easily can check that terms $a * ab = ab$ and $a + ab + abb + \cdots = a$ respectively correspond to the canonical terms $\gamma(\epsilon(ab * a)) = \gamma(\epsilon(ab)) = ab$ and $\gamma(\epsilon(a + ab + abb + \dots)) = \gamma(\epsilon(a)) = a$. Figure 3 summarizes addition and product properties while Figure 4 is analogous for application properties.

| *Associativity* | *Commutativity* | *Absorption* |
|---|---|---|
| $t + (u+w) = (t+u) + w$ | $t + u = u + t$ | $t = t + (t*u)$ |
| $t * (u*w) = (t*u) * w$ | $t * u = u * t$ | $t = t * (t+u)$ |

| *Distributive* | *Identity* | *Idempotence* | *Annihilator* |
|---|---|---|---|
| $t + (u*w) = (t+u) * (t+w)$ | $t = t + 0$ | $t = t + t$ | $1 = 1 + t$ |
| $t * (u+w) = (t*u) + (t*w)$ | $t = t * 1$ | $t = t * t$ | $0 = 0 * t$ |

**Fig. 3.** Sum and product satisfy the properties of a completely distributive lattice.

| *Associativity* | *Absorption* | *Identity* |
|---|---|---|
| $t \cdot (u \cdot w) = (t \cdot u) \cdot w$ | $t = t + u \cdot t \cdot w$ | $t = 1 \cdot t$ |
| | $u \cdot t \cdot w = t * u \cdot t \cdot w$ | $t = t \cdot 1$ |

| *Addition distributivity* | *Product distributivity* | *Annihilator* |
|---|---|---|
| $t \cdot (u+w) = (t \cdot u) + (t \cdot w)$ | $c \cdot (d * e) = (c \cdot d) * (c \cdot e)$ | $0 = t \cdot 0$ |
| $(t+u) \cdot w = (t \cdot w) + (u \cdot w)$ | $(c * d) \cdot e = (c \cdot e) * (c \cdot e)$ | $0 = 0 \cdot t$ |

**Fig. 4.** Properties of the application '$\cdot$' operator. Note: $c$, $d$ and $e$ denote a causes instead of arbitrary causal terms.

For practical purposes, simplification of causal terms can be done by applying the algebraic properties shown in Figures 3 and 4. For instance, the examples from $\Pi_6$ and $\Pi_7$ containing redundant information can now be derived as follows:

$$\begin{aligned}
a * a \cdot b &= (a * 1 \cdot a \cdot b) && \text{identity for '}\cdot\text{'} \\
&= 1 \cdot a \cdot b && \text{absorption for '}\cdot\text{'} \\
&= a \cdot b && \text{identity for '}\cdot\text{'}
\end{aligned}$$

$$\begin{aligned}
a + a \cdot b + a \cdot b \cdot b + \ldots &= a + 1 \cdot a \cdot b + a \cdot b \cdot b + \ldots && \text{identity for '}\cdot\text{'} \\
&= a + a \cdot b \cdot b + \ldots && \text{absorption for '}\cdot\text{'} \\
&= a + 1 \cdot a \cdot b \cdot b + \ldots && \text{identity for '}\cdot\text{'} \\
&\ldots && \ldots \\
&= a && \text{absorption for '}\cdot\text{'}
\end{aligned}$$

Let us see another example involving distributivity. The term $ab * c + a$ can be derived as follows:

$$\begin{aligned}
a \cdot b * c + a &= (a \cdot b + a) * (c + a) && \text{distributivity} \\
&= (1 \cdot a \cdot b + a) * (c + a) && \text{identity for '}\cdot\text{'} \\
&= (a + 1 \cdot a \cdot b) * (c + a) && \text{commutativity for '}+\text{'} \\
&= a * (c + a) && \text{absorption for '}\cdot\text{'} \\
&= a && \text{absorption for '}*\text{'}
\end{aligned}$$

## 3 Positive programs and minimal models

Let us describe now how to use the causal algebra to evaluate causal logic programs. A *signature* is a pair $\langle At, Lb\rangle$ of sets that respectively represent the set of *atoms* (or *propositions*) and the set of labels. As usual, a *literal* is defined as an atom $p$ (positive literal) or its negation $\neg p$ (negative literal). In this paper, we will concentrate on programs without disjunction in the head, leaving the treatment of disjunction for a future study.

**Definition 8 (Causal logic program).** *Given a signature $\langle At, Lb\rangle$ a* (causal) logic program *$\Pi$ is a set of rules of the form:*

$$t : L_0 \leftarrow L_1 \wedge \ldots \wedge L_m \wedge \text{not } L_{m+1} \wedge \ldots \wedge \text{not } L_n$$

*where $t$ is a causal term over $Lb$, $L_0$ is a literal or $\bot$ (the* head *of the rule) and $L_1 \wedge \ldots \wedge L_m \wedge$ not $L_{m+1} \wedge \ldots$ not $L_n$ is a conjunction of literals (the* body *of the rule). An empty body is represented as $\top$.* □

For any rule $\phi$ of the form $t : L_0 \leftarrow L_1 \wedge \ldots \wedge L_m \wedge$ *not* $L_{m+1} \wedge \ldots$ *not* $L_n$ we define $label(\phi) = t$. Most of the following definitions are standard in logic programming. We denote $head(\phi) = L_0$, $B^+$ (resp. $B^-$) to represent the conjunction of all positive (resp. negative) literals $L_1 \wedge \ldots \wedge L_n$ (resp. *not* $L_{m+1} \wedge \ldots \wedge$ *not* $L_n$) that occur in $B$. A logic program is *positive* if $B^-$ is empty for all rules ($n = m$), that is, if it contains no negations. Unlabelled rules are assumed to be labelled with the element 1 which, as we saw, is the identity for application '$\cdot$'. $\top$ (resp. $\bot$) represent truth (resp. falsity). If $n = m = 0$ then $\leftarrow$ can be dropped.

Given a signature $\langle At, Lb\rangle$ a *causal interpretation* is a mapping $I : At \longrightarrow \mathcal{V}_{Lb}$ assigning a causal value to each atom. Partial order $\leq$ is extended over interpretations so that given two interpretations $I, J$ we define $I \leq J \stackrel{\text{def}}{=} I(p) \leq J(p)$ for each atom $p \in At$. There is a $\leq$-bottom interpretation $\mathbf{0}$ (resp. a $\leq$-top interpretation $\mathbf{1}$) that stands for the interpretation mapping each atom $p$ to 0 (resp. 1). The set of interpretations $\mathcal{I}$ with the partial order $\leq$ forms a poset $\langle \mathcal{I}, \leq\rangle$ with supremum '$+$' and infimum '$*$' that are respectively the sum and product of atom interpretations. As a result, $\langle \mathcal{I}, +, *\rangle$ also forms a complete lattice.

**Observation 1** *When $Lb = \emptyset$ the set of causal values becomes $\mathcal{V}_{Lb} = \{0, 1\}$ and interpretations collapse to classical propositional logic interpretations.* □

**Definition 9 (Causal model).** *Given a positive causal logic program $\Pi$ and a causal interpretation $I$ over the signature $\langle At, Lb\rangle$, $I$ is a causal model, written $I \models \Pi$, if and only if*

$$\big(I(L_1) * \ldots * I(L_m)\big) \cdot t \leq I(L_0)$$

*for each rule $\varphi \in \Pi$ of the form $\varphi = L_0 \leftarrow L_1, \ldots, L_m$.*

For instance, take rule (1) from Example 1 and let $I$ be an interpretation such that $I(port) = p$ and $I(starb) = s$. Then $I$ will be a model of (1) when $(p * s) \cdot a \leq$

$I(fwd)$. In particular, this holds when $I(fwd) = (p * s) \cdot a + w \cdot b$ which was the value we expected for program $\Pi_2$. But it would also hold when, for instance, $I(fwd) = a + b$ or $I(fwd) = 1$. Note that this is important if we had to accommodate other possible additional facts $(a : fwd)$ or even $(1 : fwd)$ in the program. The fact that any $I(fwd)$ greater than $(p * s) \cdot a + w \cdot b$ is also a model clearly points out the need for selecting minimal models. In fact, as happens in the case of non-causal programs, positive programs have a least model (this time, with respect to $\leq$ relation among causal interpretations) that can be computed by iterating an extension of the well-known *direct consequences operator* defined by [5].

**Definition 10 (Direct consequences).** *Given a positive logic program $\Pi$ for signature $\langle At, Lb\rangle$ and a causal interpretation $I$, the operator of* direct consequences *is a function $T_\Pi : \mathcal{I} \longrightarrow \mathcal{I}$ such that, for any atom $p \in At$:*

$$T_\Pi(I)(L_0) \stackrel{\text{def}}{=} \sum \left\{ \big(I(L_1) * \ldots * I(L_m)\big) \cdot t \;\mid\; (t : L_0 \leftarrow L_1 \wedge \ldots \wedge L_m) \in \Pi \right\}$$

In order to prove some properties of this operator, an important observation should be made: since the set of causal values forms now a lattice, causal logic programs can be translated to *Generalized Annotated Logic Programming* (GAP). GAP is a general a framework for multivalued logic programming where the set of truth values must to form an upper semilattice and rules (*annotated clauses*) have the following form:

$$L_0 : \rho \leftarrow L_1 : \mu_1 \;\&\; \ldots \;\&\; L_m : \mu_m \tag{3}$$

where $L_0, \ldots, L_m$ are literals, $\rho$ is an *annotation* (may be just a truth value, an *annotation variable* or a *complex annotation*) and $\mu_1, \ldots, \mu_m$ are values or annotation variables. A complex annotation is the result to apply a total continuous function to a tuple of annotations. For instance $\rho$ can be a complex annotation $f(\mu_1, \ldots, \mu_m)$ that applies the function $f$ to a m-tuple $(\mu_1, \ldots, \mu_m)$ of annotation variables in the body of (3). Given a positive program $\Pi$, each rule $\varphi \in \Pi$ of the form

$$t : L_0 \leftarrow L_1 \wedge \ldots \wedge L_m \tag{4}$$

is translated to an annotated clause $GAP(\varphi)$ of the form of (3) where $\mu_1, \ldots, \mu_m$ are annotation variables that capture the causal values of each body literal. The head complex annotation corresponds to the function $\rho \stackrel{\text{def}}{=} (\mu_1 * \ldots * \mu_m) \cdot t$ that forces the head to inherit the causal value obtained by applying the rule label $t$ to the product of the interpretation of body literals $\mu_1 * \ldots * \mu_m$. The translation of a program $\Pi$ is simply defined as:

$$GAP(\Pi) \stackrel{\text{def}}{=} \{GAP(\varphi) \mid \varphi \in \Pi\}$$

A complete description of GAP restricted semantics, denoted as $\models^r$, is out of the scope of this paper (the reader is referred to [6]). For our purposes, it suffices to observe that the following important property is satisfied.

**Theorem 2.** *A positive causal logic program $\Pi$ can be translated to a general annotated logic program $GAP(\Pi)$ s.t. a causal interpretation $I \models \Pi$ if and only if $I \models^r GAP(\Pi)$. Furthermore, $T_\Pi(I) = R_{GAP(\Pi)}(I)$ for any interpretation $I$.*

**Corollary 1.** *Given a positive logic program $\Pi$ the following properties hold:*

1. *Operator $T_\Pi$ is monotonic.*
2. *Operator $T_\Pi$ is continuous.*
3. *$T_\Pi \uparrow^\omega (\mathbf{0}) = lfp(T_\Pi)$ is the least model of $\Pi$.*
4. *The iterative computation $T_\Pi \uparrow^k (\mathbf{0})$ reaches the least fixpoint in $n$ steps for some positive integer $n$.*

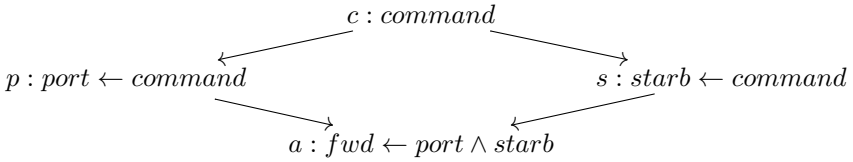
*Proof.* Directly follows from Theorem 2 and Theorems 1, 2 and 3 in [6].

The existence of a least model for a positive program and its computation using $T_\Pi$ is an interesting result, but it does not provide any information on the relation between the causal value it assigns to each atom with respect to its role in the program. As we will see, we can establish a direct relation between this causal value and the idea of *proof* in the positive program. Let us formalise next the idea of proof tree.

**Definition 11 (Proof tree).** *Given a causal logic program $\Pi$, a* proof tree *is a directed acyclic graph $T = \langle V, E\rangle$, where vertices $V \subseteq \Pi$ are rules from the program, and $E \subseteq V \times V$ satisfying:*

*(i) There is at exactly one vertex without outgoing edges denoted as $sink(T)$.*
*(ii) For each rule $\varphi = (t : L_0 \leftarrow B) \in V$ and for each atom $L_i \in B^+$ there is exactly one $\varphi'$ with $(\varphi', \varphi) \in E$ and this rule satisfies $head(\varphi') = L_i$.* □

Notice that condition (ii) forces us to include an incoming edge for each atom in the positive body of a vertex rule. As a result, source vertices must be rules with empty positive body, or just facts in the case of positive programs. Another interesting observation is that, proof *tree* do not require an unique parent for each vertex. For instance, in Example 1, if both $port$ and $starb$ were obtained as a consequence of some command made by the captain, we could get instead a proof tree, call it $T_1$, of the form:




**Definition 12 (Proof path).** *Given a proof tree $T = \langle V, E\rangle$ we define a* proof path *for $T$ as a concatenation of terms $t_1 \ldots t_n$ satisfying:*

1. *There exists a rule $\varphi \in V$ with $label(r) = t_1$ such that $\varphi$ is a source, that is, there is no $\varphi'$ s.t. $(\varphi', \varphi) \in E$.*
2. *For each pair of consecutive terms $t_i, t_{i+1}$ in the sequence, there is some edge $(\varphi_i, \varphi_{i+1}) \in E$ s.t. $label(\varphi_i) = t_i$ and $label(\varphi_{i+1}) = t_{i+1}$.*
3. *$label(sink(T)) = t_n$.* □

Let us write $Paths(T)$ to stand for the set of all proof paths for a given proof tree $T$. We define the cause associated to any tree $T = \langle V, E\rangle$ as the causal term $cause(T) \stackrel{\text{def}}{=} \prod_{t \in Paths(T)} t$. As an example, $cause(T_1) = (c \cdot p \cdot a) * (c \cdot s \cdot a)$. Also $(p \cdot a) * (s \cdot a)$ and $w \cdot b$ correspond to each tree in Figure 1.

**Theorem 3.** *Let $\Pi$ be a positive program and $I$ be the least model of $\Pi$, then for each atom $p$:*

$$I(p) = \sum_{T \in PT_p} cause(T)$$

*where $PT_p = \{T = \langle V, E\rangle \mid head(sink(T)) = p\}$ is a set of proof trees with nodes $V \subseteq \Pi$.*

From this result, it may seem that our semantics is just a direct translation of the syntactic idea of proof trees. However, the semantics is actually a more powerful notion that allows detecting redundancies, tautologies and inconsistencies. In fact, the expression $\sum_{T \in PT_p} cause(T)$ may contain redundancies and is not, in the general case, in normal form. As an example, recall the program $\Pi_6$:

$$a : p \qquad b : q \leftarrow p \qquad r \leftarrow p \wedge q$$

that has only one proof tree for $p$ whose cause would correspond to $I(r) = a * a \cdot b$. But, by absorption, this is equivalent to $I(r) = a \cdot b$ pointing out that the presence of $p$ in rule $r \leftarrow p \wedge q$ is redundant.

A corollary of Theorem 3 is that we can replace a rule label by a different one, or by 1 (the identity for application '$\cdot$') and we get the same least model, modulo the same replacement in the causal values for all atoms.

**Corollary 2.** *Let $\Pi$ be a positive program, $I$ the least model of $\Pi$, $l \in Lb$ be a label, $m \in Lb \cup \{1\}$ and $\Pi^l_m$ (resp. $I^l_m$) be the program (resp. interpretation) obtained after replacing each occurrence of $l$ by $m$ in $\Pi$ (resp. in the interpretation of each atom in $I$). Then $I^l_m$ is the least model of $\Pi^l_m$.* □

In particular, replacing a label by $m = 1$ has the effect of removing it from the signature. Suppose we make this replacement for all atoms in $Lb$ and call the resulting program and least model $\Pi_1^{Lb}$ and $I_1^{Lb}$ respectively. Then $\Pi_1^{Lb}$ is just the non-causal program resulting from $\Pi$ after removing all labels and it is easy to see (Observation 1) that $I_1^{Lb}$ coincides with the least classical model of this program[2]. Moreover, this means that for any positive program $\Pi$, if $I$ is its least model, then the classical interpretation:

$$I'(p) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } I(p) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

is the least classical model of $\Pi$ ignoring its labels.

## 4 Default negation and stable models

Consider now the addition of negation, so that we deal with arbitrary programs. To illustrate this, we introduce a variation in Example 1 introducing the *qualification problem* from [7]: actions for moving the boat forward can be disqualified if an *abnormal*

[2] Note that $I^{Lb}$ is Boolean: if assigns either 0 or 1 to any atom in the signature.

*situation* occurs (for instance, that the boat is anchored, any of the oars are broken, the sail is full of holes, etc.) . As usual this can be represented using default negation as shown in the set of rules $\Pi_8$:

$$\begin{array}{cc} p : port \qquad s : starb & w : fwind \\ a : fwd \leftarrow port \wedge starb \wedge not\ ab\_a & b : fwd \leftarrow fwind \wedge not\ ab\_b \\ ab\_a \leftarrow anchored & ab\_b \leftarrow anchored \\ ab\_a \leftarrow broken\_oar1 & ab\_b \leftarrow holed\_sail \\ ab\_a \leftarrow broken\_oar2 & \ldots \end{array}$$

The causes that justify an atom should not be a list of not occurred abnormal situations. For instance, in program $\Pi_8$ where no abnormal situation occurs, the causal value that justify atom $fwd$ should be $(p \cdot a * s \cdot a) + (w \cdot b)$ as in the program $\Pi_2$ where abnormal situations are not included. References to the not occurred abnormal situations (*not* $anchored$, *not* $broken\_oar1\ldots$) are not mentioned. Default negation does not affect the causes justifying an atom when the default holds. Of course, when the default does not hold, for instance adding the fact $anchored$ to the above program, $fwd$ becomes false. Thus, we introduce the following straightforward rephrasing of the traditional program reduct [2].

**Definition 13 (Program reduct).** *The reduct of a program $\Pi$ with respect to an interpretation $I$, written $\Pi^I$ is the result of the following transformations on $\Pi$:*

1. *Removing all rules s.t. $I(B^-) = 0$*
2. *Removing all negative literals from the rest of rules.* □

A causal interpretation $I$ is a *causal stable model* of a causal program $\Pi$ if $I$ is the least model of $\Pi^I$. This definition allows us to extend Theorem 3 to normal programs in a direct way:

**Theorem 4 (Main theorem).** *Let $\Pi$ be a causal program and $I$ be causal stable model of $\Pi$, then for each atom $p$:*

$$I(p) = \sum_{T \in PT_p} cause(T) \qquad \textit{where}$$

$PT_p = \{T = \langle V, E\rangle \mid head(sink(T)) = p$ *and* $V \subseteq \{(t : q \leftarrow B) \in \Pi \mid I(B^-) \neq 0\}\}$. □

That is, the only difference now is that the set of proof trees $PT_p$ is formed with rules whose negative body is not false $I(B^-) \neq 0$ (that is, they would generate rules in the reduct).

**Corollary 3.** *Let $\Pi$ be a normal program, $I$ a causal stable model of $\Pi$, $l \in Lb$ be a label, $m \in Lb \cup \{1\}$ and $\Pi^l_m$ (resp. $I^l_m$) be the program (resp. interpretation) obtained after replacing every occurrence of $l$ by $m$ in $\Pi$ (resp. in the interpretation of each atom in $I$). Then $I^l_m$ is a causal stable model of $\Pi^l_m$.* □

As in the case of positive programs, replacing a label by $m = 1$ has the effect of removing it from the signature. Then, for any normal program $\Pi$, if $I$ is a causal stable model, then the classical interpretation:

$$I'(p) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } I(p) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

is a classical stable model of $\Pi$ ignoring its labels. It is easy to see that not only the above program $\Pi_8$ has an unique causal stable model that corresponds to:

$$\begin{array}{lll} I(port) & = p & I(ab\_f) & = 0 \\ I(starb) & = s & I(anchored) & = 0 \\ I(fwind) & = w & I(broken\_oar1) & = 0 \\ I(fwd) & = (p \cdot a * s \cdot a) + (w \cdot b) & \ldots & = 0 \end{array}$$

but also the program obtained from it ignoring the labels has an unique standard stable model $\{port, starb, fwind, fwd\}$ that corresponds to the atoms whose interpretations differ from 0 in the former.

## 5 Conclusions

In this paper we have provided a multi-valued semantics for normal logic programs whose truth values form a lattice of causal chains. A causal chain is nothing else but a concatenation of rule labels that reflects some sequence of rule applications. In this way, a model assigns to each true atom a value that contains justifications for its derivation from existing rules. We have further provided three basic operations on the lattice: an addition, that stands for alternative, independent justifications; a product, that represents joint interaction of causes; and a concatenation that acts as a chain constructor. We have shown that this lattice is completely distributive and provided a detailed description of the algebraic properties of its three operations.

A first important result is that, for positive programs, there exists a least model that coincides with the least fixpoint of a direct consequences operator, analogous to [5]. With this, we are able to prove a direct correspondence between the semantic values we obtain and the syntactic idea of proof tree. The main result of the paper generalises this correspondence for the case of stable models for normal programs.

Many open topics remain for future study. For instance, ongoing work is currently focused on implementation, complexity assessment, extension to disjunctive programs or introduction of strong negation. Regarding expressivity, an interesting topic is the introduction of new syntactic operators for inspecting causal information like checking the influence of a particular event or label in a conclusion, expressing necessary or sufficient causes, or even dealing with counterfactuals. Another interesting topic is removing the syntactic reduct definition in favour of some full logical treatment of default negation, as happens for (non-causal) stable models and their characterisation in terms of Equilibrium Logic [8]. This would surely simplify the quest for a necessary and sufficient condition for strong equivalence, following similar steps to [9]. It may also allow extending the definition of causal stable models to an arbitrary syntax and to the

first order case, where the use of variables in labels may also introduce new interesting features.

There are also other areas whose relations deserve to be formally studied. For instance, the introduction of a strong negation operator will immediate lead to a connection to Paraconsistency approaches. In particular, one of the main problems in the area of Paraconsistency is deciding which parts of the theory do not propagate or *depend* on an inconsistency. This decision, we hope, will be easier in the presence of causal justifications for each derived conclusion. A related area for which similar connections can be exploited is Belief Revision. In this case, causal information can help to decide which *relevant* part of a revised theory must be withdrawn in the presence of new information that would lead to an inconsistency if no changes are made. A third obvious related area is Debugging in Answer Set Programming, where we try to explain discrepancies between an expected result and the obtained stable models. In this field, there exists a pair of relevant approaches [10, 11] to whom we plan to compare. Finally, as potential applications, our main concern is designing a high level action language on top of causal logic programs with the purpose of modelling some typical scenarios from the literature on causality in Artificial Intelligence.

## References


1. Cabalar, P.: Causal logic programming. In Erdem, E., Lee, J., Lierler, Y., Pearce, D., eds.: Correct Reasoning. Volume 7265 of Lecture Notes in Computer Science., Springer (2012) 102–116
2. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R.A., Bowen, K.A., eds.: Logic Programming: Proc. of the Fifth International Conference and Symposium (Volume 2). MIT Press, Cambridge, MA (1988) 1070–1080
3. Artëmov, S.N.: Explicit provability and constructive semantics. Bulletin of Symbolic Logic **7**(1) (2001) 1–36
4. Stumme, G.: Free distributive completions of partial complete lattices. Order **14** (1997) 179–189
5. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM **23**(4) (1976) 733–742
6. Kifer, M., Subrahmanian, V.S.: Theory of generalized annotated logic programming and its applications. Journal of Logic Programming **12** (1992)
7. McCarthy, J.: Epistemological problems of artificial intelligence. In Reddy, R., ed.: IJCAI, William Kaufmann (1977) 1038–1044
8. Pearce, D.: Equilibrium logic. Ann. Math. Artif. Intell. **47**(1-2) (2006) 3–41
9. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Trans. Comput. Log. **2**(4) (2001) 526–541
10. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In Fox, D., Gomes, C.P., eds.: AAAI, AAAI Press (2008) 448–453
11. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. TPLP **9**(1) (2009) 1–56

# Query Answering in Object Oriented Knowledge Bases in Logic Programming: Description and Challenge for ASP

Vinay K. Chaudhri[1], Stijn Heymans[1], Michael Wessel[1], and Tran Cao Son[2]

[1] Artificial Intelligence Center, SRI International, Menlo Park, CA 94025, USA
[2] Computer Science Department, New Mexico State University, NM 88003, USA

**Abstract.** Research on developing efficient and scalable ASP solvers can substantially benefit by the availability of data sets to experiment with. `KB_Bio_101` contains knowledge from a biology textbook, has been developed as part of Project Halo, and has recently become available for research use. `KB_Bio_101` is one of the largest KBs available in ASP and the reasoning with it is undecidable in general. We give a description of this KB and ASP programs for a suite of queries that have been of practical interest. We explain why these queries pose significant practical challenges for the current ASP solvers.

## 1 Introduction

The `KB_Bio_101` represents knowledge from a textbook used for advanced high school and introductory college biology courses [19]. The KB was developed by SRI as part of their work for Project Halo[3] and contains a concept taxonomy for the whole textbook and detailed rules for 20 chapters of the textbook. SRI has tested the educational usefulness of this KB in the context of an intelligent textbook called *Inquire*[4].

The `KB_Bio_101` was originally developed using a knowledge representation and reasoning system called Knowledge Machine (KM) [9]. To express `KB_Bio_101` in answer set programming (ASP) required us to define a conceptual modeling layer called *Object Oriented Knowledge Base* or OOKB [6]. The goal of this paper is not to introduce OOKB as a more complete specification and analysis of formal properties of OOKBs are available elsewhere [6]. OOKB is of more general interest as it supports conceptual modeling primitives that are commonly found in description logic (DL) family of languages such as a facility to define classes and organize them into a hierarchy, define partitions, ability to define relations (also known as slots) and organize them into a relation hierarchy, support for domain, range and qualified number constraints, support for defining sufficient conditions of a class, and support for descriptive rules. The features in OOKB also overlap with the features of logic programming (LP) languages such as

[3] `http://www.projecthalo.com/`
[4] `http://www.aaaivideos.org/2012/inquire_intelligent_textbook/`

$\mathbb{FDNC}$ [11], *Datalog*$^{\pm}$ [5], and *ASP*$^{fs}$ [1] in its support for function symbols. It differs from these previous LP languages as well as from the DL systems in that the functions can be used to specify graph-structured objects which cannot be done in these other languages. The reasoning with OOKBs has been proven to be undecidable [6].

The approach taken in this paper fosters work on multi-paradigm problem solving in the following ways. First, it aims to give a declarative formalization of reasoning tasks that were originally implemented in KM which is a very different paradigm for reasoning as compared to ASP. Second, the conceptual modeling primitives considered here directly overlap with many description logics, thus, providing another example of integration between ASP with DLs.

The primary objective of this paper is to introduce`KB_Bio_101`as a valuable and data set and four queries of practical interest on this KB. These queries have been found extremely useful in the context of *Inquire*. This dataset presents an excellent opportunity for further development of ASP solvers for the following reasons.

- Recent developments in ASP suggest that it could potentially provide an ideal tool for large scale KBs. Yet, most of the KBs described in the literature are fairly small. `KB_Bio_101` provides a real-world ASP program that fits this bill.
- We note that `KB_Bio_101` contains rules with function symbols for which the grounding is infinite. A simple example is a KB consisting of a single class `person`, and a single relation `has-parent`, and a statement of the form "for each `person` there exists an instance of the `has-parent` relation between this person with another individual who is also a `person`". The skolemized versions of these statements require function symbols. An obvious first challenge that must be addressed is to develop suitable grounding techniques.
- Even though rules in `KB_Bio_101` follow a small number of axiom templates, the size of this KB indicates that this could be a non-trivial task for ASP solvers.
- The `KB_Bio_101` cannot be expressed in commonly available decidable DLs because it contains graph structured descriptions. Efficient reasoning with graph structures is an area of active recent research [15, 16], and since there exists an export of `KB_Bio_101` for DL systems also [7], it provides an ideal usecase to explore the relative effectiveness of DL reasoners vs ASP solvers on a common problem.
- The reasoning tasks of computing differences between two concepts and finding relationships between two individuals are computationally intensive tasks. The implementations of these tasks in *Inquire* rely on graph algorithms and trade completeness for efficiency. These tasks will present a tough challenges to ASP solvers.
- Last but not the least, we believe that the KB could entice the development and/or experimentation with new solvers for extended classes of logic programs (e.g., language with existential quantifiers or function symbols).

In addition to the challenges listed above, it will be possible to define multiple new challenges of increasing difficulty that can be used to motivate further research and development of ASP solvers.

## 2 Background: Logic Programming and OOKB

### 2.1 Logic Programming

A logic program $\Pi$ is a set of rules of the form

$$c \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{1}$$

where $0 \leq m \leq n$, each $a_i$ is a literal of a first order language and $not\ a_j$, $m < j \leq n$, is called a negation as failure literal (or naf-literal). $c$ can be a literal or omitted. A rule (program) is non-ground if it contains some variable; otherwise, it is a ground rule (program). When $n = 0$, the rule is called a *fact*. When $c$ is omitted, the rule is a *constraint*. Well-known notions such as substitution, the Herbrand universe $\mathcal{U}_\Pi$, and Herbrand base $\mathcal{B}_\Pi$ of a program $\Pi$ are defined as usual.
The semantics of a program is defined over ground programs. For a ground rule $r$ of the form (1), let $pos(r) = \{a_1, \ldots, a_m\}$ and $neg(r) = \{a_{m+1}, \ldots, a_n\}$. A set of ground literals $X$ is consistent if there exists no atom $a$ s.t. $\{a, \neg a\} \subseteq X$. A ground rule $r$ is *satisfied* by $X$ if (*i*) $neg(r) \cap X \neq \emptyset$; (*ii*) $pos(r) \setminus X \neq \emptyset$; or (*iii*) $c \in X$.
Let $\Pi$ be a ground program. For a consistent set of ground literals $S$, the *reduct* of $\Pi$ w.r.t. $S$, denoted by $\Pi^S$, is the program obtained from the set of all rules of $\Pi$ by deleting (**i**) each rule that has a naf-literal $not\ a$ in its body with $a \in S$, and (**ii**) all naf-literals in the bodies of the remaining rules. $S$ is an *answer set* of $\Pi$ [13] if it satisfies the following conditions: (**i**) If $\Pi$ does not contain any naf-literal then $S$ is the minimal set of ground literals satisfying all rules in $\Pi$; and (**ii**) If $\Pi$ contains some naf-literal then $S$ is an answer set of $\Pi$ if $S$ is the answer set of $\Pi^S$.
For a non-ground program $\Pi$, a set of literals in $\mathcal{B}_\Pi$ is an answer set of $\Pi$ if it is an answer set of $ground(\Pi)$ that is the set of all possible ground rules obtained from instantiating variables with terms in $\mathcal{U}_\Pi$. $\Pi$ is *consistent* if it has an answer set. $\Pi$ *entails* a ground literal $a$, $\Pi \models a$, if $a$ belongs to every answer set of $\Pi$.
For convenience in notation, we will make use of choice atoms as defined in [20] that can occur in a rule wherever a literal can. Answer sets of logic programs can be computed using answer set solvers (e.g., CLASP [12], **dlv** [8]).

### 2.2 Object-Oriented Knowledge Bases

We will now review the notion of an OOKB [6]. We note that an OOKB could be viewed as a logic program with function symbols and the language of OOKBs contains features that cannot be represented in previous investigated classes of function symbols such as $\mathbb{FDNC}$ [11], $Datalog^{\pm}$ [5], or $ASP^{fs}$ [1]. In essense, an OOKBs is a logic program consisting of the following components:

- *Taxonomic Knowledge:* This group of facts encodes the class hierarchy, the relation hierarchy, individual constants and their class membership. It contains ASP-atoms of the following form:

$$class(c) \quad (2)$$
$$individual(i) \quad (3)$$
$$subclass\_of(c_1, c_2) \quad (4)$$
$$disjoint(c_1, c_2) \quad (5)$$
$$instance\_of(i, c) \quad (6)$$
$$relation(r) \quad (7)$$
$$range(r, c) \quad (8)$$
$$domain(r, c) \quad (9)$$
$$subrelation\_of(r_1, r_2) \quad (10)$$
$$compose(r_1, r_2, r_3) \quad (11)$$
$$inverse(r_1, r_2) \quad (12)$$

  The predicate names are self-explanatory.
- *Descriptive statements:* Relationships between individuals are encoded in OOKB by descriptive statements of the form:

$$value(r, f(X), g(X)) \leftarrow instance\_of(X, c) \quad (13)$$
$$value(r, X, g(X)) \leftarrow instance\_of(X, c) \quad (14)$$

  where $f$ and $g$ are unary functions, called *Skolem functions*, such that $f \neq g$ and $c$ is a class. Axiom 13 (or 14) describes a relation value of individuals belonging to class $c$, encoded by the atom $value(r, f(X), g(X))$ (or $value(r, X, f(X))$. It states that for each individual $X$ in $c$, $f(X)$ (or $X$) is related to $g(X)$ via the relation $r$. An example use of axiom 14 is: Every Eukaryotic Cell has part a Nucleus, where *Eukaryotic Cell* and *Nucleus* are individuals from these two classes, and *has part* is a relationship between those individuals. It is required that if $f$ (or $g$) appears in (13) or (14), then the OOKB also contains the following rule

$$instance\_of(f(X), c_f) \leftarrow instance\_of(X, c) \quad \text{or} \quad (15)$$
$$instance\_of(g(X), c_g) \leftarrow instance\_of(X, c) \quad (16)$$

  which specify the class of which $f(X)$ (resp. $g(X)$) is a member. For example, if $f(X)$ represents a nucleus individual, then $c_f$ will be the class Nucleus.
- *Cardinality constraints on relations:* OOKB allows cardinality constraints on relations to be specified by statements of the following form:

$$constraint(t, f(X), r, d, n) \leftarrow instance\_of(X, c) \quad (17)$$

  where $r$ is a relation, $n$ is a non-negative integer, $d$ and $c$ are classes, and $t$ can either be *min*, *max*, or *exact*. This constraint states that for each instance $X$ of the class $c$, the set of values of relation $r$ restricted on $f(X)$—which must occur in a relation value literal $value(r, f(X), g(X))$ of $c$—has minimal (resp. maximal, exactly) $n$ values belonging to the class $d$. The head of (17) is called a constraint literal of $c$.
- *Sufficient conditions:* A *sufficient condition* of a class $c$ defines sufficient conditions for membership of that class based on the relation values and

constraints applicable to an instance:

$$instance\_of(X, c) \leftarrow Body(\boldsymbol{X}) \tag{18}$$

where $Body(\boldsymbol{X})$ is a conjunction of relation value literals, instance-of literals, constraint-literals of $c$, and $X$ is a variable occurring in the body of the rule.

- *(In)Equality between individual terms:* The rules in this group specify in/equality between terms, which are constructable from Skolem functions and the variable $X$ ($t_1$ and $t_2$), and have the followimg form:

$$eq(t_1, t_2) \leftarrow instance\_of(X, c) \tag{19}$$
$$neq(t_1, t_2) \leftarrow instance\_of(X, c) \tag{20}$$

- *Domain-independent axioms:* An OOKB also contains a set of domain-independent axioms $\Pi_R$ for inheritance reasoning, reasoning about the relation values of individuals (rules (25)—(27)), in/equality between terms (rules (28)—(40)), and enforcing constraints (rules (42)—(47)).

$$subclass\_of(C, B) \leftarrow subclass\_of(C, A), subclass\_of(A, B). \tag{21}$$
$$instance\_of(X, C) \leftarrow instance\_of(X, D), subclass\_of(D, C). \tag{22}$$
$$disjoint(C, D) \leftarrow disjoint(D, C). \tag{23}$$
$$\neg instance\_of(X, C) \leftarrow instance\_of(X, D), disjoint(D, C). \tag{24}$$
$$value(U, X, Z) \leftarrow compose(S, T, U), value(S, X, Y), value(T, Y, Z). \tag{25}$$
$$value(T, X, Y) \leftarrow subrelation\_of(S, T), value(S, X, Y). \tag{26}$$
$$value(T, Y, X) \leftarrow inverse(S, T), value(S, X, Y). \tag{27}$$
$$eq(X, Y) \leftarrow eq(Y, X) \tag{28}$$
$$eq(X, Z) \leftarrow eq(X, Y), eq(Y, Z), X \neq Z \tag{29}$$
$$\leftarrow eq(X, Y), neq(X, Y) \tag{30}$$
$$\{substitute(X, Y)\} \leftarrow eq(X, Y). \tag{31}$$
$$\leftarrow eq(X, Y), \{substitute(X, Z) : eq(X, Z)\}0, \tag{32}$$
$$\{substitute(Y, Z) : eq(Y, Z)\}0.$$
$$\leftarrow substitute(X, Y), substitute(X, Z), \tag{33}$$
$$X \neq Y, X \neq Z, Y \neq Z. \tag{34}$$
$$\leftarrow substitute(X, Y), X \neq Y, neq(X, Y). \tag{35}$$
$$substitute(Y, Z) \leftarrow substitute(X, Z), X \neq Z, eq(X, Y). \tag{36}$$
$$is\_substituted(X) \leftarrow substitute(X, Y), X \neq Y. \tag{37}$$
$$substitute(X, X) \leftarrow term(X), not\ is\_substituted(X). \tag{38}$$
$$term(X) \leftarrow value(S, X, Y). \tag{39}$$
$$term(Y) \leftarrow value(S, X, Y). \tag{40}$$
$$value_e(S, P, Q) \leftarrow value(S, X, Y), substitute(X, P), substitute(Y, Q). \tag{41}$$
$$\leftarrow value(S, X, Y), domain(S, C), not\ instance\_of(X, C). \tag{42}$$
$$\leftarrow value(S, X, Y), range(S, C), not\ instance\_of(Y, C). \tag{43}$$
$$\leftarrow constraint(min, Y, S, D, M), \tag{44}$$

$$\{value_e(S,Y,Z) : instance\_of(Z,D)\}\ M-1.$$
$$\leftarrow constraint(max,Y,S,D,M), \quad (45)$$
$$M+1\{value_e(S,Y,Z) : instance\_of(Z,D)\}.$$
$$\leftarrow constraint(exact,Y,S,D,M), \quad (46)$$
$$\{value_e(S,Y,Z) : instance\_of(Z,D)\}\ M-1.$$
$$\leftarrow constraint(exact,Y,S,D,M), \quad (47)$$
$$M+1\{value_e(S,Y,Z) : instance\_of(Z,D)\}.$$

For a detailed explanation of the above rules, please refer to [6]. An *OO-domain* is a collection of rules of the form (2)—(20). ¿From now on, whenever we refer to an OOKB, we mean the prorgram $D \cup \Pi_R$, denoted by $KB(D)$, where $D$ is the OO-domain of the OOKB[5].

### 2.3 `KB_Bio_101`: An OOKB Usage and Some Key Characteristics

The `KB_Bio_101` is an instance of OOKB and is available in ASP format[6]. The KB is based on an upper ontology called the Component Library [3]. The biologists used a knowledge authoring system called AURA to represent knowledge from a biology textbook. As an example, in Figure 1, we show an example AURA graph. The white node labeled as `Eukaryotic-Cell` is the root node and represents the universally quantified variable $X$, whereas the other nodes shown in gray represent existentials, or the Skolem functions $f_n(X)$. The nodes labeled as `has_part` and `is_inside` represent the relation names. The authoring process in

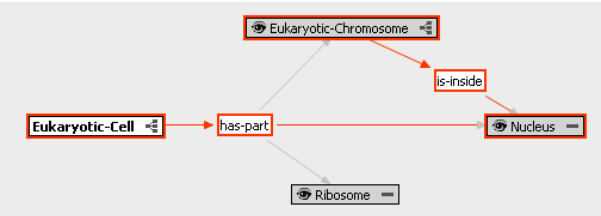



**Fig. 1.** Example graph for "Eukaryotic-Cell"

AURA can be abstractly characterized as involving three steps: *inherit, specialize and extend.* For example, the biologist creates the class `Eukaryotic-Cell` as a subclass of `Cell`. While doing so, the system would first inherit the relation values defined for `Cell` which in this case is a `Chromosome`, and show it in the graphical editor. The biologist then uses a gesture in the editor to specialize the inherited `Chromosome` to a `Eukaryotic-Chromosome`, and then introduces a new `Nucleus` and relates it to the `Eukaryotic-Chromosome`, via an `is-inside` relationship. The

[5] In [6], general OOKBs, that can contain arbitrary logic programming rules, were defined. The discussion in this paper is applicable to general OOKBs as well.

[6] See `http://www.ai.sri.com/~halo/public/exported-kb/biokb.html`

inherited `Chromosome` value for the `has-part` relationship, is thus, specialized to `Eukaryotic-Chromosome` and extended by connecting it to the `Nucleus` by using an `is-inside` relationship.
The statistics about the size of the exported OOKB are summarized in Table 1. In total `KB_Bio_101` has more than 300,000 non-ground rules. It contains 746 individuals which are members of classes which represent constants of measurements, colors, shapes, quantity, priority, etc. The KB does not contain individuals of biology classes such as `cell`, `ribosome`, etc. For computing properties of an individual or comparing individuals, the input needs to introduce the individuals.

| | | | |
|---|---|---|---|
| classes | 6430 | *domain* constraints | 449 |
| individuals | 746 | *range* constraints | 447 |
| relations | 455 | *inverse* relation statements | 442 |
| *subclass_of* statements | 6993 | *compose* statements | 431 |
| *subrelation_of* statements | 297 | qualified number constraints | 936 |
| *instance_of* statements | 714 | sufficient conditions | 198 |
| *disjoint*-ness statements | 18616 | descriptive rules | 6430 |
| avg. number of Skolem functions in each descriptive rule | 24 | equality statements | 108755 |

**Table 1.** Statistics on `KB_Bio_101`

## 3 Queries in OOKBs

We will now describe the queries given an OOKB, say $KB(D)$. These queries play a central role in the educational application `Inquire` [17] which employs the knowledge encoded in `KB_Bio_101`. These queries were developed by extensive analysis of the questions from an exam, the questions at the back of the book, and the questions that are educationally useful [4, 18].
We divide these queries into four groups. The first type of queries which includes the first two queries asks about facts and relatiolnships. The second type of queries asks about the taxonomic information. These first two question types are usually referred to as the *wh*-questions. The third type is about the differences and similarities between individuals from different classes. This type of query has been traditionally studied as an example of analogical reasoning [10]. The fourth type of queries that includes the last two questions query for relationships between concepts and are unique to our work.

- what is a `eukaryotic cell`?
- what process provides raw materials for the `citric acic cycle` during *cellular respiration*?
- is `oocyte` a subclass of a `eukaryotic cell`?
- describe the differences and similarities between `mitochondrions` and `chloroplasts`
- What is the relationship between a `mitochondrion` and a `chloroplast`

- in the absence of `oxygen`, `yeast cells` can obtain `energy` by*which process*?

Let $Z$ be a set of literals of $KB(D)$, $r$ be a relation, and $i$ be an individual from a class $c$. $\mathcal{T}(i)$ denotes the set of terms constructable from Skolem functions and the individual $i$. We characterize the set of pairs in the relation $r$ w.r.t. $Z$ in $KB(D)$ by the set $V(r,i,c,Z) = \{(r,x,y) \mid value(r,x,y) \in Z, x,y \in \mathcal{T}(i)\}$ if $instance\_of(i,c) \in Z$; otherwise, $V(r,i,c,Z)=\emptyset$.

**Definition 1 (Value set of an individual).** *Let $KB(D)$ be an OOKB. For an answer set $M$ of $KB(D)$, the* value set of an individual $i$ at a class $c$ *w.r.t. $M$, $\Sigma(i,c,M)$, is defined by $\Sigma(i,c,M) = \bigcup_{relation(r)\in M} V(i,c,r,M)$.*

Observe that the rules (29)—(41) indicate that $KB(D)$ can have multiple answer sets. Nevertheless, the structure of $KB(D)$ allows us to prove the following important property of answer sets of $KB(D)$.

**Proposition 1.** *Let $KB(D)$ be an OOKB. For every two answer sets $M_1$ and $M_2$ of $KB(D)$, every literal in $M_1 \setminus M_2$ has one of the following forms: (*i*) $substitute(x,y)$; (*ii*) $is\_substituted(x,y)$; or (*iii*) $value_e(r,x,y)$.*

The above proposition indicates that $\Sigma(i,c,M_1) = \Sigma(i,c,M_2)$ for arbitrary individual $i$ and class $c$ and answer sets $M_1$ and $M_2$ of $KB(D)$. The relationship between atoms of the form $value(r,x,y)$ and $value_e(r,x,y)$ is as follows.

**Proposition 2.** *Let $KB(D)$ be an OOKB, $i$ an individual, and $c$ a class. For every answer sets $M$ of $KB(D)$, we have that $value_e(r,x,y) \in M$ iff there exists $x', y'$ such that (*i*) $M$ contains the following atoms $eq(x',x)$, $eq(y',y)$, $substitute(x',x)$, and $substitute(y',y)$; and (*ii*) $(r,x',y') \in \Sigma(i,c,M)$.*

The significance of these two propositions is that cautious reasoning about values of individuals at classes can be accomplished by computing *one* answer set of $KB(D)$. As we will see, the majority of queries is related to this type of reasoning. We next describe, for each query $Q$, an input program $I(Q)$ and a set $R(Q)$ of rules for computing the answer of $Q$. Throughout the section, $KB$ denotes an arbitrary but fixed OOKB $KB(D)$ and $KB(Q) = KB(D) \cup I(Q) \cup R(Q)$.

### 3.1 Subsumption Between Classes ($\mathbf{Q_1}$)

Subsumption requires us to compute whether a class $c_1$ is subsumed by a class $c_2$, i.e., whether for each answer set $M$ of $KB(Q_1)$, we have for each $instance\_of(x,c_1) \in M$ also $instance\_of(x,c_2) \in M$. We can answer this question by introducing a fresh constant $i$ in the OOKB and set $I(Q_1) = \{instance\_of(i,c_1)\}$. $R(Q_1)$ consists of a rule:

$$subclass\_of(c_1,c_2) \leftarrow instance\_of(i,c_2) \tag{48}$$

Indeed, we then have that a class $c_1$ is subsumbed by $c_2$ iff for each answer set $M$ of $KB(Q_1)$, $subclass\_of(c_1,c_2) \in M$. Proposition 1 can be extended to $KB(Q_1)$ and thus we only need to compute one answer set of $KB(Q_1)$. Note that this shows how, as in description logics, subsumption can be reduced to entailment in the OOKB framework. We can show that

**Proposition 3.** *If $KB(Q_1)$ has an answer set $M$ and $subclass\_of(c_1, c_2) \in M$ then $c_1$ is subsumed by $c_2$.*

We note that computing answer sets of $KB(Q_1)$ is not a simple task (see [6]). In particular, the problem for `KB_Bio_101` is quite challenging due to its size and the potential infiniteness of the grounding program of $KB(Q_1)$.
One can define many more taxonomic queries than what we have considered here. Some examples of such queries are as follows. Given a class $C$, compute all its super classes or subclasses? Given a class, return only most specific superclass? Given two classes, return there nearest common superclass?
Some of the taxonomic queries can require a higher order representation. For example, given two classes, compute a class description that is their union or intersection. Such queries are straightforward in a DL system, and are examples of capabilities that are challenging for the current ASP systems.

### 3.2 Description of an Individual ($Q_2$)

Queries about the description of an individual ask for a description of an individual of a class $c$, represented by a fresh constant $i$ in the language of $KB(D)$. This query can be represented by the program $I(Q_2) = \{get\_value(i, c).instance\_of(i, c).\}$ where $get\_value(i, c)$ encodes the query of "inquiring about values of $i$ at the class $c$." We will now discuss the answer to this query. Intuitively, a complete description of $i$ should contain the following information:

- $C(c)$=$\{d \mid KB(D) \models subclass\_of(c, d)\}$, the classes from which $i$ inherits its relation values; and
- its relation values, i.e., the triples in $\Sigma(i, c, M)$ where $M$ is a given answer set of $R(Q_2)$.

Computing a complete description of $i$ could be achieved by the following rules:

$$out\_member\_of(Y) \leftarrow get\_value(I, C), instance\_of(I, C), instance\_of(I, Y). \quad (49)$$

$$out\_value(R, X, Y) \leftarrow get\_value(I, C), value(R, X, Y), relation(R), term\_of(X, I), term\_of(Y, I). \quad (50)$$

where $term\_of(X, I)$ defines a term $(X)$ that is constructable from Skolem functions and an individual $(I)$, $out\_member\_of(d)$ indicates that $i$ is an instance of the class $d$ (i.e., $d \in C(c)$), and $out\_value(r, x, y)$ says that $KB(D) \models value(r, x, y)$. This answer is correct but may contain *too much* information for users of an OOKB who have knowledge about the class hierarchy. This is because the above description could also include values that $i$ can inherit from the superclasses of $c$. This can be seen in the next example.

*Example 1.* Let us consider the class `Eukaryotic cell`. The description of this class contains 88 statements of the form (13)—(14) that involve 167 classes and 150 equality specifications. A first-level answer[7] computed using (49)–(50)

[7] Current solvers can only approximate the answer due to the infiniteness of the grounding program. We computed the answer by limiting the maximum nesting level for complex terms of the term to be 1 (e.g., the option `maxnesting` in **dlv**).

contains 9 atoms of the form $out\_member\_of(x)$ which indicate that a `eukaryotic cell` is also a `cell`, a `living entity`, a `physical object`, etc. In addition, there are 643 atoms of the form $out\_value(r, x, y)$ which contains inverse, composition, sub-relation, and the relation value defined in statements of the form (13)—(14) and those that are obtained by the rules (25)–(27).

The example highlights two challenges in computing the description of an individual. First, since the grounding of the KB is infinite, it raises the question of what counts as an adequate grounding that returns a sufficient description of an individuals? Second, for practical query answering applications that use `KB_Bio_101`, one must post-process the results to deciding which subset of the answers should be preesnted to the user. It should be noted that because of the infiniteness of the grounded KB, current ASP solvers can be used to approximate the answers, by setting depth bounds. Whether this will result in acceptable performance, both in terms of the quality of the answers and the efficiency, is a topic open for future research.

### 3.3 Comparing between Classes ($Q_3$)

A comparison query takes the general form of "What are the differences/similarities between $c_1$ and $c_2$?" (e.g., "what are the differences between *chromosome* and *ribosome*?"). More specific versions of the query may ask for specific kinds of differences, e.g., structural differences.
The query can be represented and answered by (*i*) introducing two new constants $i_1$ and $i_2$ which are instances of $c_1$ and $c_2$, respectively; and (*ii*) identifying the differences and similarities presented in the descriptions of $i_1$ and $i_2$. We therefore encode $I(Q_3)$ using the following program:

$$instance\_of(i_1, c_1). \quad instance\_of(i_2, c_2). \quad comparison(i_1, c_1, i_2, c_2). \tag{51}$$

Let us first discuss the features that can be used in comparing individuals of two classes. Individuals from two classes can be distinguished from each other using different dimensions, either by their superclass relationship or by the relations defined for each class. More specifically, they can be differentiated from each other by the generalitation and/or specialitation between classes; or the properties of instances belonging to them. We will refer to these two dimensions as *class-dimension* and *instance-dimension*, respectively. We therefore define the following notions, given an answer set $M$ of $KB(Q_3)$:

- *The set of similar classes between* $c_1$ *and* $c_2$: is the intersection between the set of superclasses of $c_1$ and of $c_2$ $U(c_1, c_2) = C(c_1) \cap C(c_2)$.
- *The set of different classes between* $c_1$ *and* $c_2$: is the set difference between the set of superclasses of $c_1$ and of $c_2$ $D(c_1, c_2) = (C(c_1) \backslash C(c_2)) \cup (C(c_2) \backslash C(c_1))$.

where $C(c)$ denotes the set of superclasses of $c$.
We next discuss the question of what should be considered as a similar and/or different property between individuals of two different classes. Our formalization is motivated from the typical answers to this type of question such as an answer "a chromosome has a part as protein but a ribosome does not" to the query

"what is the different between a chromosome and a ribosome?" This answer indicates that for each chromosome $x$ there exists a part of $x$, say $f(x)$, which is a protein, i.e., $value(has\_part, x, f(x))$ and $instance\_of(f(x), protein)$ hold; furthermore, no part of a ribosome, say $y$, is a protein, i.e., there exists no $g$ such that $value(has\_part, y, g(y))$ and $instance\_of(g(y), protein)$ hold.
For a set of literals $M$ of $KB(Q_3)$ and a class $c$ with $instance\_of(i, c) \in M$, let $T(i, c)$ be the set of triples $(r, p, q)$ such that $(r, x, y) \in \Sigma(i, c, M)$, $instance\_of(x, p) \in M$, and $instance\_of(y, q) \in M$. $p$ ($q$) is called the domain (range) of $r$ if $(r, p, q) \in T(i, c)$. We define

- *The set of similar relations between $c_1$ and $c_2$*: is the set $R^s(c_1, c_2)$ of relations $s$ such that (*i*) $c_1$ and $c_2$ are domain of $s$; (*ii*) $c_1$ and $c_2$ are range of $s$; or (*iii*) there exist $(p, q)$ such that $(s, p, q) \in T(i_1, c_1) \cap T(i_2, c_2)$.
- *The set of different relations between $c_1$ and $c_2$*: is the set $R^d(c_1, c_2)$ of relations $s$ such that (*i*) $c_1$ is and $c_2$ is not a domain of $s$ or vice versa; (*ii*) $c_1$ is and $c_2$ is not a range of $s$ vice versa; or (*iii*) there exist $(p, q)$ such that $(s, p, q) \in (T(i_1, c_1) \setminus T(i_2, c_2)) \cup (T(i_2, c_2) \setminus T(i_1, c_1))$.

An answer to $\mathbf{Q_3}$ must contain information from $U(c_1, c_2)$, $D(c_1, c_2)$, $R^s(c_1, c_2)$, and $R^d(c_1, c_2)$. Computing $U(c_1, c_2)$ and $D(c_1, c_2)$ rely on the rules for determining the most specific classes among a group of classes which can easily be implemented using the naf-operator.
We now describe the set of rules $R(Q_3)$, dividing it into different groups. First, the set of rules for computing $U(c_1, c_2)$ is as follows:

$$shared(C, P, Q) \leftarrow comparison(X, P, Y, Q), subclass\_of(P, C), subclass\_of(Q, C). \quad (52)$$

The rule identifies the classes that are superclass of both $c_1$ and $c_2$. We can show that $KB(Q_3) \models shared(c, c_1, c_2)$ iff $c \in U(c_1, c_2)$.
The next set of rules is for computing $D(c_1, c_2)$.

$$dist(C, P, Q) \leftarrow comparison(X, P, Y, Q), subclass\_of(P, C), not\ subclass\_of(Q, C). \quad (53)$$
$$dist(C, P, Q) \leftarrow comparison(X, P, Y, Q), not\ subclass\_of(P, C), subclass\_of(Q, C). \quad (54)$$

The two rules identify the classes that are superclass of $c_1$ but not $c_2$ and vice versa. Again, we can show that $KB(Q_3) \models dist(c, c_1, c_2)$ iff $c \in D(c_1, c_2)$.
For computing $R^s(c_1, c_2)$ and $R^d(c_1, c_2)$, we need to compute the sets $T(i_1, c_1)$ and $T(i_2, c_2)$. For this purpose, we define two predicates $t_1$ and $t_2$ such that for every answer set $M$ of $KB(Q_3)$, $t_k(s, p, q) \in M$ iff $(s, p, q) \in T(i_k, c_k)$ for $k = 1, 2$. Before we present the rules, let us denote a predicate $msc\_of$, called the *most specific class of an individual*, by the following rules.

$$not\_msc\_of(X, P) \leftarrow subclass\_of(Q, P), instance\_of(X, P), instance\_of(X, Q). \quad (55)$$
$$msc\_of(X, P) \leftarrow instance\_of(X, P), not\ not\_msc\_of(X, P). \quad (56)$$

These rules state that the class $p$ is the most specific class of an individual $x$ if $x$ is a member of $p$ and $x$ is not an instance of any subclass $q$ of $p$. This will allow us to define the set $T(i_1, c_1)$ and $T(i_2, c_2)$ as follows.

$$3\{t_1(R, P, Q), \leftarrow comparison(X_1, C_1, Y_1, C_2), value(R, X, Y), \quad (57)$$

$$\begin{aligned} q\_d(R,P), \quad & term\_of(Y,X_1), term\_of(X,X_1), \\ q\_r(R,Q)\} \quad & msc\_of(X,P), msc\_of(Y,Q). \\ 3\{t_2(R,P,Q), \leftarrow \; & comparison(X_1,C_1,Y_1,C_2), value(R,X,Y), \qquad (58) \\ q\_d(R,P), \quad & term\_of(X,Y_1), term\_of(Y,Y_1), \qquad (59) \\ q\_r(R,Q)\} \quad & msc\_of(X,P), msc\_of(Y,Q). \end{aligned}$$

The following rules identify relations that are similar between $c_1$ and $c_2$:

$$shared\_property(R) \leftarrow comparison(X_1,C_1,Y_1,C_2), t_1(R,C_1,Q_1), t_2(R,C_2,Q_2). \quad (60)$$
$$shared\_property(S) \leftarrow comparison(X_1,C_1,Y_1,C_2), t_1(R,P_1,C_1), t_2(R,P_2,C_2). \quad (61)$$
$$shared\_property(S) \leftarrow comparison(X_1,C_1,Y_1,C_2), t_1(R,P,Q), t_2(R,P,Q). \quad (62)$$

The rules say that individuals $i_1$ and $i_2$ from class $c_1$ and $c_2$ respectively share a relation $r$. The first rule says that $i_k$ $(k = 1, 2)$ is a source in the relation $r$ (i.e., there exists some $t_k$ such that $(r, i_k, t_k) \in \Sigma(i_k, c_k, M)$); The second rule says that $i_k$ is a destination in the relation $r$ (i.e., the first rule: there exists some $t_k$ such that $(r, t_k, i_k) \in \Sigma(i_k, c_k, M)$). The third rule says that there exist some pair $t_k^1, t_k^2$ such that $t_k^1$ and $t_k^2$ are instances of the same class and $(r, t_k^1, t_k^2) \in \Sigma(i_k, c_k, M)$.
The set of rules for computing $R^d(c_1, c_2)$ is similar to the above set of rules. It is omitted here for space reason.
The key challenge in computing the differences/similarities between classes in `KB_Bio_101` are the same as for $Q_2$. First, since the grounded program is infinite, one has to determine what is an adequate description that should be used for the purposes of comparsion. Second, even though the computation will return all differences and similarties, the users are frequently interested in knowing about salient differences. The current AURA system uses a complex set of heuristics to post process the results to group and rank the results to draw out the salience. The description of such heuristics is outside the scope of the present paper.

### 3.4 Relationship between Individuals ($Q_4$)

A relationship query takes the general form of "What is the relationship between individual $i_1$ and individual $i_2$?", e.g., "what is the relationship between a `biomembrane` and a `carbohydrate`"? Since this type of query refers to a path between two individuals, it can involve significant search in the KB making it especially suitable for solution by ASP solvers. In more specific forms of this query, the choice of relationships can be limited to a specific subset of relationships in the KB. For example, "What is the structural or functional relationship between individual $i_1$ and individual $i_2$?" We can formulate this query as follows. Given a set of literals $M$ of an OOKB and a set of relations $S$, a sequence of classes alternated with relation $\omega = (c_1, s_1, c_2, s_2, \ldots, s_{n-1}, c_n)$ is called a *path between $q_1$ and $q_n$* with restrictive relations $S$ in $M$ if there exists $instance\_of(t, c_1) \in M$ and Skolem functions $f_1 = id, f_2, \ldots, f_{n-1}$ such that $value(s_i, f_i(t), f_{i+1}(t)) \in M$ for $i = 1, \ldots, n-1$ and $instance\_of(f_i(t), c_i) \in M$ for $i \geq 2$ and $s_i \in S$ for

$1 \leq i < n$. A query of type $\mathbf{Q_4}$ asks for a path between $c_1$ and $c_2$ with restrictive relations in $S$ and is encoded by the program $I(Q_4)$:

$$instance\_of(i_1, c_1). \quad instance\_of(i_2, c_2). \quad p\_relation(c_1, c_2). \quad include(r). (r \in S)$$

The answer to the query should indicate paths between $c_1$ and $c_2$ with restrictive relations in $S$. Observe that an answer can be generated by $(i)$ selecting some atoms of the form $value(s, x, y)$ such that $s \in S$; and $(ii)$ checking whether these atoms create a path from $c_1$ to $c_2$. We next present the set of rules $R(Q_4)$, dividing them into two groups that implement the steps $(i)$ and $(ii)$ as follows.

$$p\_segment(R, E, C, F, D) \leftarrow include(R), value(R, E, F), instance\_of(E, C), \quad (63)$$
$$instance\_of(F, D).$$
$$\{seg(S, E, C, F, D)\} \leftarrow p\_segment(S, E, C, F, D). \quad (64)$$
$$\leftarrow p\_relation(C_1, C_2), \{seg(\_, \_, C_1, \_, \_)\}0. \quad (65)$$
$$\leftarrow p\_relation(C_1, C_2), 2\{seg(\_, \_, C_1, \_, \_)\}. \quad (66)$$
$$\leftarrow p\_relation(C_1, C_2), \{seg(\_, \_, \_, \_, C_2)\}0. \quad (67)$$
$$\leftarrow p\_relation(C_1, C_2), 2\{seg(\_, \_, \_, \_, C_2)\}. \quad (68)$$

The first rule defines possible segments of the path. The second rule, a choice rule, picks some arbitrary segments to create the path. A segment is represented by the atom $seg(s, e, c, e', c')$ that encodes a relation $s$ between $e$ (an instance of class $c$) and $e'$ (an instance of class $c'$). The rest of the rules eliminate combinations that do not create a path from $c_1$ to $c_2$. For example, the first two constraints make sure that there must be exactly one segment starting from $c_1$; the next two ensure that there must be exactly one segment that ends at $c_2$. The next four constraints make sure that the segments create a path.

$$\leftarrow p\_relation(C_1, C_2), seg(S, E, C, E_1, D), D \neq C_2, \{seg(\_, E_1, D, \_, \_)\}0. \quad (69)$$
$$\leftarrow p\_relation(C_1, C_2), seg(S, E, C, E_1, D), D \neq C_2, 2\{seg(\_, E_1, D, \_, \_)\}. \quad (70)$$
$$\leftarrow p\_relation(C_1, C_2), seg(S, E, C, E_1, D), D \neq C_2, C \neq C_1, \{seg(\_, \_, \_, E, C)\}0. \quad (71)$$
$$\leftarrow p\_relation(C_1, C_2), seg(S, E, C, E_1, D), D \neq C_2, C \neq C_1, 2\{seg(\_, \_, \_, E, C)\}. \quad (72)$$

Even if one could define a suitable finite grounding of `KB_Bio_101`, computing $KB(\mathbf{Q_4})$ can be exponential in the worst case. The implementation of this query in AURA relies on a set of heuristics and depth-bound incomplete reasoning. E.g., one heuristic involves first performing the search on the subclass relationships. The existing implementation is unsatisfactory as it misses out important relationships. In an ideal implementation, one would first compute all candidate paths, and then rank them based on user provided critieria. Computing all such paths especially at the runtime has been infeasible in AURA so far. We hope that ASP could provide a solution for an efficient path computation.

## 4 Discussion

We observe that there was no use of default negation in the axioms (2)-(20) that specify OOKB. The default negation is used in the domain independent axioms,

for example, in axiom (38) and in axioms (53)-(54). In principle, default negation could be used in axioms (13) or axiom (14), but in our practical experience in developing `KB_Bio_101` such usage has not been necessary. That is primarily because while formalizing the textbook knowledge, one typically requires classical negation. It is only for query answering that the usage of negation becomes critical. If one generalizes OOKB to other domains, it may well be necessary to use default negation in the domain specification axioms (2)-(20), but we have not considered such usage in our work so far. Since default negation is necessary to specify query answering for OOKB, ASP provides a compelling paradigm for declarative specification of such reasoning.

Let us also consider comparison between using ASP vs DLs for OOKB queries presented here. There are two key features of OOKBs that are not directly expressible in description logics: graph-structured objects and (in)equality statements. Using axioms (13) and (14), it is possible to define a graph structure. It is well known that graph structured descriptions usually lead to undecidability in reasoning [16]. In(equality) statements as in axiom (19) and (20), allow us to relate skolem functions that have been introduced as part of two different class descriptions. Such modeling paradigm is not supported by DLs. Of course, the reasoning with OOKBs in full generality is undecidable, and it is an open question whether there exist decidable fragments of OOKB for which the reasoning is decidable [6].

Another important difference between a DL and ASP is in handling of constraints. To illustrate this difference, consider a KB that has a statement: every person has exactly two parents, and further individuals $p_1$, $p_2$, $p_3$ and $p_4$, such that $p_2$, $p_3$ and $p_4$ are the parents of $p_1$. With axioms (43)-(47), such a KB will be inconsistent. In contrast, most DL system will infer that either $p_2$ must be equal to $p_3$, or $p_3$ must be equal to $p_4$, or $p_4$ must be equal to $p_2$. The semantics of constraints in AURA conform to the semantics captured in axioms (43)-(47).

Our work on formalizing the OOKB queries in ASP has been only theoretical, and an experimental evaluation is open for future work. Some example answers of the queries considered in Section 3 which are produced by the Inquire system can be seen at [17].

## 5 Conclusions

We described the contents of an OOKB knowledge base, and formulated ASP programs for answering four classes of practically interesting queries. We also presented a practical OOKB, `KB_Bio_101`, whose size and necessary features make the computation of the answers to these queries almost impossible using contemporary ASP solvers. The specific challenges include developing suitable grounding strategies and dealing with potential undecidability in reasoning with an OOKB. Given the large overlap in features supported by OOKB and DLs, the `KB_Bio_101` also presents a unique dataset which could be used to explore relative tradeoffs in reasoning efficiency across these two different paradigms. Being a concrete OOKB, `KB_Bio_101` presents a real challenge for the development

of ASP-solvers. This also calls for the development of novel query answering methods with huge programs in ASP. We welcome engaging with both the ASP and DL research communities so that `KB_Bio_101` could be used as a driver for advancing the state of the art in efficient and scalable reasoning.

**Acknowledgment**

This work was funded by Vulcan Inc. and SRI International.

## References

1. M. Alviano, W. Faber, and N. Leone. Disjunctive ASP with functions: Decidable queries and effective computation. *TPLP*, 10(4-6):497–512, 2010.
2. F. Baader, I. Horrocks, and U. Sattler. Description Logics. In *Handbook of Knowledge Representation*. Elsevier.
3. K. Barker, B. Porter, and P. Clark. A library of generic concepts for composing knowledge bases. In *Proc. 1st Int Conf on Knowledge Capture*, 2001. 14–21.
4. P. Clark. and V. Chaudhri and S. Mishra and J. Thomere and K. Barker and B. Porter. Enabling Domain Experts to Convey Questions to Machine: A Modified, Template-Based Approach In *Proc. 2nd Int Conf on Knowledge Capture*, 2003. 14–21.
5. A. Calì, G. Gottlob, and T. Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *Database Theory - ICDT 2009*. ACM.
6. V. Chaudhri, S. Heymans, M. Wessel, and T. C. Son. Object Oriented Knowledge Bases in Logic Programming. In Technical Communications of International Conference on Logic Programming, 2013.
7. V. Chaudhri, S. Heymans, and M. Wessel. `KB_Bio_101`: A Challenge for OWL Reasoners. In *Proceedings of the Workshop on OWL Reasoner Evaluation*, 2013.
8. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. KRR, 406–417, 1998.
9. P. Clark and B. Porter. *KM (v2.0 and later): Users Manual*, 2011.
10. S. Nicholson and K. Forbus Answering Comparison Questions in SHAKEN: A Progress Report *AAAI Spring Symposium on Mining Answers From Text and Knowledge Bases*, 2002.
11. T. Eiter and M. Simkus. FDNC: Decidable nonmonotonic disjunctive logic programs with function symbols. *ACM TOCL*, 11(2), 2010.
12. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. LPNMR'07, LNAI 4483, 260–265. Springer-Verlag, 2007.
13. M. Gelfond and V. Lifschitz. Logic programs with classical negation. *ICLP*, 579–597, 1990.
14. D. Gunning, V. K. Chaudhri, P. Clark, K. Barker, S-Y. Chaw, M. Greaves, B. Grosof, A. Leung, D. McDonald, S. Mishra, J. Pacheco, B. Porter, A. Spaulding, D. Tecuci, and J. Tien. Project Halo Update—Progress Toward Digital Aristotle. *AI Magazine*, pages 33–58, 2010.
15. D. Magka, B. Motik, B., and I. Horrocks. Modeling Structured Domains using Description Graphs and Logic Programming. In *DL 2012*.
16. B. Motik, B. C. Grau, I. Horrocks, and U. Sattler. Representing ontologies using description logics, description graphs, and rules. AIJ, 173:1275-1309, 2009.

17. A. Overholtzer, A. Spaulding, V. K. Chaudhri, and D. Gunning. Inquire: An Intelligent Textbook. In *Proceedings of AAAI Video Competition Track*, 2012. `http://www.aaaivideos.org/2012/inquire_intelligent_textbook/`.
18. V. Chaudhri, B. Cheng, A. Overholtzer, J. Roschelle, A. Spaulding, P. Clark, M. Greaves, D. Gunning. Inquire Biology: A Textbook that Answers Questions. In *AI Magazine*, Vol 34, No 3, September 2013.
19. J. B. Reece, L. A. Urry, M. L. Cain, S. A. Wasserman, P. V. Minorsky, and R. B. Jackson. *Campbell Biology, 9/E.* Benjamin Cummings, 2011.
20. P. Simons, N. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
21. M. Wessel, V. Chaudhri, and S. Heymans. Automatic Strengthening of Graph-Structured Knowledge Bases. In *3rd International Workshop on Graph Structures for Knowledge Representation and Reasoning*, 2013.

# The DIAMOND System for Argumentation: Preliminary Report*

Stefan Ellmauthaler and Hannes Strass

Computer Science Institute, Leipzig University

**Abstract** Abstract dialectical frameworks (ADFs) are a powerful generalisation of Dung's abstract argumentation frameworks. In this paper we present an answer set programming based software system, called DIAMOND (DIAlectical MOdels eNcoDing). It translates ADFs into answer set programs whose stable models correspond to models of the ADF with respect to several semantics (i.e. admissible, complete, stable, grounded).

## 1 Introduction

Formal argumentation has established itself as a vibrant subfield of artificial intelligence, contributing to such diverse topics as legal decision making and multi-agent interactions. A particular, well-known formalism to model argumentation scenarios are Dung's abstract argumentation frameworks [1]. In that model, arguments are treated as abstract atomic entities. The only information given about them is a binary relation expressing that one argument attacks another.

A criticism often advanced against Dung frameworks is their restricted expressive capability of allowing only attacks between arguments. This leads to quite a number of extensions of Dung AFs, for example with attacks from sets of arguments [2], attacks on attacks [3] and meta-argumentation [4]. Unifying these and other extensions to AFs, Brewka and Woltran [5] proposed a general model, abstract dialectical frameworks (ADFs). In ADFs, attack, support, joint support, combined attacks and many more relations between arguments (called statements there) can be expressed, while the whole formalism stays on the same abstraction level as Dung's.

In this paper we present the DIAMOND software system that computes models of ADFs with respect to several different semantics. The name DIAMOND abbreviates "DIAlectical MOdels eNcoDing" and hints at the fact that DIAMOND is built on top of the state of the art in answer set programming: abstract dialectical frameworks are encoded into logic programs, and an answer set solver is used to compute the models of the ADF. By providing an expressive argumentation formalism with an implementation, we pave the way for practical applications of ADFs in scenarios where dialectical aspects are of interest, for example in group decision making.

The paper proceeds as follows. We first introduce the necessary background in abstract dialectical frameworks and answer set programming. We then present

* This research has been supported by DFG projects BR 1817/7-1 and FOR 1513.

the DIAMOND system – how ADFs are represented there, and how the ADF semantics are encoded into answer set programs. We conclude with a contrasting discussion of the most significant related work.

## 2 Background

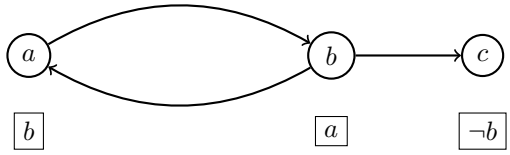
An abstract dialectical framework (ADF) [5] is a directed graph whose nodes represent statements or positions that can be accepted or not. The links represent dependencies: the status of a node $s$ only depends on the status of its parents (denoted $par(s)$), that is, the nodes with a direct link to $s$. In addition, each node $s$ has an associated acceptance condition $C_s$ specifying the exact conditions under which $s$ is accepted. $C_s$ is a function assigning to each subset of $par(s)$ one of the truth values $\mathbf{t}$, $\mathbf{f}$. Intuitively, if for some $R \subseteq par(s)$ we have $C_s(R) = \mathbf{t}$, then $s$ will be accepted provided the nodes in $R$ are accepted and those in $par(s) \setminus R$ are not accepted.

**Definition 1.** *An* abstract dialectical framework *is a tuple* $D = (S, L, C)$ *where*

- $S$ *is a set of statements (positions, nodes),*
- $L \subseteq S \times S$ *is a set of links,*
- $C = \{C_s\}_{s \in S}$ *is a set of total functions* $C_s : 2^{par(s)} \to \{\mathbf{t}, \mathbf{f}\}$.

In many cases it is convenient to represent the acceptance condition of a statement $s$ by a propositional formula $\varphi_s$, as is done in our running example.

*Example 1.* Consider the ADF $D = (S, L, C)$ with a support cycle and one attack relation: $S = \{a, b, c\}$, $L = \{(a, b), (b, a), (b, c)\}$, $\varphi_a = b, \varphi_b = a, \varphi_c = \neg b$. This ADF can also be represented as a graph, where the nodes are statements and the relations between them are directed edges. The boxes below each node are the acceptance conditions for the particular statement.




In recent work [6], we redefined several standard ADF semantics and defined additional ones. In this paper, we use these revised definitions, which are based on three-valued logic.[1] The three truth values true ($\mathbf{t}$), false ($\mathbf{f}$) and unknown ($\mathbf{u}$) are partially ordered by $\leq_i$ according to their information content: we have $\mathbf{u} <_i \mathbf{t}$ and $\mathbf{u} <_i \mathbf{f}$ and no other pair in $<_i$, which intuitively means that the classical truth values contain more information than the truth value unknown. On the set of truth values, we define a meet operation, *consensus*, which assigns

[1] For further details on those newly introduced semantics we refer the interested reader to Brewka et al. [6].

$\mathbf{t} \sqcap \mathbf{t} = \mathbf{t}$, $\mathbf{f} \sqcap \mathbf{f} = \mathbf{f}$, and returns $\mathbf{u}$ otherwise. The information ordering $\leq_i$ extends in a straightforward way to valuations $v_1, v_2$ over $S$ in that $v_1 \leq_i v_2$ iff $v_1(s) \leq_i v_2(s)$ for all $s \in S$. Obviously, a three-valued interpretation $v$ is two-valued if all statements are mapped to either true or false. For a three-valued interpretation $v$, we say that a two-valued interpretation $w$ *extends* $v$ iff $v \leq_i w$. We denote by $[v]_2$ the set of all two-valued interpretations that extend $v$. A three-valued interpretation $E_v$ has an *associated extension* $E_v = \{s \in S \mid v(s) = \mathbf{t}\}$.

Brewka and Woltran [5] defined an operator $\Gamma_D$ over three-valued interpretations. For each statement $s$, the operator returns the consensus truth value for its acceptance formula $\varphi_s$, where the consensus takes into account all possible two-valued interpretations $w$ that extend the input valuation $v$.

**Definition 2.** *Let $D$ be an ADF and $v$ be a three-valued interpretation. Then the interpretation $\Gamma_D(v)$ is given by $s \mapsto \bigsqcap\{w(\varphi_s) \mid w \in [v]_2\}$. Furthermore $v$ is* admissible *iff $v \leq_i \Gamma_D(v)$;* complete *iff $\Gamma_D(v) = v$, that is, $v$ is a fixpoint of $\Gamma_D$;* grounded *iff $v$ is the $\leq_i$-least fixpoint of $\Gamma_D$.*

*A two-valued interpretation $v$ is a* model of $D$ *iff $\Gamma_D(v) = v$; it is a* stable model *of $D = (S, L, C)$ iff $v$ is a model of $D$ and $E_v$ equals the grounded extension of the reduced ADF $D^v = (E_v, L^v, C^v)$, where $L^v = L \cap (E_v \times E_v)$ and for $s \in E_v$ we set $\varphi_s^v = \varphi_s[r/\mathbf{f} : v(r) = \mathbf{f}]$.*

*Example 2.* We will now show the models with respect to the different semantics for the ADF introduced in Example 1. For readability, we write interpretations $v$ as sets of literals $L_v = \{s \in S \mid v(s) = \mathbf{t}\} \cup \{\neg s \mid s \in S, v(s) = \mathbf{f}\}$. There are

- five admissible interpretations: $\emptyset$, $\{a, b\}$, $\{a, b, \neg c\}$ $\{\neg a, \neg b, c\}$, $\{\neg a, \neg b\}$,
- three complete models: $\{\neg a, \neg b, c\}$, $\emptyset$, $\{a, b, \neg c\}$; of which $\emptyset$ is grounded;
- two models: $\{a, b, \neg c\}$, $\{\neg a, \neg b, c\}$, of which one is stable: $\{\neg a, \neg b, c\}$

Brewka et al. [6] also defined an approach to handle preferences in ADFs. The approach generalises the one for AFs from Amgoud and Cayrol [7]. Since DIAMOND also implements this treatment of preferences, we recall it here. For this approach, the links are restricted to links that are attacking or supporting.

**Definition 3.** *A prioritised ADF (PADF) is a tuple $P = (S, L^+, L^-, >)$ where $S$ is the set of nodes, $L^+$ and $L^-$ are subsets of $S \times S$, the supporting and attacking links, and $>$ is a strict partial order (irreflexive, transitive, antisymmetric) on $S$ representing preferences among the nodes.*

Here $(a, b) \in >$ (alternatively: $a > b$) expresses that $a$ is preferred to $b$. The semantics of prioritised ADFs is given by a translation to standard ADFs: $P$ translates to $(S, L^+ \cup L^-, C)$, where for each statement $s \in S$ the acceptance condition $C_s$ is defined as: $C_s(M) = \mathbf{t}$ iff for each $a \in M$ such that $(a, s) \in L^-$ and not $s > a$ we have: for some $b \in M$, $(b, s) \in L^+$ and $b > a$. Intuitively, an attacker does not succeed if the attacked node is more preferred or if there is a more preferred supporting node.

### 2.1 Answer Set Programming

A *propositional normal logic program* $\Pi$ is a set of finite rules $r$ over a set of ground atoms $\mathcal{A}$. A rule $r$ is of the form $\alpha \leftarrow \beta_1, \ldots, \beta_m, \text{not } \beta_{m+1} \ldots, \text{not } \beta_n$, where $\alpha \in \mathcal{A}$, $\beta_i \in \mathcal{A}$ are ground atoms and $m \leq n \leq 0$. Each rule consists of a *body* $B(r) = \{\beta_1, \ldots, \beta_m, \text{not } \beta_{m+1} \ldots, \text{not } \beta_n\}$ and a *head* $H(r) = \{\alpha\}$, divided by the $\leftarrow$-symbol. We will split up the body into two parts, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_m\}$ and $B^-(r) = \{\text{not } \beta_{m+1} \ldots, \text{not } \beta_n\}$. A rule $r$ is said to be positive if $B^-(r) = \emptyset$ and a program $\Pi$ is positive if every rule $r \in \Pi$ is positive. For a positive program $\Pi$, its immediate consequence operator $T_\Pi$ is defined for $S \subseteq \mathcal{A}$ by $T_\Pi(S) = \{H(r) \in \mathcal{A} \mid r \in \Pi, B^+(r) \subseteq S\}$. A set $A \subseteq \mathcal{A}$ of ground atoms is a *minimal model* of a positive propositional logic program $\Pi$ iff $A$ is the least fixpoint of $T_\Pi$. To allow rules with negative body atoms, Gelfond and Lifschitz [8] proposed the stable model semantics (also called answer set semantics).

**Definition 4.** *Let $A \subseteq \mathcal{A}$ be a set of ground atoms. $A$ is a* stable model *for the propositional normal logic program $\Pi$ iff $A$ is the minimal model of the reduced program $\Pi^A$, where $\Pi^A = \{H(r) \leftarrow B^+(r) \mid r \in \Pi, B^-(r) \cap A = \emptyset\}$.*

We use *clasp* from the Potsdam Answer Set Solving Collection *Potassco*[2] [9] as the back-end answer set solver for our software system. Potassco allows us to use an enriched input language where in addition to the above pictured propositional logic programs we can use first-order variables and predicates. Ground atoms are generally written in lower case while variables are represented with upper case characters. Additionally *Potassco* offers features like aggregates, cardinality constraints, choice rules and conditional literals. For further details we refer to the recent book by Gebser et al. [10].

## 3 DIAMOND

Our software system DIAMOND is a collection of answer set programming encodings and tools to compute the various models with respect to the semantics for a given ADF. The different encodings are designed around the *Potsdam Answer Set Solving Collection (Potassco)* [9] and the additional provided tools utilise clasp as solver, too. Note that the encodings for DIAMOND are built in a modular way. To compute the models of an ADF with respect to a semantics, different modules need to be grounded together to get the desired behaviour.

DIAMOND is available for download and experimentation at the web page `http://www.informatik.uni-leipzig.de/~ellmau/diamond`. There we also provide further documentation on its usage. In short, DIAMOND is a Python-script,[3] which can be invoked via the command line. Different switches are used to designate the desired semantics, and the input file is given as a file name or via the standard input. The options for the command line are as follows:

[2] Available at `http://potassco.sourceforge.net`
[3] Python is available at `http://www.python.org`.

```
usage: diamond.py [-h] [-cf] [-m] [-sm] [-g] [-c] [-a]
       [--transform_pform | --transform_prio] [-all] [--version] instance
positional arguments:
   instance              File name of the ADF instance
optional arguments:
   -h, --help            show this help message and exit
   -cf, --conflict-free  compute the conflict free sets
   -m, --model           compute the two-valued models
   -sm, --stablemodel    compute the stable models
   -g, --grounded        compute the grounded model
   -c, --complete        compute the complete models
   -a, --admissible      compute the admissible models
   --transform_pform     transform a propositional formula ADF before the computation
   --transform_prio      transform a prioritized ADF before the computation
   -all, --all           compute all sets and models
   --version             prints the current version
```

We next describe how specific ADF instances are represented in DIAMOND.

### 3.1 Instance Representation

In order to represent an ADF for DIAMOND its acceptance conditions need to be in the functional representation as given in Definition 1. The statements of an ADF are declared by the predicate `s`, and the links are represented by the binary predicate `l`, such that `l(b,a)` reflects that there is a link from $b$ to $a$. The acceptance condition is modelled via the unary and tertiary predicates `ci` and `co`. Intuitively `ci` (resp. `co`) identifies the parents which need to be accepted, such that the acceptance condition maps to true (i.e. *in*) (resp. false (i.e. *out*)). To achieve a flat representation of each set of parent statements, we use an arbitrary third term in the predicate to identify them. To express what happens to a statement when none of the parents is accepted we use the unary versions of `ci` and `co`. Here is the DIAMOND representation of Example 1:

```
s(a). s(b). s(c). l(b,a). l(a,b). l(b,c).
co(a). ci(a,1,b). co(b). ci(b,1,a). ci(c). co(c,1,b).
```

The first line declares the statements and links. The second line expresses the acceptance conditions: statement $a$ is *out* if $b$ is *out* and *in* if $b$ is; likewise $b$ gets the same status as $a$; statement $c$ is *in* if $b$ is *out*, and $c$ is *out* if $b$ is *in*.

As a part of the DIAMOND software bundle, we also provide an ECL$^i$PS$^e$ Prolog[4] [11] program that transforms acceptance functions given as formulas into the functional representation used by DIAMOND.

We have chosen this functional representation of acceptance conditions for pragmatic reasons. An alternative would have been to represent acceptance conditions by propositional formulas. In this case, computing a single step of the operator would entail solving several **NP**-hard problems. The standard way to

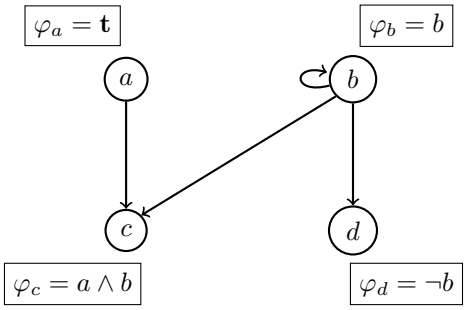
[4] ECL$^i$PS$^e$ is available at `http://eclipseclp.org/`.

solve these is the use of saturation [12], which however causes undesired side-effects when employed together with meta-ASP [13]. Furthermore, other ADF semantics (e.g. preferred) utilise concepts like $\subseteq$-minimality, which also require the use of meta-argumentation. We plan to extend DIAMOND to further semantics and therefore chose the functional representation of acceptance conditions to forestall potential implementation issues.

Due to compatibility considerations, it is possible for DIAMOND to understand the propositional formula representation as well as a PADF. The propositional formula representation uses the unary predicate `statement` to identify statements. The binary predicate `ac(s,`$\phi$`)` associates to each statement `s` one formula $\phi$. Each formula $\phi$ is constructed in the usual inductive way, where atomic formulae are other statements and the truth constants (i.e. `c(v)` and `c(f)`) and the operators are written as functions. The allowed operators are `neg, and, or, imp,` and `iff` for their respective logical operators. To describe a PADF, we use the unary predicate `s` to describe the set of statements. In addition the support (i.e. $L^+$) and attack (i.e. $L^-$) links are represented by the binary predicates `lp` and `lm` (i.e. positive resp. negative links). To express a preference, such as $a > b$, we use the predicate `pref(a,b)`. Note that DIAMOND provides a method to translate propositional formula ADFs and PADFs into ADFs with total functions and only computes the models using the functional representation.

For illustration, let us look at another, slightly more complicated example.

*Example 3.* Consider the ADF $D_2 = (S_2, L_2, C_2)$ with $S_2 = \{a, b, c, d\}$, $L_2 = \{(a, c), (b, b), (b, c), (b, d)\}$, and $C_2 = \{\varphi_a = \mathbf{t}, \varphi_b = b, \varphi_c = a \wedge b, \varphi_d = \neg b\}$.




For this ADF there are

- 16 admissible interpretations: $\emptyset$, $\{a\}$, $\{b\}$, $\{\neg b\}$, $\{b, \neg d\}$, $\{a, b\}$, $\{a, \neg b\}$, $\{\neg b, d\}$, $\{\neg b, \neg d\}$, $\{a, b, c\}$, $\{a, b, \neg d\}$, $\{a, \neg b, d\}$, $\{a, \neg b, \neg c\}$, $\{\neg b, \neg c, d\}$, $\{a, b, c, \neg d\}$, $\{a, \neg b, \neg c, d\}$
- three complete models: $\{a\}$, $\{a, b, c, \neg d\}$, $\{a, \neg b, \neg c, d\}$; of these, $\{a\}$ is the grounded model;
- two models: $\{a, b, c, \neg d\}$, $\{a, \neg b, \neg c, d\}$, of which one is stable: $\{a, \neg b, \neg c, d\}$. Its propositional formula representation for DIAMOND (inherited from `ADFsys`) is given by the following ASP code:

```
statement(a). statement(b). statement(c). statement(d).
ac(a,c(v)).
ac(b, b).
ac(c, and(a,b)).
ac(d, neg(b)).
```

The functional ASP representation of the same ADF looks thus:

```
s(a). s(b). s(c). s(d).
l(a,c). l(b,b). l(b,c). l(b,d).
ci(a).
co(b). ci(b,1,b).
co(c). co(c,1,a). co(c,2,b). ci(c,3,a). ci(c,3,b).
ci(d). co(d,1,b).
```

Arguably, the formula representation is easier to read for humans.

### 3.2 Implementation of $\Gamma_D$

Since all of the semantics are defined via the operator $\Gamma_D$, we will now present how the implementation of the operator is done in DIAMOND. The unary predicate `step` with an arbitrary term is used to apply the operator several times. The input for the operator is given by the predicates `in` and `out` to represent mappings to **t** and **f**. The resulting interpretation can be read off the predicates `valid` and `unsat`. Predicates `fp` and `nofp` denote whether a fixpoint is reached or not. First, DIAMOND decides which of the mappings to **t** are still of interest (`cii`) (i.e. which of those can still be satisfied under the given interpretation):

```
ciui(S,J,I) :- lin(X,S,I), not ci(S,J,X), ci(S,J).
ciui(S,J,I) :- lout(X,S,I), ci(S,J,X).
cii(S,J,I) :- not ciui(S,J,I), ci(S,J), step(I).
```

The predicates `lin` and `lout` are those links between arguments which are already decided by the given three-valued interpretation. The binary predicate `ci` (resp. `co`) is just the projection of its tertiary version to express that at least one predicate with a specific statement occurs in a specific acceptance condition. The treatment of the interesting mappings to **f** (`coi`) is dual:

```
coui(S,J,I) :- lin(X,S,I), not co(S,J,X), co(S,J).
coui(S,J,I) :- lout(X,S,I), co(S,J,X).
coi(S,J,I) :- not coui(S,J,I), co(S,J), step(I).
```

Afterwards it is checked whether there exist two-valued extensions of the given interpretation that are a model or not, which is denoted by the predicates `pmodel` (resp. `imodel`). Then a statement can be seen to be `valid` (resp. `unsat`) if there does not exist an interpretation which is not a model (is a model). The predicate `verum` (resp. `falsum`) represents that the acceptance condition is always true (resp. false).

```
pmodel(S,I) :- cii(S,J,I). pmodel(S,I) :- verum(S), step(I).
pmodel(S,I) :- not lin(S,I), ci(S), step(I).
pmodel(S,I) :- not lin(S,I), ci(S), step(I).
valid(S,I) :- pmodel(S,I), not imodel(S,I).

imodel(S,I) :- coi(S,J,I).
imodel(S,I) :- falsum(S), step(I).
imodel(S,I) :- not lin(S,I), co(S), step(I).
unsat(S,I) :- imodel(S,I), not pmodel(S,I).
```

At last, either `nofp` or `fp` is deduced. To achieve this, DIAMOND checks whether the application of the operator resulted in an interpretation that is different from the given one.

```
nofp(I) :- in(X,I), not valid(X,I), step(I).
nofp(I) :- valid(X,I), not in(X,I), step(I).
nofp(I) :- out(X,I), not unsat(X,I), step(I).
nofp(I) :- unsat(X,I), not out(X,I), step(I).
fp(I) :- not nofp(I), step(I).
```

### 3.3 Semantics

The admissible model is computed by the use of a guess and check approach. At first a three-valued interpretation is guessed, by an assignment of the statements to be `in`, `out`, or neither. The last two lines remove all guesses which violate the definition of the admissible model (i.e. check which guesses are right):

```
step(0).
{in(S,0):s(S)}.
{out(S,0):s(S)}.
:- in(S,0), out(S,0).
:- in(S), not valid(S,0).
:- out(S), not unsat(S,0).
```

The complete model encoding uses the same concept as used for the admissible model. The only difference is that the guessed model needs to be a fixpoint. To this effect the last two rules of the above encoding are replaced by the constraint "`:- nofp(0).`".

To compute the grounded model, we need to apply $\Gamma_D$ until a fixpoint is reached. This is done via a sequence of steps, where the result of one step is taken as the used given interpretation for the next step:

```
maxit(I) :- I:={s(S)}. step(0).
in(S,I+1) :- valid(S,I). out(S,I+1) :- unsat(S,I).
step(I+1) :- step(I), not maxit(I).
in(S) :- fp(I), in(S,I).
out(S) :- fp(I), out(S,I).
udec(S) :- fp(I), s(S), not in(S), not out(S).
```

Note that we use the number of statements as the upper bound on the number of operator applications as this is the maximal number of steps needed to reach a fixpoint.

To implement the model semantics, the operator is not essential: as the model is only two-valued, there do not remain undecided parts. So each variable is mapped to a truth-value and therefore every acceptance condition may only map to one value (i.e. **t** or **f**). The encoding just guesses a two-valued interpretation and checks whether the guessed interpretation agrees with the acceptance conditions of each statement or not. The stable model combines the encoding for models with the operator encoding to check for each model whether it is also the grounded extension of its reduced ADF or not.

## 4 Discussion and Future Work

We presented the DIAMOND software system that uses answer set programming to compute models of abstract dialectical frameworks under various semantics. DIAMOND can be seen as a continuation of the trend to utilise ASP for implementing abstract argumentation. The most important existing tool in this line of work is the ASPARTIX system[5] [14] for computing extensions of Dung argumentation frameworks.

Quite recently, Ellmauthaler and Wallner presented their system `ADFsys`[6] for determining the semantics of ADFs [15]. Since their system likewise uses answer set programming, it is natural to ask where the differences lie. For one, after the discovery of several examples where some original ADF semantics do not behave as intended, Brewka et al. [6] proposed revised and generalised versions of these semantics. The DIAMOND system implements the new semantics while `ADFsys` still computes the old versions. For another, `ADFsys` relies solely on the representation of acceptance conditions via propositional formulas, while DIAMOND can additionally deal with functional representations. Due to the new semantics it is not trivial to compare those two systems. In fact only the model and the grounded semantics have not changed. During preliminary tests, we used different methods to generate randomised ADF instances. Depending on the used generation method, DIAMOND could compete with `ADFsys` and even outperform it. Alas, there were also instances for which `ADFsys` outperformed DIAMOND. We consider it an important future task to determine specific classes of ADFs that distinguish the two systems, and to connect these ADF classes to possible real-world applications.

To adapt `ADFsys` to the new semantics, it would be needed to decide at each operation of $\Gamma_D$ which acceptance formulae are (under the given three-valued interpretation) irrefutable (resp. unsatisfiable). To solve such an embedded co-**NP**

[5] ASPARTIX is available at `http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/`

[6] `ADFsys` is available at `http://www.dbai.tuwien.ac.at/research/project/argumentation/adfsys/`

problem it would be necessary to use the saturation technique or similar concepts, which will make the use of disjunctive logic programs obligatory. Therefore there would also be issues with more complex semantics (like the preferred semantics). There the use of meta-ASP would conflict with the use of saturation in the disjunctive program.

Apart from the semantics implemented in this paper, there are also ADF semantics that DIAMOND cannot yet deal with – these remain for future work. For example, the preferred semantics is based on maximisation, and so we will need meta-ASP to implement that. In general, ADFs are a quite new formalism, and we expect that further ADF semantics will be defined in the future. Naturally, we plan to implement these new semantics using the infrastructure already available through DIAMOND.

Another future research interest concerns a possible practical application for ADFs: We intend to analyse discussions in social media, where opinions and viewpoints can be modelled by statements that are in some relation to each other. ADF semantics can guide the respective online community, for example as to what positions everybody can agree on, or how a group decision can be justified. Such an approach was proposed by Toni and Torroni [16] as a possible application of assumption-based argumentation frameworks [17]. However, assumption-based argumentation inherits the expressiveness limitations of abstract argumentation, that is, it can also express only attack relationships between statements. We expect that ADFs with their greater expressiveness are better suited to model online interactions in social media.

A similar application of argumentation in online social communities is the approach by Snaith et al. [18]. They utilise their database for arguments in the Argument Interchange Format [19] to capture discussions via different blogging-sites and use their tool TOAST [20] to compute an acceptable consensus about the issues under discussion. Again we think that ADFs are more suitable for this application due to their expressiveness.

## References

1. Dung, P.M.: On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. Artificial Intelligence **77** (1995) 321–358
2. Nielsen, S.H., Parsons, S.: A generalization of Dung's abstract framework for argumentation: Arguing with sets of attacking arguments. In: Argumentation in Multi-Agent Systems. Volume 4766 of LNCS., Springer (2006) 54–73
3. Modgil, S.: Reasoning about preferences in argumentation frameworks. Artif. Intell. **173**(9-10) (2009) 901–934
4. Boella, G., Gabbay, D.M., van der Torre, L., Villata, S.: Meta-argumentation modelling I: Methodology and techniques. Studia Logica **93**(2–3) (2009) 297–355
5. Brewka, G., Woltran, S.: Abstract Dialectical Frameworks. In: KR. (2010) 102–111
6. Brewka, G., Ellmauthaler, S., Strass, H., Wallner, J.P., Woltran, S.: Abstract Dialectical Frameworks Revisited. In: IJCAI, AAAI Press (August 2013) To appear.
7. Amgoud, L., Cayrol, C.: On the acceptability of arguments in preference-based argumentation. In: UAI, Morgan Kaufmann (1998) 1–7

8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. (1988) 1070–1080
9. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications **24**(2) (2011) 105–124
10. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers (2012)
11. Schimpf, J., Shen, K.: ECLiPSe – from LP to CLP. CoRR **abs/1012.4240** (2010)
12. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Annals of Mathematics and Artificial Intelligence **15**(3–4) (1995) 289–323
13. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theory and Practice of Logic Programming **11**(4–5) (2011) 821–839
14. Egly, U., Gaggl, S.A., Woltran, S.: Answer-set programming encodings for argumentation frameworks. Argument and Computation **1**(2) (2010) 147–177
15. Ellmauthaler, S., Wallner, J.P.: Evaluating Abstract Dialectical Frameworks with ASP. [21] 505–506
16. Toni, F., Torroni, P.: Bottom-up argumentation. In Modgil, S., Oren, N., Toni, F., eds.: TAFA. Volume 7132 of Lecture Notes in Computer Science., Springer (2011) 249–262
17. Bondarenko, A., Dung, P.M., Kowalski, R.A., Toni, F.: An abstract, argumentation-theoretic approach to default reasoning. Artificial Intelligence **93** (1997) 63–101
18. Snaith, M., Bex, F., Lawrence, J., Reed, C.: Implementing argublogging. [21] 511–512
19. Rahwan, I., Reed, C.: The Argument Interchange Format. In: Argumentation in Artificial Intelligence. Springer (2009) 383–402
20. Snaith, M., Reed, C.: Toast: Online aspic+ implementation. [21] 511–512
21. Verheij, B., Szeider, S., Woltran, S., eds.: Computational Models of Argument - Proceedings of COMMA 2012, Vienna, Austria, September 10-12, 2012. In Verheij, B., Szeider, S., Woltran, S., eds.: COMMA. Volume 245 of Frontiers in Artificial Intelligence and Applications., IOS Press (2012)

# A System for Interactive Query Answering with Answer Set Programming

Martin Gebser, Philipp Obermeier, and Torsten Schaub*

Universität Potsdam, Institut für Informatik

**Abstract.** Reactive answer set programming has paved the way for incorporating online information into operative solving processes. Although this technology was originally devised for dealing with data streams in dynamic environments, like assisted living and cognitive robotics, it can likewise be used to incorporate facts, rules, or queries provided by a user. As a result, we present the design and implementation of a system for interactive query answering with reactive answer set programming. Our system *quontroller* is based on the reactive solver *oclingo* and implemented as a dedicated front-end. We describe its functionality and implementation, and we illustrate its features by some selected use cases.

## 1 Introduction

Traditional logic programming [1, 2] is based upon query answering. Unlike this, logic programs under the stable model semantics [3] are implemented by model generation based systems, viz. answer set solvers [4]. Although the latter also allows for checking whether a query is entailed by some stable model, there is so far no way to explore a domain at hand by posing consecutive queries without relaunching the solver. The same applies to the interactive addition and/or deletion of temporary program parts that come in handy during theory exploration, for instance, when dealing with hypotheses.

An exemplary area where such exploration capacities would be of great benefit is bio-informatics (cf. [5–10]). Here, we usually encounter problems with large amounts of data, resulting in runs having substantial grounding and solving times. Furthermore, problems are often under-constrained, thus yielding numerous alternative solutions. In such a setting, it would be highly beneficial to explore a domain via successive queries and/or under certain hypotheses. For instance, for determining nutritional requirements for sustaining maintenance or growth of an organism, it is important to indicate seed compounds needed for the synthesis of other compounds. Now, rather than continuously analyzing several thousand stable models (or their intersection or union), a biologist may rather perform interactive "in-silico" experiments by temporarily adding compounds and subsequently exploring the resulting models by posing successive queries.

We address this shortcoming and show how recently developed systems for *reactive* answer set programming (ASP) [11, 12] can be harnessed to provide query answering

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

and theory exploration capacities. In fact, reactive ASP was conceived for incorporating online information into operative ASP solving processes. Although this technology was originally devised for dealing with data streams in dynamic environments, like assisted living and cognitive robotics, it can likewise be used to incorporate facts, rules, or queries provided by a user. As a result, we present the design and implementation of a system for interactive query answering and theory exploration with ASP. Our system *quontroller*[1] is based on the reactive answer set solver *oclingo* and implemented as a dedicated front-end. We describe its functionality and implementation, and we illustrate its features on a running example.

## 2 Approach

In order to provide dedicated support for query answering, the *quontroller* encapsulates *oclingo* along with its basic front-end for entering online progressions. The basic idea is to condition the stable models of an underlying logic program via *query programs*, temporarily asserting atoms to be contained in stable models of interest. For circumventing restrictions due to the modularity requirement of reactive ASP (cf. [11, 12]) and enabling repeated assertions of an atom in a series of queries, the *quontroller* associates query programs with sequence numbers and exploits *oclingo*'s step counter to automatically map their contents. In the following, we detail this idea on the well-known example of $n$-coloring.

To begin with, Listing 1 provides an ASP encoding of $n$-coloring. The encoding applies to graphs represented by facts over predicate `edge`/2. Given such facts, the nodes of the graph are extracted in Line 6–7, each node is marked with exactly one of `n` colors in Line 10, and Line 11 forbids that connected nodes are marked with the same color. In Line 14–15, stable models are projected onto atoms over predicate `mark`/2.

Unlike with one-shot ASP solving, we do not combine the encoding in Listing 1 with fixed facts, but rather aim at a selective addition as well as withdrawal of atoms over `edge`/2. In order to prepare reactive ASP rules for this purpose, the instructions in Listing 2 can be fed into the *quontroller*. They are then mapped to the language of *oclingo* as shown in Listing 3. The resulting reactive ASP rules are divided into a static **`#base`** part (Line 1–13), a stepwise **`#cumulative`** part (Line 15–25), and a stepwise **`#volatile`** part (Line 27–34). The stepwise parts are instantiated for successive step numbers replacing the constant `t`, where instances of **`#cumulative`** rules are gathered over steps and **`#volatile`** ones are discarded when progressing to the next step.

In more detail, the **`#domain`** instructions in Line 2 and 3 of Listing 2 are mapped to the static rules in Line 4 and 7 of Listing 3. They define the (*quontroller*-internal) predicates `_domain_edge`/2 and `_domain_mark`/2, which provide the domains of instances of `edge`/2 and `mark`/2 that can be asserted by query programs. In particular, the rule in Line 4 expresses that edges may connect distinct nodes with labels running from `1` to `4`, and the rule in Line 7 declares that each node can be marked with the colors provided by `color`/1. Furthermore, the instruction in Line 4 of Listing 2 is mapped to the static choice rule in Line 10 of Listing 3. This rule compensates for the lack of facts over

[1] To be pronounced 'cointreau'-ler; URL: `potassco.sourceforge.net/labs.html`.

```
1  % n colors
2  #const n = 3.
3  color(1..n).

5  % extract nodes from edges
6  node(X) :- edge(X,_).
7  node(X) :- edge(_,X).

9  % generate n-coloring
10 1 { mark(X,C) : color(C) } 1 :- node(X).
11 :- edge(X,Y), mark(X;Y,C).

13 % display n-coloring
14 #hide.
15 #show mark/2.
```

Listing 1: ASP encoding of $n$-coloring (`encoding.lp`)

```
1  #setup.
2  #domain edge(X,Y) : X := 1..4, Y := 1..4, X != Y.
3  #domain mark(X,C) : X := 1..4, color(C).
4  #choose edge/2.
5  #define edge/2.
6  #query mark/2.
7  #show edge/2.
8  #endsetup.
```

Listing 2: *quontroller* setup instructions (`setup.ini`)

`edge`/2 and allows for instantiating the original encoding in Listing 1 relative to the domain given by `_domain_edge`/2. Again relying on `_domain_edge`/2, the instruction in Line 7 of Listing 2 is mapped to the **`#show`** statement in Line 13 of Listing 3 for including atoms over `edge`/2 (in addition to those over `mark`/2) in output projections.

The **`#cumulative`** and **`#volatile`** parts in Listing 3 deal with the instructions in Line 5 and 6 of Listing 2. In particular, the **`#external`** statements in Line 18 and 23 of Listing 3 declare (*quontroller*-internal) instances of `_assert_mark(X1,X2,t)` and `_assert_edge(X1,X2,t)` as potential online inputs from query programs, where `X1` and `X2` are instantiated relative to domains given by static rules and the constant `t` is added as an argument to distinguish separate queries. The rules in Line 19–20 and 24–25 of Listing 3 further define the (*quontroller*-internal) predicates `_derive_mark`/3 and `_derive_edge`/3 for indicating "active" assertions from query programs. Given that such assertions may remain active over several queries, the rules defining `_derive_mark`/3 and `_derive_edge`/3 include one case for reflecting current assertions (Line 19 and 24) and another for passing on former assertions (Line 20 and 25). As a consequence, instances of `_derive_mark(X1,X2,t)` and `_derive_edge(X1,X2,t)` capture active assertions regarding the original predicates

```
1  #base.
2
3  % #domain edge(X,Y) : X := 1..4, Y := 1..4, X != Y.
4  _domain_edge(X,Y) :- X := 1..4, Y := 1..4, X != Y.
5
6  % #domain mark(X,C) : X := 1..4, color(C).
7  _domain_mark(X,C) :- X := 1..4, color(C).
8
9  % #choose edge/2.
10 { edge(X1,X2) : _domain_edge(X1,X2) }.
11
12 % #show edge/2.
13 #show edge(X1,X2) : _domain_edge(X1,X2).
14
15 #cumulative t.
16
17 % #query mark/2.
18 #external _assert_mark(X1,X2,t) : _domain_mark(X1,X2).
19 _derive_mark(X1,X2,t) :- _domain_mark(X1,X2), _assert_mark(X1,X2,t).
20 _derive_mark(X1,X2,t) :- _domain_mark(X1,X2), _derive_mark(X1,X2,t-1).
21
22 % #define edge/2.
23 #external _assert_edge(X1,X2,t) : _domain_edge(X1,X2).
24 _derive_edge(X1,X2,t) :- _domain_edge(X1,X2), _assert_edge(X1,X2,t).
25 _derive_edge(X1,X2,t) :- _domain_edge(X1,X2), _derive_edge(X1,X2,t-1).
26
27 #volatile t.
28
29 % #query mark/2.
30 :- _domain_mark(X1,X2), _derive_mark(X1,X2,t), not mark(X1,X2).
31
32 % #define edge/2.
33 :- _domain_edge(X1,X2), _derive_edge(X1,X2,t), not edge(X1,X2).
34 :- _domain_edge(X1,X2), edge(X1,X2), not _derive_edge(X1,X2,t).
```

Listing 3: Mapping of *quontroller* setup instructions in `setup.ini` to reactive ASP rules

`mark`/2 and `edge`/2, and corresponding matches are established via **`#volatile`** integrity constraints. For one, the **`#query`** instruction in Line 6 of Listing 2 is mapped to the integrity constraint in Line 30 of Listing 3, thus requiring `mark(X1,X2)` to hold at any step where `_derive_mark(X1,X2,t)` indicates an active assertion, and the analogous integrity constraint in Line 33 is obtained in view of the **`#define`** instruction in Line 5 of Listing 2. The latter is complemented by another integrity constraint in Line 34, denying `edge(X1,X2)` to hold when `_derive_edge(X1,X2,t)` does not indicate any active assertion. That is, a **`#query`** instruction expresses that assertions may require atoms to belong to stable models of interest, and a **`#define`** instruction is stronger by additionally claiming some active assertion for atoms to hold.

After launching *oclingo* with the encoding in Listing 1 and the reactive ASP rules in Listing 3 (via '`quontroller.py -o encoding.lp -c setup.ini`'), the *quontroller* is ready to process query programs provided by a user. An exemplary stream of query programs is shown in Figure 1(a), and Figure 1(b) provides its counterpart in the syntax of *oclingo*'s basic front-end. In fact, the *quontroller* maps query programs to available stream constructs and automatically performs replacements for interacting with reactive ASP rules. To begin with, the keywords '**`#query`**.' and '**`#endquery`**.', which encapsulate individual query programs, are mapped to '**`#step`** `q : 0.` **`#forget`** `q-1`.' and '**`#endstep`**.', where `q` is the sequence number

```
1  #query.
2  #assert : e(1).
3  edge(1,2).
4  edge(1,3).
5  edge(2,3).
6  edge(2,4).
7  edge(3,4).
8  #endquery.

10 #query.
11 #assert.
12 mark(1,1).
13 #endquery.

15 #query.
16 #assert : e(2).
17 edge(1,4).
18 #endquery.

20 #query.
21 #retract : e(2).
22 #assert.
23 mark(1,1).
24 mark(2,2).
25 #endquery.

27 #query.
28 #retract : e(1).
29 #endquery.

31 #stop.
```

(a) *quontroller* query stream

```
1  #step 1 : 0.  #forget 0.
2  #assert : e(1).
3  _assert_edge(1,2,1).
4  _assert_edge(1,3,1).
5  _assert_edge(2,3,1).
6  _assert_edge(2,4,1).
7  _assert_edge(3,4,1).
8  #endstep.

10 #step 2 : 0.  #forget 1.
11 #volatile : 1.
12 _assert_mark(1,1,2).
13 #endstep.

15 #step 3 : 0.  #forget 2.
16 #assert : e(2).
17 _assert_edge(1,4,3).
18 #endstep.

20 #step 4 : 0.  #forget 3.
21 #retract : e(2).
22 #volatile : 1.
23 _assert_mark(1,1,4).
24 _assert_mark(2,2,4).
25 #endstep.

27 #step 5 : 0.  #forget 4.
28 #retract : e(1).
29 #endstep.

31 #stop.
```

(b) Mapping of *quontroller* query stream in (a)

Fig. 1: A *quontroller* query stream and its mapping to a reactive ASP online progression

of a query program and the **#forget** directive enables simplifications of reactive ASP rules for yet undefined **#external** atoms introduced at step `q-1`. Also note that '`: 0`' in **#step** directives tells *oclingo* not to increment the step counter on unsatisfiability.

Each query program may include labeled assertions, as declared via '**#assert** `: e(1).`' in Line 2 of Figure 1(a) and 1(b). Such a construct expresses that subsequently provided rules remain active until the labeled assertion is explicitly retracted. In view of its reactive ASP rules, the *quontroller* however replaces each head atom `p(...)` of a rule (or fact) to assert by its internal representation `_assert_p(...,q)`, where `q` is again the sequence number of the query program at hand. For instance, '`edge(1,2).`' is mapped to '`_assert_edge(1,2,1).`' in Line 3 of Figure 1(a) and 1(b). By means of reactive ASP rules capturing the **#define**

instruction in Line 5 of Listing 2, the instances of `_assert_edge`/3 provided by facts in Line 3–7 of Figure 1(b) are matched with the original atoms over `edge`/2. As a consequence, the first query program yields six stable models, in which the unconnected nodes `1` and `4` share one of the colors `1`, `2`, or `3` and the nodes `2` and `3` are marked with distinct remaining colors.

The second query program in Line 10–13 of Figure 1(a) includes '`mark(1,1).`' as an unlabeled assertion, indicated by the keyword '**`#assert.`**' The latter is mapped to the stream construct '**`#volatile`** `: 1.`' (cf. Line 11 of Figure 1(b)), meaning that the internal representation '`_assert_mark(1,1,2).`' of the assertion expires "automatically" in the next step. Technically, such expiration is implemented by adding assumption literals to the bodies of transient rules, i.e. '`_assert_mark(1,1,2).`' is internally turned into '`_assert_mark(1,1,2) :- _expire(3).`' and `_expire(3)` holds up to step `3` where it is then permanently falsified. However, in the second step, the assertion of color `1` for node `1` leads to two stable models of interest among the six obtained in the first step. Also note that the *quontroller* language includes '**`#assert.`**' in order to indicate the beginning of query parts in which head atoms are replaced by internal representations, so that any rules to be left untouched can still be provided beforehand.

Summarizing the remaining query programs, the labeled assertion `e(2)` in the third query program turns the graph represented by atoms over `edge`/2 into a clique of four nodes, so that no stable model is obtained in the third step. Hence, `e(2)` is retracted in the fourth step (by discharging an assumption literal associated with `e(2)`), and the additional unlabeled assertion of colors for the nodes `1` and `2` leads to a single stable model of interest. Note that '`mark(1,1).`' is turned into '`_assert_mark(1,1,4).`' in Line 23 of Figure 1(b), while '`_assert_mark(1,1,2).`' has been used in Line 12. The rewriting by the *quontroller* thus avoids a clash with *oclingo*'s modularity requirement and enables repeated assertions of the "same" atom (in separate query programs). Finally, the empty (projection of a) stable model is obtained after retracting all instances of `_assert_edge`/3 in the last query program, and '**`#stop.`**' afterwards signals the end of the query stream to the *quontroller*.

## 3 Discussion

We presented a simple yet effective extension of reactive ASP that allows for interactive query answering and theory exploration with ASP. This was accomplished by means of a mapping scheme between queries and reactive ASP rules along with the assumption-based solving capacities of *oclingo*. With it, programs can be temporarily added to the solving process, either for an initially limited number of interactions or until they are interactively withdrawn again. A typical use case of limited program parts are integrity constraints, representing queries automatically vanishing after having been posed. Unlike this, an assertion allows, for instance, for exploring the underlying domain under user-defined hypotheses. All subsequent solving processes then include the asserted information until it is retracted by the user. The possibility of reusing ground rules as well as recorded conflict information over a sequence of queries distinguishes reactive ASP from ordinary one-shot reasoning methods. As future work, we want to study the performance of query answering with the *quontroller* on challenging benchmark problems.

*Acknowledgments* This work was partially funded by DFG grant SCHA 550/9-1. We are grateful to the anonymous reviewers for their suggestions.

## References

1. Clocksin, W., Mellish, C.: Programming in Prolog. Springer (1981)
2. Lloyd, J.: Foundations of Logic Programming. Springer (1987)
3. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. Proc. ICLP, MIT (1988) 1070–1080
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool (2012)
5. Baral, C., Chancellor, K., Tran, N., Tran, N., Joy, A., Berens, M.: A knowledge based approach for representing and reasoning about signaling networks. Proc. ISMB, (2004) 15–22
6. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. Proc. AAAI, AAAI (2008) 436–441
7. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. TPLP Journal **11**(2-3) (2011) 323–360
8. Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., Veber, P.: Repair and prediction (under inconsistency) in large biological networks with answer set programming. Proc. KR, AAAI (2010) 497–507
9. Ray, O., Whelan, K., King, R.: Logic-based steady-state analysis and revision of metabolic networks with inhibition. Proc. CISIS, IEEE (2010) 661–666
10. Videla, S., Guziolowski, C., Eduati, F., Thiele, S., Grabe, N., Saez-Rodriguez, J., Siegel, A.: Revisiting the training of logic models of protein signaling networks with ASP. Proc. CMSB, Springer (2012) 342–361
11. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. Proc. LPNMR, Springer (2011) 54–66
12. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Preliminary report. Proc. KR, AAAI (2012) 613–617

# Generating Shortest Synchronizing Sequences using Answer Set Programming

Canan Güniçen, Esra Erdem, and Hüsnü Yenigün

Sabanci University, Orhanli Tuzla, Istanbul 34956, Turkey
{cananguniçen,esraerdem,yenigun}@sabanciuniv.edu

**Abstract.** For a finite state automaton, a synchronizing sequence is an input sequence that takes all the states to the same state. Checking the existence of a synchronizing sequence and finding a synchronizing sequence, if one exists, can be performed in polynomial time. However, the problem of finding a shortest synchronizing sequence is known to be NP-hard. In this work, the usefulness of Answer Set Programming to solve this optimization problem is investigated, in comparison with brute-force algorithms and SAT-based approaches.

**Keywords:** finite automata, shortest synchronizing sequence, ASP

## 1 Introduction

For a state based system that reacts to events from its environment by changing its state, a synchronizing sequence is a specific sequence of events that brings the system to a particular state no matter where the system initially is. Synchronizing sequences have found applications in many practical settings. In model based testing, it can be used to bring the unknown initial state of an implementation to a specific state to start testing [1, 2]. As Natarajan [3] and Eppstein [4] explain, it can be used to orient a part to a certain position on a conveyor belt. Another interesting application is from biocomputing, where one can use a DNA molecule encoding a synchronizing sequence to bring huge number of identical automata (in the order of $10^{12}$ automata/$\mu l$) to a certain restart state [5].

Such a state based system can be formalized as a *finite state automaton* (FA). We restrict ourselves to *deterministic* FA, which is defined as a tuple $A = (Q, \Sigma, \delta)$, where $Q$ is a finite set of states, $\Sigma$ is a finite input alphabet, and $\delta : Q \times \Sigma \mapsto Q$ is a transition function, defining how each state of $A$ is changed by the application of inputs. The transition function $\delta$ is extended to words in $\Sigma^\star$ naturally as $\delta(q, \varepsilon) = q$, $\delta(q, wx) = \delta(\delta(q, w), x)$, where $q \in Q$, $w \in \Sigma^\star$, $x \in \Sigma$, and $\varepsilon$ is the empty word. A FA $A = (Q, \Sigma, \delta)$ is called *completely specified* when $\delta$ is a total function. We will only consider completely specified FA in this work. Figure 1 is an example of a FA.

We can now define a synchronizing sequence formally. Given an FA $A = (Q, \Sigma, \delta)$, an input sequence $w \in \Sigma^\star$ is called a *synchronizing sequence* for $A$ if $\forall q, q' \in Q$, $\delta(q, w) = \delta(q', w)$. As an example, *baab* is a synchronizing sequence for $A_1$ given in Figure 1.

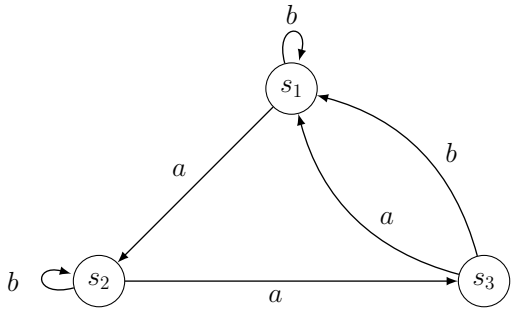



Fig. 1: An example FA $A_1$

Synchronizing sequences attracted much attention from a theoretical point of view as well. In the literature, a synchronizing sequence is also referred to as a synchronizing word, reset sequence, or a reset word. Not every FA has a synchronizing sequence, and one can check the existence of a synchronizing sequence for a given FA in polynomial time. On the other hand, the problem of finding a shortest synchronizing sequence is known to be NP-hard [4]. For this reason, several heuristic approaches have been suggested to compute short synchronizing sequences [4, 6–8]. These algorithms guarantee a synchronizing sequence of length $O(n^3)$ where $n$ is the number of states in the FA. The best known upper bound is $n(7n^2+6n-16)/48$ [9]. However, it has been conjectured by Černý almost half a century ago that this upper bound is $(n-1)^2$ [10, 11] after providing a class of FA with $n$ states whose shortest synchronizing sequence is of length $(n-1)^2$. The conjecture is shown to hold for certain classes of automata [4, 5, 12–15]. However, the conjecture is still open in general, and it is one of the oldest open problems of finite state automata theory.

Despite the fact that it is NP-hard, considering the computation of shortest synchronizing sequences is still useful. Such attempts are valuable both for understanding the characteristics of shortest synchronizing sequence problem (see e.g. [16]) and for forming a base line for the performance evaluation of heuristics for computing short synchronizing sequences.

In this work, we formulate the problem of computing a shortest synchronizing sequence in Answer Set Programming (ASP) [17, 18]—a knowledge representation and reasoning paradigm with an expressive formalism and efficient solvers. The idea of ASP is to formalize a given problem as a "program" and to solve the problem by computing models (called "answer sets" [19]) of the program using "ASP solvers", such as Clasp [20].

After we represent the shortest synchronizing sequence problem in ASP, we experimentally evaluate the performance and effectiveness of ASP, in comparison with two other approaches, one based on SAT [16] and the other on a brute-force algorithm [21]. For our experiments with the SAT-based approach, we extend the SAT formulation of the existence of a synchronizing sequence of a given length [16], to FA with more than two input symbols.

The rest of the paper is organized as follows. In Section 2 we present four different ASP formulations for the problem. An existing SAT formulation [16] is extended to FAs with more than two inputs in Section 3. The experimental results are given in Section 4 to compare the approaches. Finally, in Section 5 we give concluding remarks and some future research directions.

## 2 ASP Formulations of the Shortest Synchronizing Sequence Problem

Let us first consider the decision version of the shortest synchronizing sequence problem: For an FA $A = (Q, \Sigma, \delta)$ and a positive integer constant $c$, decide whether $A$ has a synchronizing word $w$ of length $c$.

Without loss of generality, we represent states and input symbols of an FA $A = (Q, \Sigma, \delta)$, by the range of numbers $1..n$ and $1..k$ ($n = |Q|$, $k = |\Sigma|$), respectively. Then an FA $A = (Q, \Sigma, \delta)$ can be described in ASP by three forms of atoms given below:

- $state(s)$ $(1 \leq s \leq n)$ describing the states in $Q$,
- $symbol(j)$ $(1 \leq j \leq k)$ describing the input symbols in $\Sigma$, and
- $transition(s, j, s')$ $(1 \leq s, s' \leq n,\ 1 \leq j \leq k)$ describing the transitions $\delta(s, j) = s'$.

We represent possible lengths $i$ of sequences by atoms of the form $step(i)$ $(1 \leq i \leq c)$.

A synchronizing sequence of length $c$ is characterized by atoms of the form $synchro(i, x)$ $(1 \leq i \leq c,\ 1 \leq x \leq k)$ describing that the $i$'th symbol of the word is $x$.

Using these atoms, we can represent the decision version of the shortest synchronizing sequence problem with a "generate-and-test" methodology used in various ASP formulations. In the following, we present two different ASP formulations based on this approach.

In these ASP formulations, we use an auxiliary concept of a *path in $A$ characterized by a sequence $w_1, w_2, \ldots, w_x$ of symbols in $\Sigma$*, which is defined as a sequence $q_1, q_2, \ldots, q_{x+1}$ of states in $Q$ such that $\delta(q_i, w_i) = q_{i+1}$ for every $i$ $(1 \leq i \leq x)$. The existence of such a path of length $i$ in $A$ from a state $s$ to a state $q$ (i.e., the reachability of a state $q$ from a state $s$ by a path of length $i$ in $A$) characterized by the first $i$ symbols of a word $w$ is represented by atoms of the form $path(s, i+1, q)$ defined as follows:

$$\begin{aligned} &path(s, 1, s) \leftarrow state(s) \\ &path(s, i+1, q) \leftarrow path(s, i, r), synchro(i, x), transition(r, x, q), \\ &\qquad state(s), state(r), state(q), symbol(x), step(i) \end{aligned} \tag{1}$$

### 2.1 Connecting All States to a Sink State

In the first ASP formulation, which we call $ASP_1$, first we "generate" a sequence $w$ of $c$ symbols by the following choice rule:

$$1\{synchro(i,j) : symbol(j)\}1 \leftarrow step(i) \tag{2}$$

where $step(i)$ is defined by a set of facts:

$$step(i) \leftarrow \qquad (1 \leq i \leq c) \tag{3}$$

Next, we ensure that it is a synchronizing sequence by "testing" that it does not violate the condition:

$C_1$ There exists a sink state $f \in Q$ such that every path in $A$ characterized by $w$ ends at $f$.

by adding the following constraints

$$\leftarrow sink(f), not\ path(s, c+1, f), state(s), state(f) \tag{4}$$

where $sink(f)$ describes a sink state:

$$1\{sink(f) : state(f)\}1 \leftarrow . \tag{5}$$

The union of the program $ASP_1$ that consists of the rules (2), (3), (4), (5), (1), with a set of facts describing an FA $A$ has an answer set iff there exists a synchronizing sequence of length $c$ for $A$.

### 2.2 Merging States Pairwise

In the second ASP formulation, which we call $ASP_2$, first we "generate" a sequence $w$ of $c$ symbols by the choice rule (2).

Next, we ensure that it is a synchronizing sequence by "testing" that it does not violate the following condition, instead of constraint $C_1$:

$D_1$ For every pair $q_i, q_{i+1}$ of states in $Q = \{q_1, q_2, \ldots, q_n\}$, $\delta(q_i, w) = \delta(q_{i+1}, w)$.

by adding the following cardinality constraints

$$\leftarrow 1\{merged(r) : state(r), r < n\}n - 2 \tag{6}$$

where $merged(r)$ describes that there exists a state $s$ reachable from the states $r$ and $r+1$ by paths characterized by the first $i$ symbols of $w$ for some $i$ $(1 \leq i \leq c)$:

$$\begin{aligned} merged(r) \leftarrow\ & path(r, i, s), path(r+1, i, s), \\ & state(s), state(r), state(r+1), step(i) \end{aligned} \tag{7}$$

The union of the program $ASP_2$ that consists of the rules (2), (3), (6), (7), (1), with a set of facts describing an FA $A$ has an answer set iff there exists a synchronizing sequence of length $c$ for $A$.

### 2.3 Optimization

The ASP formulations given in Section 2.1 and Section 2.2, with a set of facts describing an FA $A$, have answer sets if the given FA $A$ has a synchronizing sequence of length $c$. In order to find the length of the shortest synchronizing sequence, one can perform a binary search on possible values of $c$.

In this section, we present another ASP formulation where we let the ASP solver first decide the length $l$ of a shortest synchronizing sequence, where $l \leq c$:

$$1\{shortest(l) : 1 \leq l \leq c\}1 \leftarrow \tag{8}$$

and declare possible lengths of sequences:

$$step(j) \leftarrow shortest(i) \qquad (1 \leq j \leq i \leq c). \tag{9}$$

Next, we ensure that $l$ is indeed the optimal value, by the following optimization statement

$$\#minimize[shortest(l) = l] \leftarrow \tag{10}$$

We denote by $ASP_1^{opt}$ (resp. $ASP_2^{opt}$) the ASP formulation obtained from $ASP_1$ (resp. $ASP_2$) by adding the rules (8) and (10), and replacing the rules (3) by the rules (9). If $ASP_1^{opt}$ (resp. $ASP_2^{opt}$) with a set of facts describing an FA $A$ has an answer set $X$ then $X$ characterizes a shortest synchronizing sequence for $A$.

## 3 SAT Formulation of the Shortest Synchronizing Sequence Problem

In [16], a SAT formulation of the problem of checking if an FA $A$ has a synchronizing sequence of a certain length is presented. However, this formulation is given only for FA with two input symbols. We extend this SAT formulation to FA with any number of input symbols as follows.

We first define a boolean operator $\nabla$ that will simplify the description of our SAT formulation. For a given set of boolean variables $\{r_1, r_2, \ldots, r_k\}$, we define $\nabla\{r_1, r_2, \cdots, r_k\}$ as follows:

$$\nabla\ \{r_1,r_2,\cdots,r_k\} \equiv ((r_1 \Rightarrow (\ \neg\ r_2 \wedge \neg\ r_3 \wedge \cdots \wedge \neg\ r_k)) \wedge (r_2 \Rightarrow (\ \neg\ r_1 \wedge \neg\ r_3 \wedge \cdots \wedge \neg\ r_k)) \wedge \cdots (r_k \Rightarrow (\ \neg\ r_1 \wedge \neg\ r_2 \wedge \cdots \wedge \neg\ r_{k-1})) \wedge (r_1 \vee r_2 \vee \cdots \vee r_k\ ))$$

Intuitively, $\nabla\{r_1, r_2, \cdots, r_k\}$ is true with respect to an interpretation $I$ iff exactly one of the variables $r_i$ is true and all the others are false with respect to $I$.

Checking the existence of a synchronizing sequence of length $c$ is converted into a SAT problem by considering the following boolean formulae. Below we use the notation $[c]$ to denote the set $\{1, 2, \ldots, c\}$.

– $F_1$: An input sequence of length $c$ has to be created. At each step of this input sequence, there should be exactly one input symbol being used. For this purpose, we generate Boolean variables $X_{l,x}$ which should be set to true (by an interpretation) only if at step $l$ the input symbol $x$ is used. The following formulae make sure that only one input symbol is picked for each step $l$.

$$\sigma_1 = \bigwedge_{l \in [c]} (\nabla \{X_{l,x} | x \in X\})$$

– $F_2$: Similar to what we accomplish in ASP formulations by atoms of the form $path(s, i, q)$, we need to trace the state reached when the input sequence guessed by formula $\sigma_1$ is applied. For this purpose, boolean variables $S_{i,j,k}$ (which we call *state tracing variables*) are created which are set to true (by an interpretation) only if we are at state $q_k$ at step $j$ when we start from state $q_i$. We first make sure that for each starting state and at any step, there will always be exactly one current state.

$$\sigma_2 = \bigwedge_{i \in [n], l \in [c]} (\nabla \{S_{i,l,j} | j \in n\})$$

– $F_3$: The initial configuration of the FA $A$ must be indicated. For this purpose state tracing variables should be initialized for their first step.

$$\sigma_3 = \bigwedge_{i \in [n]} (S_{i,1,i})$$

– $F_4$: Again, similar to the constraints in ASP formulations, over atoms of the form $path(s, i, q)$, we have the corresponding SAT formulae to make sure that state tracing variables are assigned according to the transitions of the FA $A$. For each state $q_j$ and input $x$ of $A$, if we have $\delta(q_j, x) = q_k$ in $A$, then we generate the following formulae:

$$\sigma_4 = \bigwedge_{i,j \in [n], l \in [c], x \in X} ((S_{i,l,j} \bigwedge X_{l,x}) \Rightarrow S_{i,l+1,k})$$

– $F_5$: A synchronizing sequence $w$ merges all the states at a sink state after the application of $w$. We use boolean variable $Y_i$ to pick a sink state. Since only one of the states has to be a sink state, we introduce the following formulae:

$$\sigma_5 = (\nabla \{Y_i | i \in [n]\})$$

– $F_6$: Finally, we need to make sure that all the states reach the sink state picked by $F_5$ at the end of the last step after the application of the synchronizing sequence guessed by formulae $F_1$.

$$\sigma_6 = \bigwedge_{i,j \in [n]} (Y_i \Rightarrow S_{j,c+1,i})$$

The conjunction of all formulae introduced above is a Boolean formula that is satisfiable iff there exists a synchronizing sequence of FA $A$ of length $c$.

## 4 Experimental Study

In this section, we present the experimental study carried out to compare the performance of the ASP formulations, the SAT formulation, and the brute–force algorithm for generating a shortest synchronizing sequence.

We first present our experiments using finite automata that are generated randomly. Given the number of states and the number of input symbols, an FA is generated by assigning the next state of each transition randomly. If the FA generated in this way does not have a synchronizing sequence, then it is discarded. Otherwise, it is included in the set of FAs used in our experiments. We generated 100 random FAs this way for each number of states we used in the experiments (except for the biggest set of tests with 50 states and 4-6 input symbols, where we use only 50 FAs to speed up the experiments).

The implementation of the brute–force algorithm in the tool COMPAS [21] is used. The brute–force algorithm could be used for FAs with up to 27 states. Beyond this number of states, COMPAS could not complete the computation due to memory restrictions.

We implemented tools that create ASP and SAT formulations from a given FA and an integer constant $c$ as explained in Section 2, Section 3, and also the SAT formulation given in [16] for FAs with two inputs only.

In the results given below, the formulations $ASP_1$, $ASP_2$, $ASP_1^{opt}$, and $ASP_2^{opt}$ refer to the ASP formulations given in Section 2. $SAT_1$ and $SAT_2$ refer to the SAT formulations given in [16] and Section 3, respectively. $BF$ refers to the brute–force algorithm.

Note that the ASP formulations $ASP_1^{opt}$ and $ASP_2^{opt}$ report the length of a shortest synchronizing sequence, provided that the constant $c$ given is not smaller than the length of a shortest synchronizing sequence. When $c$ is not big enough, another experiment is performed by doubling the value of $c$. We report only the results from successful $ASP_1^{opt}$ and $ASP_2^{opt}$ experiments, where a sufficiently large $c$ is given. An experimental study is presented in [16] where the length of the shortest synchronizing sequence is reported to be around $2\sqrt{n}$ on the average for an $n$ state automaton with two input symbols. We therefore initially take the value of $c$ as $2\sqrt{n}$.

On the other hand, the ASP formulations $ASP_1$ and $ASP_2$, and also the SAT formulations $SAT_1$ and $SAT_2$, only report if a synchronizing sequence of length $c$ exists or not. Therefore, one has to try several $c$ values to find the length of the shortest synchronizing sequence using these formulations. In our experiments with these formulations, we find the length of a shortest synchronizing sequence by applying a binary search on the value of $c$, by using a script that invokes the ASP solver for each attempt on a possible value of $c$ separately. We similarly take the initial value of $c$ to be $2\sqrt{n}$ as explained above. The time reported is the total time taken by all the attempts until the length of a shortest synchronizing sequence is found. The memory reported is the average memory usage in these attempts.

The experiments are carried out using MiniSat 2.2.0 [22] and Clingo 3.0.3 [23] running on Ubuntu Linux where the hardware is a 2.4Ghz Intel Core-i3 machine.

Table 1: Experiments on FAs with 2 input symbols (time - secs)

| $n$ | $ASP_1$ | $ASP_2$ | $ASP_1^{opt}$ | $ASP_2^{opt}$ | $SAT_1$ | $BF$ |
|---|---|---|---|---|---|---|
| 5 | 0 | 0 | 0 | 0 | 7 | 0 |
| 10 | 2 | 2 | 2 | 11 | 10 | 0 |
| 15 | 1 | 2 | 2 | 5 | 10 | 0 |
| 20 | 4 | 6 | 5 | 8 | 12 | 4 |
| 25 | 6 | 12 | 7 | 14 | 13 | 73 |
| 26 | 7 | 11 | 8 | 15 | 14 | 145 |
| 27 | 9 | 12 | 8 | 14 | 15 | 289 |

Table 2: Experiments on FAs with 2 input symbols (memory - kBytes)

| $n$ | $ASP_1$ | $ASP_2$ | $ASP_1^{opt}$ | $ASP_2^{opt}$ | $SAT_1$ |
|---|---|---|---|---|---|
| 5 | 7750 | 7731 | 7622 | 7620 | 7677 |
| 10 | 8169 | 7278 | 8154 | 8160 | 7983 |
| 15 | 6284 | 6566 | 6465 | 6300 | 7847 |
| 20 | 6909 | 6911 | 6943 | 6947 | 7810 |
| 25 | 6769 | 6775 | 6937 | 6779 | 8066 |
| 26 | 7151 | 6798 | 7136 | 6822 | 8213 |
| 27 | 7123 | 7106 | 6744 | 7119 | 8113 |

In Table 1 and Table 2, the time and the memory performance of the formulations $ASP_1$, $ASP_2$, $SAT_1$, and the brute–force algorithm are given. We could not get a report on the memory usage of COMPAS for the brute–force algorithm, hence no data is given for the brute–force algorithm in Table 2. In this set of experiments, the number of states $n$ ranges between 5 and 27, and the number of input symbols is fixed to 2.

In Table 3 and Table 4, the time and the memory performance of the formulations $ASP_1^{opt}$, $ASP_2^{opt}$, and $SAT_2$ are given on FAs with the number of states $n \in \{30, 40, 50\}$ and the number of inputs $k \in \{2, 4, 6\}$.

Table 1 shows that the brute–force approach uses much more time than the other approaches, especially as the size of the FA gets bigger. Therefore, after a certain threshold size, the brute–force approach is not an alternative.

By investigating the results given in Table 1 and Table 3, one can see that the ASP formulation approach of $ASP_1$ and $ASP_1^{opt}$ perform better than $ASP_2$ and $ASP_2^{opt}$, in general. This may be due to that the number of ground instances of (4) and (5) is smaller than that of (6) and (7). However, after intelligent grounding of Clingo, the program sizes of $ASP_1^{opt}$ and $ASP_2^{opt}$ become comparable. On the other hand, we have observed that $ASP_2^{opt}$ leads to more backtracking and restarts compared to $ASP_1^{opt}$. For example, for an instance of 50 states and 2 input symbols, $ASP_1^{opt}$ leads to 82878 choices and no restarts, whereas $ASP_2^{opt}$ leads to 137276 choices and 4 restarts. This may be due to that, in $ASP_1^{opt}$with respect to (4) and (5) , once a sink node is selected, for every state,

Table 3: Experiments on FAs with different number of inputs (time - secs)

| $n$ | $k$ | $ASP_1$ | $ASP_2$ | $ASP_1^{opt}$ | $ASP_2^{opt}$ | $SAT_2$ |
|---|---|---|---|---|---|---|
| 30 | 2 | 4 | 17 | 4 | 18 | 49 |
| 30 | 4 | 66 | 80 | 45 | 57 | 101 |
| 30 | 6 | 208 | 405 | 160 | 231 | 490 |
| 40 | 2 | 33 | 45 | 71 | 122 | 222 |
| 40 | 4 | 348 | 380 | 244 | 311 | 472 |
| 40 | 6 | 1158 | 1400 | 707 | 980 | 2133 |
| 50 | 2 | 93 | 120 | 117 | 146 | 430 |
| 50 | 4 | 902 | 1101 | 835 | 833 | 2975 |
| 50 | 6 | 3205 | 4010 | 2705 | 3032 | 7492 |

Table 4: Experiments on FAs with different number of inputs (memory - kBytes)

| $n$ | $k$ | $ASP_1$ | $ASP_2$ | $ASP_1^{opt}$ | $ASP_2^{opt}$ | $SAT_2$ |
|---|---|---|---|---|---|---|
| 30 | 2 | 6063 | 7143 | 4973 | 6140 | 42764 |
| 30 | 4 | 7309 | 7438 | 5278 | 7457 | 49681 |
| 30 | 6 | 7735 | 7621 | 7496 | 10457 | 52250 |
| 40 | 2 | 6709 | 6029 | 5616 | 7448 | 67362 |
| 40 | 4 | 7050 | 7073 | 7880 | 8550 | 78697 |
| 40 | 6 | 7764 | 8024 | 7983 | 10234 | 84671 |
| 50 | 2 | 7222 | 8336 | 8965 | 16072 | 86453 |
| 50 | 4 | 8438 | 9056 | 9931 | 12843 | 85157 |
| 50 | 6 | 8773 | 9228 | 10729 | 14092 | 93118 |

existence of a path of a fixed length is checked; on the other hand, in $ASP_2^{opt}$with respect to (6) and (7), for every state, existence of two paths of the same length is checked, which may intuitively lead to more backtracking and restarts. On the other hand, the memory performances of all ASP approaches are similar, as displayed by Table 2 and Table 4.

As for the comparison of the ASP and SAT approaches, one can see that the ASP approaches are both faster and uses less memory than the SAT approach, in general. However, the ASP approach seems to have a faster increase in the running time compared to the SAT approach. This trend needs to be confirmed by further experiments.

We also experimented with finite state automata from MCNC'91 benchmarks [24]. We used only those finite state machine examples in this benchmark set that correspond to completely specified finite state automata. The results of these experiments are given in Table 5 for the time comparison. We obtained similar results to what we have observed in our experiments on random finite state automata. The time performance of the ASP approaches are better than the SAT approach in these experiments as well. We note that the benchmark example "dk16", which is also the automaton having the largest number of states, has the longest running time among the automata in the benchmark set.

Table 5: Experiments on FAs from MCNC benchmarks (time - msecs)

| Name | $n$ | $k$ | $ASP_1$ | $ASP_2$ | $ASP_1^{opt}$ | $ASP_2^{opt}$ | $SAT_2$ |
|---|---|---|---|---|---|---|---|
| bbtas | 6 | 4 | 15 | 18 | 10 | 14 | 52 |
| beecount | 7 | 8 | 18 | 18 | 10 | 9 | 89 |
| dk14 | 7 | 8 | 18 | 20 | 15 | 17 | 62 |
| dk15 | 4 | 8 | 13 | 14 | 7 | 7 | 20 |
| dk17 | 8 | 4 | 27 | 26 | 8 | 8 | 69 |
| dk27 | 7 | 2 | 22 | 21 | 6 | 6 | 45 |
| dk512 | 15 | 2 | 33 | 27 | 11 | 12 | 278 |
| dk16 | 27 | 4 | 191 | 231 | 132 | 127 | 21253 |
| lion9 | 9 | 4 | 91 | 138 | 36 | 137 | 449 |
| MC | 4 | 8 | 18 | 18 | 7 | 7 | 53 |

However, the running time does not depend only on the number of states. The number of input symbols and the length of the shortest synchronizing sequence would also have an effect. For the comparison of the memory used for these examples, all ASP approaches used around 6 MBytes of memory, whereas the SAT approach used minimum 6 MBytes and maximum 38 MBytes memory for these experiments.

## 5 Conclusion and Future Work

In this paper, the problem of finding a shortest synchronizing sequence for a FA is formulated in ASP. Four different ASP formulations are given. Also an extension of the SAT formulation of the same problem given in [16] is suggested.

The performance of these formulations are compared by an experimental evaluation. The ASP and SAT formulations are shown to scale better than the brute–force approach. The experiments indicate that the ASP formulations perform better than the SAT approach. However this needs to be further investigated with an extended set of experiments.

Based on the encouraging results obtained from this work, using ASP to compute some other special sequences used in finite state machine based testing can be considered as a future research direction. For example checking the existence of, and computing a Preset Distinguishing Sequence is a PSPACE–hard problem [1]. Although checking the existence and computing an Adaptive Distinguishing Sequence [1] can be performed in polynomial time, generating a minimal Adaptive Distinguishing Sequence is an NP–hard problem. These hard problems can be addressed by using ASP.

## References

1. Lee, D., Yannakakis, M.: Testing finite-state machines: State identification and verification. IEEE Trans. Comput. **43**(3) (March 1994) 306–320

2. Kohavi, Z., Jha, N.K.: Switching and Finite Automata Theory. Cambridge University Press (2010)
3. Natarajan, B.K.: An algorithmic approach to the automated design of parts orienters. In: FOCS, IEEE Computer Society (1986) 132–142
4. Eppstein, D.: Reset sequences for monotonic automata. SIAM J. Comput. **19**(3) (1990) 500–510
5. Ananichev, D.S., Volkov, M.V.: Synchronizing monotonic automata. Theor. Comput. Sci. **327**(3) (2004) 225–239
6. Trahtman, A.: Some results of implemented algorithms of synchronization. In: 10th Journees Montoises d'Inform., Liege, Belgium (2004)
7. Roman, A.: Synchronizing finite automata with short reset words. Applied Mathematics and Computation **209**(1) (2009) 125–136
8. Kudlacik, R., Roman, A., Wagner, H.: Effective synchronizing algorithms. Expert Systems with Applications **39**(14) (2012) 11746–11757
9. Trahtman, A.N.: Modifying the upper bound on the length of minimal synchronizing word. In Owe, O., Steffen, M., Telle, J.A., eds.: FCT. Volume 6914 of Lecture Notes in Computer Science., Springer (2011) 173–180
10. Černý, J.: A remark on homogeneous experiments with finite automata. Mat.-Fyz. Časopis Sloven. Akad. Vied **14** (1964) 208–216
11. Černý, J., Pirick, A., Rosenauerov, B.: On directable automata. Kybernetika **07**(4) (1971) (289)–298
12. Kari, J.: Synchronizing finite automata on eulerian digraphs. Theor. Comput. Sci. **295** (2003) 223–232
13. Ananichev, D.S., Volkov, M.V.: Synchronizing generalized monotonic automata. Theor. Comput. Sci. **330**(1) (2005) 3–13
14. Trakhtman, A.: The cernú conjecture for aperiodic automata. Discrete Mathematics & Theoretical Computer Science **9**(2) (2007)
15. Volkov, M.V.: Synchronizing automata preserving a chain of partial orders. Theor. Comput. Sci. **410**(37) (2009) 3513–3519
16. Skvortsov, E., Tipikin, E.: Experimental study of the shortest reset word of random automata. In: Proceedings of the 16th international conference on Implementation and application of automata. CIAA'11, Berlin, Heidelberg, Springer-Verlag (2011) 290–298
17. Lifschitz, V.: What is answer set programming? In: Proc. of AAAI, MIT Press (2008) 1594–1597
18. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12) (2011) 92–103
19. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
20. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. In: Proc. of LPNMR. (2007) 260–265
21. Chmiel, K., Roman, A.: Compas: a computing package for synchronization. In: Proceedings of the 15th international conference on Implementation and application of automata. CIAA'10, Berlin, Heidelberg, Springer-Verlag (2011) 79–86
22. Eén, N., Sörensson, N.: An extensible sat-solver. In: Proc. of SAT. (2003) 502–518
23. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications **24**(2) (2011) 107–124
24. Yang, S.: Logic synthesis and optimization benchmarks user guide version 3.0 (1991)

# On the Semantics of Gringo

Amelia Harrison, Vladimir Lifschitz, and Fangkai Yang

University of Texas, Austin, Texas, USA
{ameliaj, vl, fkyang}@cs.utexas.edu

**Abstract.** Input languages of answer set solvers are based on the mathematically simple concept of a stable model. But many useful constructs available in these languages, including local variables, conditional literals, and aggregates, cannot be easily explained in terms of stable models in the sense of the original definition of this concept and its straightforward generalizations. Manuals written by designers of answer set solvers usually explain such constructs using examples and informal comments that appeal to the user's intuition, without references to any precise semantics. We propose to approach the problem of defining the semantics of GRINGO programs by translating them into the language of infinitary propositional formulas. This semantics allows us to study equivalent transformations of GRINGO programs using natural deduction in infinitary propositional logic.

## 1 Introduction

In this note, Gringo is the name of the input language of the grounder GRINGO,[1] which is used as the front end in many answer set programming (ASP) systems. Several releases of GRINGO have been made public, and more may be coming in the future; accordingly, we can distinguish between several "dialects" of the language Gringo. We concentrate here on Version 4, released in March of 2013. (It differs from Version 3, described in the *User's Guide* dated October 4, 2010,[2] in several ways, including the approach to aggregates—it is modified as proposed by the ASP Standardization Working Group.[3])

The basis of Gringo is the language of logic programs with negation as failure, with the syntax and semantics defined in [6]. Our goal here is to extend that semantics to a larger subset of Gringo. Specifically, we would like to cover arithmetical functions and comparisons, conditions, and aggregates.[4]

[1] `http://potassco.sourceforge.net/.`

[2] The *User's Guide* can be downloaded from the Potassco website (Footnote 1). It is posted also at `http://www.cs.utexas.edu/users/vl/teaching/lbai/clingo_guide.pdf`.

[3] `https://www.mat.unical.it/aspcomp2013/ASPStandardization.`

[4] The subset of Gringo discussed in this note includes also constraints, disjunctive rules, and choice rules, treated along the lines of [7] and [3]. The first of these papers introduces also "classical" (or "strong") negation—a useful feature that we do not include. (Extending our semantics of Gringo to programs with classical negation

Our proposal is based on the informal and sometimes incomplete description of the language in the *User's Guide*, on the discussion of ASP programming constructs in [4], on experiments with GRINGO, and on the clarifications provided in response to our questions by its designers.

The proposed semantics uses a translation from Gringo into the language of infinitary propositional formulas—propositional formulas with infinitely long conjunctions and disjunctions. Including infinitary formulas is essential, as we will see, when conditions or aggregates use variables ranging over infinite sets (for instance, over integers).

Alternatively, the semantics of Gringo can be approached using quantified equilibrium logic [12] or its syntactic counterpart defined in [2]. This method involves translating rules into the language of first-order logic. For instance, the rule

$$p(Y) \leftarrow \mathit{count}\{X, Y : q(X, Y)\} \geq 1 \tag{1}$$

can be represented by the sentence

$$\forall y(\exists x Q(x, y) \rightarrow P(y)).$$

However, this translation is not sufficiently general. For instance, it is not clear how to represent the rule

$$\mathit{total\_hours}(N) \leftarrow \mathit{sum}\{H, C : \mathit{enroll}(C), \mathit{hours}(H, C)\} = N \tag{2}$$

from Section 3.1.10 of the Gringo 3 *User's Guide* with a first-order formula. One reason is that the aggregate *sum* is used here instead of *count*. The second difficulty is that the variable $N$ is used rather than a constant.

General aggregate expressions, as used in Gringo, can be represented by first-order formulas with generalized quantifiers.[5] The advantage of infinitary propositional formulas as the target language is that properties of these formulas, and of their stable models, are better understood. We may be able to prove, for instance, that two Gringo programs have the same stable models by observing that the corresponding infinitary formulas are equivalent in one of the natural deduction systems discussed in [8]. We give here several examples of reasoning about Gringo programs based on this idea.

The process of converting Gringo programs into infinitary propositional formulas defined in this note uses substitutions to eliminate variables. This form of grounding is quite different, of course, from the process of intelligent instantiation implemented in GRINGO and other grounders. Mathematically, it is much simpler than intelligent instantiation; as a computational procedure, it is much less efficient, not to mention the fact that sometimes it produces infinite objects. Like grounding in the original definition of a stable model [6], it is modular, in the sense that it applies to the program rule by rule, and it is applicable even if

is straightforward, using the process of eliminating classical negation in favor of additional atoms described in [7, Section 4].)

[5] Stable models of formulas with generalized quantifiers are defined by Lee and Meng [9][10][11].

the program is not safe. From this perspective, GRINGO's safety requirement is an implementation restriction.

Our description of the syntax of Gringo disregards some of the features related to representing programs as strings of ASCII characters, such as using `:-` to separate the head from the body, using semicolons, rather than parentheses, to indicate the boundaries of a conditional literal, and representing falsity (which we denote here by $\bot$) as `#false`. Since the subset of Gringo discussed in this note does not include assignments, we can disregard also the requirement that equality be represented by two characters `==`.

## 2 Syntax

We begin with a signature $\sigma$ in the sense of first-order logic that includes, among others,

(i) numerals—object constants representing all integers,
(ii) arithmetical functions—binary function constants $+$, $-$, $\times$,
(iii) comparisons—binary predicate constants $<$, $>$, $\leq$, $\geq$.

We will identify numerals with the corresponding elements of the set $\mathbf{Z}$ of integers. Object, function, and predicate symbols not listed under (i)–(iii) will be called *symbolic*. A term over $\sigma$ is *arithmetical* if it does not contain symbolic object or function constants. A ground term is *precomputed* if it does not contain arithmetical functions.

We assume that in addition to the signature, a set of symbols called *aggregate names* is specified, and that for each aggregate name $\alpha$, *the function denoted by* $\alpha$, $\widehat{\alpha}$, maps every tuple of precomputed terms to an element of $\mathbf{Z} \cup \{\infty, -\infty\}$.

**Examples.** The functions denoted by the aggregate names *count*, *max*, and *sum* are defined as follows. For any set $T$ of tuples of precomputed terms,

- $\widehat{count}(T)$ is the cardinality of $T$ if $T$ is finite, and $\infty$ otherwise;
- $\widehat{max}(T)$ is the least upper bound of the set of the integers $t_1$ over all tuples $(t_1, \dots, t_m)$ from $T$ in which $t_1$ is an integer;
- $\widehat{sum}(T)$ is the sum of the integers $t_1$ over all tuples $(t_1, \dots, t_m)$ from $T$ in which $t_1$ is a positive integer; it is $\infty$ if there are infinitely many such tuples.[6]

A *literal* is an expression of one of the forms

$$p(t_1, \dots, t_k),\ t_1 = t_2,\ \mathit{not}\ p(t_1, \dots, t_k),\ \mathit{not}\ (t_1 = t_2)$$

where $p$ is a symbolic predicate constant of arity $k$, and each $t_i$ is a term over $\sigma$, or

$$t_1 \prec t_2,\ \mathit{not}\ (t_1 \prec t_2)$$

[6] To allow negative numbers in this example, we would have to define summation for a set that contains both infinitely many positive numbers and infinitely many negative numbers. It is unclear how to do this in a natural way.

where $\prec$ is a comparison, and $t_1$, $t_2$ are arithmetical terms. A *conditional literal* is an expression of the form $H : \mathbf{L}$, where $H$ is a literal or the symbol $\bot$, and $\mathbf{L}$ is a list of literals, possibly empty. The members of $\mathbf{L}$ will be called *conditions*. If $\mathbf{L}$ is empty then we will drop the colon after $H$, so that every literal can be viewed as a conditional literal.

**Example.** If *available* and *person* are unary predicate symbols then

$$available(X) : person(X)$$

and

$$\bot : (person(X), not\ available(X))$$

are conditional literals.

An *aggregate expression* is an expression of the form

$$\alpha\{\mathbf{t} : \mathbf{L}\} \prec s$$

where $\alpha$ is an aggregate name, $\mathbf{t}$ is a list of terms, $\mathbf{L}$ is a list of literals, $\prec$ is a comparison or the symbol $=$, and $s$ is an arithmetical term.

**Example.** If *enroll* is a unary predicate symbol and *hours* is a binary predicate symbol then

$$sum\{H, C : enroll(C), hours(H, C)\} = N$$

is an aggregate expression.

A *rule* is an expression of the form

$$H_1 \mid \cdots \mid H_m \leftarrow B_1, \ldots, B_n \tag{3}$$

$(m, n \geq 0)$, where each $H_i$ is a conditional literal, and each $B_i$ is a conditional literal or an aggregate expression. A *program* is a set of rules.

If $p$ is a symbolic predicate constant of arity $k$, and $\mathbf{t}$ is a $k$-tuple of terms, then

$$\{p(\mathbf{t})\} \leftarrow B_1, \ldots, B_n$$

is shorthand for

$$p(\mathbf{t}) \mid not\ p(\mathbf{t}) \leftarrow B_1, \ldots, B_n.$$

**Example.** For any positive integer $n$,

$$\begin{array}{ll} \{p(i)\}\leftarrow & (i = 1, \ldots, n), \\ \quad \leftarrow p(X), p(Y), p(X+Y) & \end{array} \tag{4}$$

is a program.

## 3 Semantics

We will define the semantics of Gringo using a syntactic transformation $\tau$. It converts Gringo rules into infinitary propositional combinations of atoms of the form $p(\mathbf{t})$, where $p$ is a symbolic predicate constant, and $\mathbf{t}$ is a tuple of precomputed terms. Then the stable models of a program will be defined as stable models, in the sense of [13], of the set consisting of the translations of all rules of the program. Truszczynski's definition of stable models for infinitary propositional formulas is reviewed below.

Prior to defining the translation $\tau$ for rules, we will define it for ground literals, conditional literals, and aggregate expressions.

### 3.1 Review: Stable Models of Infinitary Formulas

Let $\sigma$ be a propositional signature, that is, a set of propositional atoms. The sets $\mathcal{F}_0^\sigma$, $\mathcal{F}_1^\sigma$, ... are defined as follows:

- $\mathcal{F}_0^\sigma = \sigma \cup \{\bot\}$,
- $\mathcal{F}_{i+1}^\sigma$ is obtained from $\mathcal{F}_i^\sigma$ by adding expressions $\mathcal{H}^\wedge$ and $\mathcal{H}^\vee$ for all subsets $\mathcal{H}$ of $\mathcal{F}_i^\sigma$, and expressions $F \to G$ for all $F, G \in \mathcal{F}_i^\sigma$.

The elements of $\bigcup_{i=0}^\infty \mathcal{F}_i^\sigma$ are called *(infinitary) formulas* over $\sigma$. Negation and equivalence are abbreviations.

Subsets of a signature $\sigma$ will be also called its *interpretations*. The satisfaction relation between an interpretation and a formula is defined in a natural way.

The *reduct* $F^I$ of a formula $F$ w.r.t. an interpretation $I$ is defined as follows:

- $\bot^I = \bot$.
- For $p \in \sigma$, $p^I = \bot$ if $I \not\models p$; otherwise $p^I = p$.
- $(\mathcal{H}^\wedge)^I = \{G^I \mid G \in \mathcal{H}\}^\wedge$.
- $(\mathcal{H}^\vee)^I = \{G^I \mid G \in \mathcal{H}\}^\vee$.
- $(G \to H)^I = \bot$ if $I \not\models G \to H$; otherwise $(G \to H)^I = G^I \to H^I$.

An interpretation $I$ is a *stable model* of a set $\mathcal{H}$ of formulas if it is minimal w.r.t. set inclusion among the interpretations satisfying the reducts of all formulas from $\mathcal{H}$.

### 3.2 Semantics of Well-Formed Ground Literals

A term $\mathbf{t}$ is *well-formed* if it contains neither symbolic object constants nor symbolic function constants in the scope of arithmetical functions. For instance, all arithmetical terms and all precomputed terms are well-formed; $c+2$ is not well-formed. The definition of "well-formed" for literals, aggregate expressions, and so forth is the same.

For every well-formed ground term $t$, by $[t]$ we denote the precomputed term obtained from $t$ by evaluating all arithmetical functions, and similarly for tuples of terms. For instance, $[f(2+2)]$ is $f(4)$.

The translation $\tau L$ of a well-formed ground literal $L$ is defined as follows:

- $\tau(p(\mathbf{t}))$ is $p([\mathbf{t}])$;
- $\tau(t_1 \prec t_2)$, where $\prec$ is the symbol $=$ or a comparison, is $\top$ if the relation $\prec$ holds between $[t_1]$ and $[t_2]$, and $\bot$ otherwise;
- $\tau(not\ A)$ is $\neg\tau A$.

For instance, $\tau(not\ p(f(2{+}2)))$ is $\neg p(f(4))$, and $\tau(2{+}2{=}\,4)$ is $\top$.

Furthermore, $\tau\bot$ stands for $\bot$, and, for any list $\mathbf{L}$ of ground literals, $\tau\mathbf{L}$ is the conjunction of the formulas $\tau L$ for all members $L$ of $\mathbf{L}$.

### 3.3 Global Variables

About a variable we say that it is *global*

- in a conditional literal $H : \mathbf{L}$, if it occurs in $H$ but does not occur in $\mathbf{L}$;
- in an aggregate expression $\alpha\{\mathbf{t} : \mathbf{L}\} \prec s$, if it occurs in the term $s$;
- in a rule (3), if it is global in at least one of the expressions $H_i$, $B_i$.

For instance, the head of the rule (2) is a literal with the global variable $N$, and its body is an aggregate expression with the global variable $N$. Consequently $N$ is global in the rule as well.

A conditional literal, an aggregate expression, or a rule is *closed* if it has no global variables. An *instance* of a rule $R$ is any well-formed closed rule that can be obtained from $R$ by substituting precomputed terms for global variables. For instance,

$$total\_hours(6) \leftarrow sum\{H, C : enroll(C), hours(H, C)\} = 6$$

is an instance of rule (2). It is clear that if a rule is not well-formed then it has no instances.

### 3.4 Semantics of Closed Conditional Literals

If $t$ is a term, $\mathbf{x}$ is a tuple of distinct variables, and $\mathbf{r}$ is a tuple of terms of the same length as $\mathbf{x}$, then the term obtained from $t$ by substituting $\mathbf{r}$ for $\mathbf{x}$ will be denoted by $t^{\mathbf{x}}_{\mathbf{r}}$. Similar notation will be used for the result of substituting $\mathbf{r}$ for $\mathbf{x}$ in expressions of other kinds, such as literals and lists of literals.

The result of applying $\tau$ to a closed conditional literal $H : \mathbf{L}$ is the conjunction of the formulas

$$\tau(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \rightarrow \tau(H^{\mathbf{x}}_{\mathbf{r}})$$

where $\mathbf{x}$ is the list of variables occurring in $H : \mathbf{L}$, over all tuples $\mathbf{r}$ of precomputed terms of the same length as $\mathbf{x}$ such that both $\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}$ and $H^{\mathbf{x}}_{\mathbf{r}}$ are well-formed. For instance,

$$\tau(available(X) : person(X))$$

is the conjunction of the formulas $person(r) \rightarrow available(r)$ over all precomputed terms $r$;

$$\tau(\bot : p(2 \times X))$$

is the conjunction of the formulas $\neg p(2 \times i)$ over all numerals $i$. When a conditional literal occurs in the head of a rule, we will translate it in a different way. By $\tau_h(H : \mathbf{L})$ we denote the disjunction of the formulas

$$\tau(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \wedge \tau(H^{\mathbf{x}}_{\mathbf{r}})$$

where $\mathbf{x}$ and $\mathbf{r}$ are as above. For instance,

$$\tau_h(\mathit{available}(X) : \mathit{person}(X))$$

is the disjunction of the formulas $\mathit{person}(r) \wedge \mathit{available}(r)$ over all precomputed terms $r$.

### 3.5 Semantics of Closed Aggregate Expressions

In this section, the semantics of ground aggregates proposed in [1, Section 4.1] is adapted to closed aggregate expressions. Let $E$ be a closed aggregate expression $\alpha\{\mathbf{t} : \mathbf{L}\} \prec s$, and let $\mathbf{x}$ be the list of variables occurring in $E$. A tuple $\mathbf{r}$ of precomputed terms of the same length as $\mathbf{x}$ is *admissible* (w.r.t. $E$) if both $\mathbf{t}^{\mathbf{x}}_{\mathbf{r}}$ and $\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}$ are well-formed. About a set $\Delta$ of admissible tuples we say that it *justifies* $E$ if the relation $\prec$ holds between $\widehat{\alpha}(\{[\mathbf{t}^{\mathbf{x}}_{\mathbf{r}}] : \mathbf{r} \in \Delta\})$ and $[s]$. For instance, consider the aggregate expression

$$\mathit{sum}\{H, C : \mathit{enroll}(C), \mathit{hours}(H, C)\} = 6. \tag{5}$$

In this case, admissible tuples are arbitrary pairs of precomputed terms. The set $\{(3, \mathit{cs101}), (3, \mathit{cs102})\}$ justifies (5), because

$$\widehat{\mathit{sum}}(\{(H, C)^{H,C}_{3,\mathit{cs101}}, (H, C)^{H,C}_{3,\mathit{cs102}}\}) = \widehat{\mathit{sum}}(\{(3, \mathit{cs101}), (3, \mathit{cs102})\}) = 3+3 = 6.$$

More generally, a set $\Delta$ of pairs of precomputed terms justifies (5) whenever $\Delta$ contains finitely many pairs $(h, c)$ in which $h$ is a positive integer, and the sum of the integers $h$ over all these pairs is 6.

We define $\tau E$ as the conjunction of the implications

$$\bigwedge_{\mathbf{r} \in \Delta} \tau(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \rightarrow \bigvee_{\mathbf{r} \in A \setminus \Delta} \tau(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \tag{6}$$

over all sets $\Delta$ of admissible tuples that do not justify $E$, where $A$ is the set of all admissible tuples. For instance, if $E$ is (5) then the conjunctive terms of $\tau E$ are the formulas

$$\bigwedge_{(h,c) \in \Delta} (\mathit{enroll}(c) \wedge \mathit{hours}(h, c)) \rightarrow \bigvee_{(h,c) \notin \Delta} (\mathit{enroll}(c) \wedge \mathit{hours}(h, c)).$$

The conjunctive term corresponding to $\{(3, \mathit{cs101})\}$ as $\Delta$ says: if I am enrolled in CS101 for 3 hours then I am enrolled in at least one other course.

### 3.6 Semantics of Rules and Programs

For any rule $R$, $\tau R$ stands for the conjunction of the formulas

$$\tau B_1 \wedge \cdots \wedge \tau B_n \to \tau_h H_1 \vee \cdots \vee \tau_h H_m$$

for all instances (3) of $R$. A *stable model* of a program $\Pi$ is a stable model, in the sense of [13], of the set consisting of the formulas $\tau R$ for all rules $R$ of $\Pi$.

Consider, for instance, the rules of program (4). If $R$ is the rule $\{p(i)\}$ then $\tau R$ is

$$p(i) \vee \neg p(i) \tag{7}$$

$(i = 1, \ldots, n)$. If $R$ is the rule

$$\leftarrow p(X), p(Y), p(X+Y)$$

then the instances of $R$ are rules of the form

$$\leftarrow p(i), p(j), p(i+j)$$

for all numerals $i$, $j$. (Substituting precomputed ground terms other than numerals would produce a rule that is not well-formed.) Consequently $\tau R$ is in this case the infinite conjunction

$$\bigwedge_{\substack{i,j,k \in \mathbf{Z} \\ i+j=k}} \neg(p(i) \wedge p(j) \wedge p(k)). \tag{8}$$

The stable models of program (4) are the stable models of formulas (7), (8), that is, sets of the form $\{p(i) : i \in S\}$ for all sum-free subsets $S$ of $\{1, \ldots, n\}$.

## 4 Reasoning about Gringo Programs

In this section we give examples of reasoning about Gringo programs on the basis of the semantics defined above. These examples use the results of [8], and we assume here that the reader is familiar with that paper.

### 4.1 Simplifying a Rule from Example 3.7 of User's Guide

Consider the rule[7]

$$\mathit{weekdays} \leftarrow \mathit{day}(X) : (\mathit{day}(X), \mathit{not}\ \mathit{weekend}(X)). \tag{9}$$

Replacing this rule with the fact *weekdays* within any program will not affect the set of stable models. Indeed, the result of applying translation $\tau$ to (9) is the formula

$$\bigwedge_r (\mathit{day}(r) \wedge \neg \mathit{weekend}(r) \to \mathit{day}(r)) \ \to\ \mathit{weekdays}, \tag{10}$$

[7] This rule is similar to a rule from Example 3.7 of the Gringo 3 *User's Guide* (see Footnote 2).

where the conjunction extends over all precomputed terms $r$. The formula

$$day(r) \wedge \neg weekend(r) \rightarrow day(r)$$

is intuitionistically provable. By the replacement property of the basic system of natural deduction from [8], it follows that (10) is equivalent to *weekdays* in the basic system. By the main theorem of [8], it follows that replacing (10) with the atom *weekdays* within any set of formulas does not affect the set of stable models.

### 4.2 Simplifying the Sorting Rule

The rule

$$order(X,Y) \leftarrow p(X),\, p(Y),\, X < Y,\, not\ p(Z) : (p(Z), X < Z, Z < Y) \tag{11}$$

can be used for sorting.[8] It can be replaced by either of the following two shorter rules within any program without changing that program's stable models.

$$order(X,Y) \leftarrow p(X),\, p(Y),\, X < Y,\, \bot : (p(Z), X < Z, Z < Y) \tag{12}$$

$$order(X,Y) \leftarrow p(X),\, p(Y),\, X < Y,\, not\ p(Z) : (X < Z, Z < Y) \tag{13}$$

Let's prove this claim for rule (12). By the main theorem of [8] it is sufficient to show that the result of applying $\tau$ to (11) is equivalent in the basic system to the result of applying $\tau$ to (12). The instances of (11) are the rules

$$order(i,j) \leftarrow p(i),\, p(j),\, i < j,\, not\ p(Z) : (p(Z), i < Z, Z < j),$$

and the instances of (12) are the rules

$$order(i,j) \leftarrow p(i),\, p(j),\, i < j,\, \bot : (p(Z), i < Z, Z < j)$$

where $i$ and $j$ are arbitrary numerals. The result of applying $\tau$ to (11) is the conjunction of the formulas

$$p(i) \wedge p(j) \wedge i < j \wedge \bigwedge_k (\neg p(k) \wedge i < k \wedge k < j \rightarrow p(k)) \rightarrow order(i,j) \tag{14}$$

for all numerals $i,\ j$. The result of applying $\tau$ to (12) is the conjunction of the formulas

$$p(i) \wedge p(j) \wedge i < j \wedge \bigwedge_k (\neg p(k) \wedge i < k \wedge k < j \rightarrow \bot) \rightarrow order(i,j). \tag{15}$$

By the replacement property of the basic system, it is sufficient to observe that

$$p(k) \wedge i < k \wedge k < j \rightarrow \neg p(k)$$

is intuitionistically equivalent to

$$p(k) \wedge i < k \wedge k < j \rightarrow \bot.$$

The proof for rule (13) is similar. Rule (12), like rule (11), is safe; rule (13) is not.

[8] This rule was communicated to us by Roland Kaminski on October 21, 2012.

### 4.3 Eliminating Choice in Favor of a Conditional Literal

Replacing the rule

$$\{p(X)\} \leftarrow q(X) \tag{16}$$

with

$$p(X) \leftarrow q(X), \bot : not\ p(X) \tag{17}$$

within any program will not affect the set of stable models. Indeed, the result of applying translation $\tau$ to (16) is

$$\bigwedge_r (q(r) \rightarrow p(r) \vee \neg p(r)) \tag{18}$$

where the conjunction extends over all precomputed terms $r$, and the result of applying $\tau$ to (17) is

$$\bigwedge_r (q(r) \wedge \neg\neg p(r) \rightarrow p(r)). \tag{19}$$

The implication from (18) is equivalent to the implication from (19) in the extension of intuitionistic logic obtained by adding the axiom schema

$$\neg F \vee \neg\neg F,$$

and consequently in the extended system presented in [8, Section 7]. By the replacement property of the extended system, it follows that (18) is equivalent to (19) in the extended system as well.

### 4.4 Eliminating a Trivial Aggregate Expression

The rule (1) says, informally speaking, that we can conclude $p(Y)$ once we established that there exists at least one $X$ such that $q(X, Y)$. Replacing this rule with

$$p(Y) \leftarrow q(X, Y) \tag{20}$$

within any program will not affect the set of stable models.

To prove this claim, we need to calculate the result of applying $\tau$ to rule (1). The instances of (1) are the rules

$$p(t) \leftarrow count\{X, t : q(X, t)\} \geq 1 \tag{21}$$

for all precomputed terms $t$. Consider the aggregate expression $E$ in the body of (21). Any precomputed term $r$ is admissible w.r.t. $E$. A set $\Delta$ of precomputed terms justifies $E$ if

$$\widehat{count}(\{(r, t) : r \in \Delta\}) \geq 1,$$

that is to say, if $\Delta$ is non-empty. Consequently $\tau E$ consists of only one implication (6), with the empty $\Delta$. The antecedent of this implication is the empty

conjunction $\top$, and its consequent is the disjunction $\bigvee_u q(u,t)$ over all precomputed terms $u$. Then the result of applying $\tau$ to (1) is

$$\bigwedge_t \left( \bigvee_u q(u,t) \ \rightarrow \ p(t) \right). \tag{22}$$

On the other hand, the result of applying $\tau$ to (20) is

$$\bigwedge_{t,u} (q(u,t) \rightarrow p(t)).$$

This formula is equivalent to (22) in the basic system [8, Example 2].

### 4.5 Replacing an Aggregate Expression with a Conditional Literal

Informally speaking, the rule

$$q \leftarrow count\{X : p(X)\} = 0 \tag{23}$$

says that we can conclude $q$ once we have established that the cardinality of the set $\{X : p(X)\}$ is 0; the rule

$$q \leftarrow \bot : p(X) \tag{24}$$

says that we can conclude $q$ once we have established that $p(X)$ does not hold for any $X$. We'll prove that replacing (23) with (24) within any program will not affect the set of stable models. To this end, we'll show that the results of applying $\tau$ to (23) and (24) are equivalent to each other in the extended system from [8, Section 7].

First, we'll need to calculate the result of applying $\tau$ to rule (23). Consider the aggregate expression $E$ in the body of (23). Any precomputed term $r$ is admissible w.r.t. $E$. A set $\Delta$ of precomputed terms justifies $E$ if

$$\widehat{count}(\{r : r \in \Delta\}) = 0,$$

that is to say, if $\Delta$ is empty. Consequently $\tau E$ is the conjunction of the implications

$$\bigwedge_{r \in \Delta} p(r) \rightarrow \bigvee_{r \in A \setminus \Delta} p(r) \tag{25}$$

for all non-empty subsets $\Delta$ of the set $A$ of precomputed terms. The result of applying $\tau$ to (23) is

$$\left( \bigwedge_{\substack{\Delta \subseteq A \\ \Delta \neq \emptyset}} \left( \bigwedge_{r \in \Delta} p(r) \rightarrow \bigvee_{r \in A \setminus \Delta} p(r) \right) \right) \rightarrow q. \tag{26}$$

The result of applying $\tau$ to (24), on the other hand, is

$$\left( \bigwedge_{r \in A} \neg p(r) \right) \rightarrow q. \tag{27}$$

The fact that the antecedents of (26) and (27) are equivalent to each other in the extended system can be established by essentially the same argument as in [8, Example 7]. By the replacement property of the extended system, it follows that (26) is equivalent to (27) in the extended system as well.

### 4.6 Eliminating Summation over the Empty Set

Informally speaking, the rule

$$q \leftarrow sum\{X : p(X)\} = 0 \tag{28}$$

says that we can conclude $q$ once we have established that the sum of the elements of the set $\{X : p(X)\}$ is 0. In the presence of the constraint

$$\leftarrow p(X), \tag{29}$$

replacing (28) with the fact $q$ will not affect the stable models.

To see this, first we calculate the result of applying $\tau$ to rule (28). Consider the aggregate expression $E$ in the body of (28). Any precomputed term $r$ is admissible w.r.t. $E$. A set $\Delta$ of precomputed terms justifies $E$ if

$$\widehat{sum}(\{r : r \in \Delta\}) = 0,$$

that is to say, if $\Delta$ contains no positive integers. Consequently $\tau E$ is the conjunction of the implications

$$\bigwedge_{r \in \Delta} p(r) \rightarrow \bigvee_{r \in A \setminus \Delta} p(r) \tag{30}$$

for subsets $\Delta$ of the set $A$ of precomputed terms that contain at least one positive integer. The result of applying $\tau$ to (28) is

$$\left( \bigwedge_{\substack{\Delta \subseteq A \\ \Delta \cap \mathbf{Z} \neq \emptyset}} \left( \bigwedge_{r \in \Delta} p(r) \rightarrow \bigvee_{r \in A \setminus \Delta} p(r) \right) \right) \rightarrow q. \tag{31}$$

The result of applying $\tau$ to (29), on the other hand, is

$$\bigwedge_{r \in A} \neg p(r). \tag{32}$$

For every nonempty $\Delta$, the antecedent of (30) contradicts (32). Consequently, the antecedent of (31) can be derived from (32) in the basic system. It follows that the equivalence between (31) and the atom $q$ can be derived in the basic system under assumption (32).

## 5 Conclusion

In this note we approached the problem of defining the semantics of Gringo by reducing Gringo programs to infinitary propositional formulas. We argued that this approach to semantics may allow us to study equivalent transformations of programs using natural deduction in infinitary propositional logic.

In the absence of a precise semantics, it is impossible to put the study of some important issues on a firm foundation. This includes the correctness of ASP programs, grounders, solvers, and optimization methods, and also the relationship between input languages of different solvers (for instance, the equivalence of the semantics of aggregate expressions in Gringo to their semantics in the ASP Core language and in the language proposed in [5] under the assumption that aggregates are used nonrecursively). As future work, we are interested in addressing some of these tasks on the basis of the semantics proposed in this note. Proving the correctness of the intelligent instantiation algorithms implemented in GRINGO will provide justification for our informal claim that for a safe program, the semantics proposed here correctly describes the output produced by GRINGO.

## Acknowledgements

Many thanks to Roland Kaminski and Torsten Schaub for helping us understand the input language of GRINGO. Roland, Michael Gelfond, Yuliya Lierler, Joohyung Lee, and anonymous referees provided valuable comments on drafts of this note.

## References

1. Ferraris, P.: Answer sets for propositional theories. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 119–131 (2005)
2. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence 175, 236–263 (2011)
3. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming 5, 45–74 (2005)
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)
5. Gelfond, M.: Representing knowledge in A-Prolog. Lecture Notes in Computer Science 2408, 413–451 (2002)
6. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium. pp. 1070–1080. MIT Press (1988)
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)

8. Harrison, A., Lifschitz, V., Truszczynski, M.: On equivalent transformations of infinitary formulas under the stable model semantics (preliminary report)[9]. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR) (2013), to appear
9. Lee, J., Meng, Y.: Stable models of formulas with generalized quantifiers. In: Working Notes of the 14th International Workshop on Non-Monotonic Reasoning (NMR) (2012)
10. Lee, J., Meng, Y.: Stable models of formulas with generalized quantifiers (preliminary report). In: Technical Communications of the 28th International Conference on Logic Programming (ICLP). pp. 61–71 (2012)
11. Lee, J., Meng, Y.: Two new definitions of stable models of logic programs with generalized quantifiers. In: Working Notes of the 5th Workshop of Answer Set Programming and Other Computing Paradigms (ASPOCP) (2012)
12. Pearce, D., Valverde, A.: Towards a first order equilibrium logic for nonmonotonic reasoning. In: Proceedings of European Conference on Logics in Artificial Intelligence (JELIA). pp. 147–160 (2004)
13. Truszczynski, M.: Connecting first-order ASP and the logic FO(ID) through reducts. In: Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz. Springer (2012)

[9] `http://www.cs.utexas.edu/users/vl/papers/etinf.pdf`

# Abstract Modular Systems and Solvers

Yuliya Lierler[1] and Miroslaw Truszczynski[2]

[1] University of Nebraska at Omaha
ylierler@unomaha.edu
[2] University of Kentucky
mirek@cs.uky.edu

**Abstract.** Integrating diverse formalisms into modular knowledge representation systems offers increased expressivity, modeling convenience and computational benefits. We introduce concepts of *abstract modules* and *abstract modular systems* to study general principles behind the design and analysis of model-finding programs, or *solvers*, for integrated heterogeneous multi-logic systems. We show how abstract modules and abstract modular systems give rise to *transition systems*, which are a natural and convenient representation of solvers pioneered by the SAT community. We illustrate our approach by showing how it applies to answer set programming and propositional logic, and to multi-logic systems based on these two formalisms.

## 1 Introduction

Knowledge representation and reasoning (KR) is concerned with developing formal languages and logics to model knowledge, and with designing and implementing corresponding automated reasoning tools. The choice of specific logics and tools depends on the type of knowledge to be represented and reasoned about. Different logics are suitable for common-sense reasoning, reasoning under incomplete information and uncertainty, for temporal and spatial reasoning, and for modeling and solving boolean constraints, or constraints over larger, even continuous domains. In applications in areas such as distributed databases, semantic web, hybrid constraint modeling and solving, to name just a few, several of these aspects come to play. Accordingly, often diverse logics have to be accommodated together. Similar issues arise in research on multi-context systems where the major task is to model *contextual* information and the flow of information among *contexts* [17, 7]. The contexts are commonly modeled by theories in some logics.

Modeling convenience is not the only reason why diverse logics are combined into modular hybrid KR systems. Another major motivation is to exploit in reasoning the transparent structure that comes from modularity, computational strengths of individual logics, and synergies that may arise when they are put together. Constraint logic programming [8] and satisfiability modulo theories (SMT) [20, 2] are well-known examples of formalisms stemming directly from such considerations. More recent examples include constraint answer set programming (CASP) [13], which integrates answer set programming (ASP) [6, 15, 18]) with constraint modeling languages [22], and "multi-logic" formalisms PC(ID) [16], SM(ASP) [14] and ASP-FO [4] that combine modules

expressed as logic theories under the classical semantics with modules given as answer-set programs.

The key computational task arising in KR is that of model generation. Model-generating programs or *solvers*, developed in satisfiability (SAT) and ASP proved to be effective in a broad range of KR applications. Accordingly, model generation is of critical importance in modular multi-logic systems. Research on formalisms listed above resulted in fast solvers that demonstrate gains one can obtain from their heterogeneous nature. However, the diversity of logics considered and low-level technical details of their syntax and semantics obscure general principles that are important in the design and analysis of solvers for multi-logic systems.

In this paper we address this problem by proposing a language for talking about modular multi-logic systems that (i) abstracts away the syntactic details, (ii) is expressive enough to capture various concepts of inference, and (iii) is based only on the weakest assumptions concerning the semantics. The basic elements of this language are *abstract modules*. Collections of abstract modules constitute *abstract modular systems*. We define the semantics of abstract modules and show that they provide a uniform language capable of capturing different logics, diverse inference mechanisms, and their modular combinations. Importantly, abstract modules and abstract modular systems give rise to *transition systems* of the type introduced by Nieuwenhuis, Oliveras, and Tinelli [20] in their study of SAT and SMT solvers. We show that as in that earlier work, our transition systems provide a natural and convenient representation of solvers for abstract modules and abstract modular systems. We demonstrate that they lend themselves well to extensions that capture such important solver design techniques as learning (which here comes in two flavors: *local* that is limited to single modules, and *global* that is applied across modules). Throughout the paper, we illustrate our approach by showing how it applies to propositional logic and answer set programming, and to multi-logic systems based on these two formalisms.

The results of our paper show that abstract modular systems and the corresponding abstract framework for describing and analyzing algorithms for modular declarative programming tools relying on multi-logics are useful and effective conceptualizations that can contribute to (i) clarifying computational principles of such systems and to (ii) the development of new ones.

The paper is organized as follows. We start by introducing one of the main concepts in the paper – abstract modules. We then proceed to formulating an algorithm (a family of algorithms) for finding models of such modules. We use an abstract transition system stemming from the framework by Nieuwenhuis et al. [20] for this purpose. Section 4 presents the definition of an abstract modular system and a corresponding solver based on backtrack search. We then discuss how this solver maybe augmented by such advanced SAT solving technique as learning. Section 6 provides an account on related work.

## 2 Abstract Modules

Let $\sigma$ be a fixed finite vocabulary (a set of propositional atoms). A *module* over the vocabulary $\sigma$ is a directed graph $S$ whose nodes are $\bot$ and all consistent sets of literals,

and each edge is of the form $(M, \bot)$ or $(M, Ml)$, where $l \notin M$ and $Ml$ is a shorthand for $M \cup \{l\}$. If $S$ is a module, we write $\sigma(S)$ for its vocabulary. For a set $X$ of literals, we denote $X^+ = \{a\colon a \in X\}$ and $X^- = \{a\colon \neg a \in X\}$.

Intuitively, an edge $(M, Ml)$ in a module indicates that the module supports inferring $l$ whenever all literals in $M$ are given. An edge $(M, \bot)$, $M \neq \emptyset$, indicates that there is a literal $l \in M$ such that a derivation of its dual $\bar{l}$ (and hence, a derivation of a contradiction) is supported by the module, assuming the literals in $M$ are given. Finally, the edge $(\emptyset, \bot)$ indicates that the module is "explicitly" contradictory.

A node in a module is *terminal* if no edge leaves it. A terminal node that is consistent and complete is a *model node* of the module. A set $X$ of atoms is a *model* of a module $S$ if for some model node $Y$ in $S$, $X \cap \sigma(S) = Y^+$. Thus, models of modules are not restricted to the signature of the module. Clearly, for every model node $Y$ in $S$, $Y^+$ is a model of $S$.

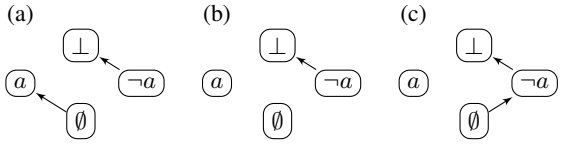



Fig. 1: Three modules over the vocabulary $\{a\}$.

A module $S$ *entails* a formula $\varphi$, written $S \models \varphi$, if for every model $I$ of $S$ we have $I \models \varphi$. It is immaterial what logic the formula $\varphi$ comes from as long as (i) the vocabulary of the logic is a subset of the vocabulary of $S$, and (ii) the semantics of the logic is given by a satisfiability relation $I \models \varphi$. A module $S$ *entails* a formula $\varphi$ wrt a set $M$ of literals (over the same vocabulary as $S$), written $S \models_M \varphi$, if for every model $I$ of $S$ such that $M^+ \subseteq I$ and $M^- \cap I = \emptyset$, $I \models \varphi$.

Clearly, if two modules over the same signature have the same model nodes, they have the same models. Semantically the three modules in Figure 1 are the same. They have the same models (each has $\{a\}$ as its only model in the signature of the module) and so they entail the same formulas. We call modules with the same models *equivalent*.

Modules represent more than just the set of their models. As already suggested above, the intended role of edges in a module is to represent allowed "local" inferences. For instance, given the empty set of literals, the first module in Figure 1 supports inferring $a$ and the third module $\neg a$. In the latter case, the inference is not "sound" as it contradicts the semantic information in the module as that module does not entail $\neg a$ with respect to the empty set of literals.

Formally, an edge from a node $M$ to a node $M'$ in a module $S$ is *sound* if $S \models_M M'$.[3] Clearly, if $M'$ has the form $Ml$ then $S \models_M M'$ if and only if $S \models_M l$. Similarly, if $M' = \bot$ then $S \models_M M'$ if and only if no model of $S$ is consistent with $M$ (that is, contains $M^+$ and is disjoint with $M^-$). A module is *sound* if all of its edges are sound,

[3] In the paper, we sometimes identify a set of literals with the conjunction of its elements. Here $M'$ is to be understood as the conjunction of its elements.

that is, if all inferences supported by the module are sound with respect to the semantics of the module given by its set of models. The modules in Figures 1(a) and (b) are sound, the one in Figure 1(c) is not. Namely, the inference of $\neg a$ from $\emptyset$ is not sound.

Given two modules $S$ and $S'$ over the same vocabulary, we say that $S$ is *equivalently contained* in $S'$, $S \sqsubseteq S'$, if $S$ and $S'$ are equivalent (have the same model nodes) and the set of edges of $S$ is a subset of the set of edges of $S'$. Maximal (wrt $\sqsubseteq$) sound modules are called *saturated*. We say that an edge from a node $M$ to $\bot$ in a module $S$ is *critical* if $M$ is a complete and consistent set of literals over $\sigma(S)$. The following properties are evident.

**Proposition 1.** *Every two modules over the same signature and with the same critical edges are equivalent. For a saturated module $S$, every sound module with the same critical edges as $S$ is equivalently contained in $S$. A module $S$ is saturated if and only if it is sound and for every set $M$ of literals and for every literal $l \notin M$, $(M, Ml)$ is an edge of $S$ whenever $S \models_M l$.*

Clearly, only the module in Figure 1(a) is saturated. The other two are not. The one in (b) is not maximal with respect to the containment relation, the one in (c) is not sound. We also note that all three modules have the same critical edges. Thus, by Proposition 1, they are equivalent, a property we already observed earlier. Finally, the module in Figure 1(b) is equivalently contained in the module in Figure 1(a).

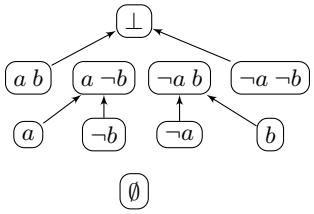



Fig. 2: An abstract module over the vocabulary $\{a, b\}$ related to the theory (1).

In practice, modules (graphs) are specified by means of theories and logics (more precisely, specific forms of inference in logics). For instance, a propositional theory $T$ over a vocabulary $\sigma$ and the inference method given by the classical concept of entailment determine a module over $\sigma$ in which (i) $(M, Ml)$ is an edge if and only if $T \cup M \models l$; and (ii) $(M, \bot)$ is an edge if and only if no model of $T$ is consistent with $M$. Figure 1(a) shows the module determined in this way by the theory consisting of the clause $a$. Similarly, Figure 2 presents such a module for the theory

$$a \vee b, \quad \neg a \vee \neg b. \tag{1}$$

This module is saturated. Also, theory (1) and the inference method given by the unit propagate rule, a classical propagator used in SAT solvers, determines this module. In other words, for the theory (1) the unit propagation rule captures entailment.

We say that a module $S$ is *equivalent* to a theory $T$ in some logic if the models of $S$ coincide with the models of $T$. Clearly, the module in Figure 2 is equivalent to the propositional theory (1).

Modules are not meant for modeling. Representations by means of logic theories are usually more concise (the size of a module is exponential in the size of its vocabulary). Furthermore, the logic languages align closely with natural language, which facilitates modeling and makes the correspondence between logic theories and knowledge they represent direct. Modules lack this connection to natural language.

The power of modules comes from the fact that they provide a uniform, syntax-independent way to describe theories and inference methods stemming from *different* logics. For instance, they represent equally well both propositional theories and logic programs under the answer-set semantics. Indeed, let us consider the logic program

$$\begin{array}{l} \{a\}, \\ b \leftarrow not\ a, \end{array} \tag{2}$$

where $\{a\}$ represents the so-called *choice rule* [23]. This program has two answer sets $\{a\}$ and $\{b\}$. Since these are also the only two models of the propositional theory (1), it is clear that the module in Figure 2 represents the program (2) and the reasoning mechanism of entailment with respect to its answer sets. Two other modules associated with program (2) are given in Figure 3. The module in Figure 3(a) represents program (2) and the reasoning on programs based on *forward chaining*; we call this module $M_{fc}$. We recall that given a set of literals, forward chaining supports the derivation of the head of a rule whose body is satisfied. We note that the module $M_{fc}$ is not equivalent to program (2). Indeed, $\{a, b\}$ is a model of $M_{fc}$ whereas it is not an answer set of (2). This is due to the fact that the critical edge from $a\,b$ to $\bot$ is unsupported by forward chaining and is not present in $M_{fc}$. On the other hand, all edges due to forward chaining are sound both in the module in Figure 2, which we call $M_e$, and $M_{fc}$. In the next section we discuss a combination of inference rules that yields a reasoning mechanism subsuming forward chaining and resulting in a module, shown in Figure 3(b), that is equivalently contained in $M_e$ and so, equivalent to the program (2). This discussion indicates that the language of modules is flexible enough to represent not only the semantic mechanism of entailment, but also syntactically defined "proof systems" — reasoning mechanisms based on specific inference rules.

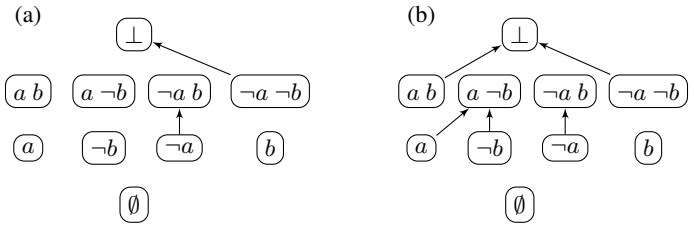



Fig. 3: Two abstract modules over the vocabulary $\{a, b\}$ related to the logic program (2).

## 3 Abstract Modular Solver: $\text{AM}_S$

Finding models of logic theories and programs is a key computational task in declarative programming. Nieuwenhuis et al. [20] proposed to use transition systems to describe search procedures involved in model-finding programs commonly called *solvers*, and developed that approach for the case of SAT. Their transition system framework can express DPLL, the basic search procedure employed by SAT solvers, and its enhancements such as conflict driven clause learning. Lierler [12] proposed a similar framework for specifying an answer set solver SMODELS. Lierler and Truszczynski [14] extended that framework to capture such modern ASP solvers as CMODELS and CLASP, as well as a PC(ID) solver MINISAT(ID).

An abstract nature (independence from language and reasoning method selection) of modules introduced in this work and their relation to proof systems makes them a convenient, broadly applicable tool to study and analyze solvers. In this section, we adapt the transition system framework of Nieuwenhuis et al. [20] to the case of abstract modules. We then illustrate how it can be used to define solvers for instantiations of abstract modules such as propositional theories under the classical semantics and logic programs under the answer-set semantics.

A *state* relative to $\sigma$ is either a special state $\bot$ (fail state) or an *ordered* consistent set $M$ of literals over $\sigma$, some possibly annotated by $\Delta$, which marks them as *decision* literals. For instance, the states relative to a singleton set $\{a\}$ of atoms are $\emptyset,\ a,\ \neg a,\ a^{\Delta},\ \neg a^{\Delta},\ \bot$.

Frequently, we consider a state $M$ as a set of literals, ignoring both the annotations and the order between its elements. If neither a literal $l$ nor its complement occur in $M$, then $l$ is *unassigned* by $M$.

Each module $S$ determines its *transition graph* $\text{AM}_S$: The set of nodes of $\text{AM}_S$ consists of the states relative to the vocabulary of $S$. The edges of the graph $\text{AM}_S$ are specified by the *transition rules* listed in Figure 4. The first three rules depend on the module, the fourth rule, *Decide*, does not. It has the same form no matter what module we consider. Hence, we omit the reference to the module from its notation.

$$\textit{Propagate}_S: \quad M \longrightarrow M\ l \ \text{ if } \ S \text{ has an edge from } M \text{ to } M\ l$$

$$\textit{Fail}_S: \quad M \longrightarrow \bot \ \text{ if } \begin{cases} S \text{ has an edge from } M \text{ to } \bot, \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_S: \quad P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l} \ \text{ if } \begin{cases} S \text{ has an edge from } P\ l\ Q \text{ to } \bot, \\ Q \text{ contains no decision literals} \end{cases}$$

$$\textit{Decide}: \quad M \longrightarrow M\ l^{\Delta} \ \text{ if } \ l \text{ is unassigned by } M$$

Fig. 4: The transition rules of the graph $\text{AM}_S$.

The graph $\text{AM}_S$ can be used to decide whether a module $S$ has a model. The following properties are essential.

**Theorem 1.** *For every sound module $S$,*

*(a) graph $\mathrm{AM}_S$ is finite and acyclic,*
*(b) for any terminal state $M$ of $\mathrm{AM}_S$ other than $\bot$, $M^+$ is a model of $S$,*
*(c) state $\bot$ is reachable from $\emptyset$ in $\mathrm{AM}_S$ if and only if $S$ is unsatisfiable (has no models).*

Thus, to decide whether a sound module $S$ has a model it is enough to find in the graph $\mathrm{AM}_S$ a path leading from node $\emptyset$ to a terminal node $M$. If $M = \bot$, $S$ is unsatisfiable. Otherwise, $M$ is a model of $S$.

For instance, let $S$ be a module in Figure 2. Below we show a path in the transition graph $\mathrm{AM}_S$ with every edge annotated by the corresponding transition rule:

$$\emptyset \quad \overset{\mathit{Decide}}{\longrightarrow} \quad b^{\Delta} \quad \overset{\mathit{Propagate}_S}{\longrightarrow} \quad b^{\Delta}\, \neg a. \tag{3}$$

The state $b^{\Delta}\, \neg a$ is terminal. Thus, Theorem 1 (b) asserts that $\{b, \neg a\}$ is a model of $S$. There may be several paths determining the same model. For instance, the path

$$\emptyset \quad \overset{\mathit{Decide}}{\longrightarrow} \quad \neg a^{\Delta} \quad \overset{\mathit{Decide}}{\longrightarrow} \quad \neg a^{\Delta}\, b^{\Delta}. \tag{4}$$

leads to the terminal node $\neg a^{\Delta}\, b^{\Delta}$, which is different from $b^{\Delta}\, \neg a$ but corresponds to the same model. We can view a path in the graph $\mathrm{AM}_S$ as a description of a process of search for a model of module $S$ by applying transition rules. Therefore, we can characterize a solver based on the transition system $\mathrm{AM}_S$ by describing a strategy for choosing a path in $\mathrm{AM}_S$. Such a strategy can be based, in particular, on assigning priorities to some or all transition rules of $\mathrm{AM}_S$, so that a solver will never apply a transition rule in a state if a rule with higher priority is applicable to the same state. For example, priorities

$$\mathit{Backtrack}_S, \mathit{Fail}_S >> \mathit{Propagate}_S >> \mathit{Decide}$$

on the transition rules of $\mathrm{AM}_S$ specify a solver that follows available inferences (modeled by edges in the module $S$) before executing a transition due to *Decide*. The path (3) in the transition graph of the module from Figure 2 follows that strategy, whereas the path (4) does not.

We now review the graph $\mathrm{DP}_F$ introduced for the classical DPLL algorithm by Nieuvenhuis et al. [20], adjusting the presentation to the form convenient for our purposes. We then demonstrate its relation to the $\mathrm{AM}_S$ graph. The set of nodes of $\mathrm{DP}_F$ consists of the states relative to the vocabulary of a CNF formula (a set of clauses) $F$. The edges of the graph $\mathrm{DP}_F$ are specified by the transition rule *Decide* of the graph $\mathrm{AM}_S$ and the rules presented in Figure 5. For example, let $F_1$ be the theory consisting of a single clause $a$. Figure 6 presents $\mathrm{DP}_{F_1}$.

For a CNF formula $F$, by $\mu(\mathrm{DP}_F)$ we denote the graph (abstract module) constructed from $\mathrm{DP}_F$ by dropping all nodes that contain decision literals. We note that for the graph $\mathrm{DP}_{F_1}$ in Figure 6, the module $\mu(\mathrm{DP}_{F_1})$ coincides with the module in Figure 1(a). This is a manifestation of a general property.

**Proposition 2.** *For every CNF formula $F$, the graph $\mu(\mathrm{DP}_F)$ is a sound abstract module equivalent to $F$. Furthermore, the graphs $\mathrm{AM}_{\mu(\mathrm{DP}_F)}$ and $\mathrm{DP}_F$ are identical.*

$$\textit{UnitPropagate}_F: \quad M \longrightarrow M\, l \text{ if } \begin{cases} C \vee l \in F \text{ and } M \models \neg C, \\ l \text{ is unassigned by } M \end{cases}$$

$$\textit{Fail}_F: \quad M \longrightarrow \bot \text{ if } \begin{cases} C \in F \text{ and } M \models \neg C, \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_F: \quad P\, l^{\Delta}\, Q \longrightarrow P\, \bar{l} \text{ if } \begin{cases} C \in F \text{ and } P\, l^{\Delta}\, Q \models \neg C, \\ Q \text{ contains no decision literals} \end{cases}$$

Fig. 5: Three transition rules of the graph $\text{DP}_F$.

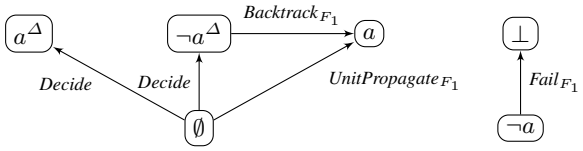



Fig. 6: The $\text{DP}_{F_1}$ graph where $F_1$ is a single clause $a$.

Theorem 1 and the fact that the module $\mu(\text{DP}_F)$ is equivalent to a CNF formula $F$ (Proposition 2) imply that the graph $\text{DP}_F$ can be used for deciding the satisfiability of $F$. It is enough to find a path leading from node $\emptyset$ to a terminal node $M$: if $M = \bot$ then $F$ is unsatisfiable; otherwise, $M$ is a model of $F$. For instance, the only terminal states reachable from the state $\emptyset$ in $\text{DP}_{F_1}$ are $a$ and $a^{\Delta}$. This translates into the fact that $a$ is a model of $F_1$. This is exactly the result that Nieuwenhuis et al. [20] stated for the graph $\text{DP}_F$:

**Corollary 1.** *For any CNF formula $F$,*

*(a) graph $\text{DP}_F$ is finite and acyclic,*
*(b) for any terminal state $M$ of $\text{DP}_F$ other than $\bot$, $M$ is a model of $F$,*
*(c) state $\bot$ is reachable from $\emptyset$ in $\text{DP}_F$ if and only if $F$ is unsatisfiable (has no models).*

We now introduce the graph $\text{AS}_{\Pi}$ that extends the DPLL graph by Nieuwenhuis et al. so that the result can be used to specify an algorithm for finding answer sets of a program. The graph $\text{AS}_{\Pi}$ can be used to form a sound module equivalent to a program $\Pi$ in the same way as we used $\text{DP}_F$ to form a sound module equivalent to a CNF formula $F$.

We assume the reader to be familiar with the concept of *unfounded sets* [26, 10]. For a set $M$ of literals and a program $\Pi$, by $U(M, \Pi)$ we denote an unfounded set on $M$ w.r.t. $\Pi$. It is common to identify logic rules of a program with sets of clauses. By $\Pi^{cl}$ we denote the set of clauses corresponding to the rules of $\Pi$. For instance, let $\Pi$ be (2), then $\Pi^{cl}$ consists of clauses $a \vee \neg a, a \vee b$.

The set of nodes of $\text{AS}_{\Pi}$ consists of the states relative to the vocabulary of program $\Pi$. The edges of the graph $\text{AS}_{\Pi}$ are specified by the transition rules of the graph $\text{DP}_{\Pi^{cl}}$ and the rules presented in Figure 7.

For a program $\Pi$, by $\mu(\text{AS}_{\Pi})$ we denote the graph (abstract module) constructed from $\text{AS}_{\Pi}$ by removing all nodes that contain decision literals.

$$\textit{Unfounded}_{\Pi}: \quad M \longrightarrow M\ \neg a \text{ if } \begin{cases} a \in U(M,\Pi) \text{ and} \\ \neg a \text{ is unassigned by } M \end{cases}$$

$$\textit{Fail}_{\Pi}: \quad M \longrightarrow \bot \text{ if } \begin{cases} a \in U(M,\Pi), a \in M, \text{ and} \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_{\Pi}: \quad P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l} \text{ if } \begin{cases} a \in U(P\ l\ Q,\Pi), a \in P\ l\ Q, \text{ and} \\ Q \text{ contains no decision literals} \end{cases}$$

Fig. 7: Transition rules of the graph $\mathrm{AS}_{\Pi}$.

**Proposition 3.** *For every program $\Pi$, the graph $\mu(\mathrm{AS}_{\Pi})$ is a sound abstract module equivalent to a program $\Pi$ under the answer set semantics. Furthermore, the graphs $\mathrm{AM}_{\mu(\mathrm{AS}_{\Pi})}$ and $\mathrm{AS}_{\Pi}$ are identical.*

From Theorem 1 and the fact that $\mu(\mathrm{AS}_{\Pi})$ is an abstract module equivalent to an answer-set program $\Pi$ it follows that the graph $\mathrm{AS}_{\Pi}$ can be used for deciding whether $\Pi$ has an answer set. It is enough to find a path in $\mathrm{AS}_{\Pi}$ leading from the node $\emptyset$ to a terminal node $M$. If $M = \bot$ then $\Pi$ has no answer sets; otherwise, $M$ is an answer set of $\Pi$.

**Corollary 2.** *For any program $\Pi$,*

*(a) graph $\mathrm{AS}_{\Pi}$ is finite and acyclic,*
*(b) for any terminal state $M$ of $\mathrm{AS}_{\Pi}$ other than $\bot$, $M^{+}$ is an answer set of $\Pi$,*
*(c) state $\bot$ is reachable from $\emptyset$ in $\mathrm{AS}_{\Pi}$ if and only if $\Pi$ has no answer sets.*

Let $\Pi$ be the program (2). Figure 3(b) presents the module $\mu(\mathrm{AS}_{\Pi})$. It is easy to see that this module is equivalently contained in the saturated module for $\Pi$ presented in Figure 2. For program $\Pi$ the inference rules of *UnitPropagate* and *Unfounded* are capable to capture all but one inference due to the entailment (the missing inference corresponds to the edge from $b$ to $\neg a\ b$ in Figure 2).

Let us now consider the graph $\mathrm{AS}^{-}_{\Pi}$ constructed from $\mathrm{AS}_{\Pi}$ by either dropping the rules $\textit{Unfounded}_{\Pi}$, $\textit{Backtrack}_{\Pi}$, $\textit{Fail}_{\Pi}$ or the rules $\textit{UnitPropagate}_{\Pi^{cl}}$, $\textit{Backtrack}_{\Pi^{cl}}$, $\textit{Fail}_{\Pi^{cl}}$. In each case, the module $\mu(\mathrm{AS}^{-}_{\Pi})$ in general is not equivalent to a program $\Pi$. This demonstrates the importance of two kinds of inferences for the case of logic programs: (i) those stemming from unit propagate and related to the fact that an answer set of a program is also its classical model; as well as (ii) those based on the concept of "unfoundedness" and related to the fact that every answer set of a program contains no unfounded sets. We note that forward chaining mentioned in earlier section is subsumed by unit propagate.

The graph $\mathrm{AS}_{\Pi}$ is inspired by the graph $\mathrm{SM}_{\Pi}$ introduced by Lierler [11] for specifying an answer set solver SMODELS [19]. The graph $\mathrm{SM}_{\Pi}$ extends $\mathrm{AS}_{\Pi}$ by two additional transition rules (inference rules or propagators): *All Rules Cancelled* and *Backchain True*. We chose to start the presentation with the graph $\mathrm{AS}_{\Pi}$ for its simplicity. We now recall the definition of $\mathrm{SM}_{\Pi}$ and illustrate how a similar result to Proposition 3 is applicable to it.

If $B$ is a conjunction of literals then by $\overline{B}$ we understand the set of the complements of literals occurring in $B$.

The set of nodes of $\mathrm{SM}_\Pi$ consists of the states relative to the vocabulary of program $\Pi$. The edges of the graph $\mathrm{SM}_\Pi$ are specified by the transition rules of the graph $\mathrm{AS}_\Pi$ and the following rules:

*All Rules Cancelled* : $M \longrightarrow M\ \neg a$ if $\begin{cases} \overline{B} \cap M \neq \emptyset \text{ for all } B \in Bodies(\Pi, a) \\ \neg a \text{ is unassigned by } M \end{cases}$

*Fail ARC*: $M \longrightarrow \bot$ if $\begin{cases} \overline{B} \cap M \neq \emptyset \text{ for all } B \in Bodies(\Pi, a), \\ a \in M,\ M \text{ contains no decision literals} \end{cases}$

*Backtrack ARC*: $P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l}$ if $\begin{cases} \overline{B} \cap M \neq \emptyset \text{ for all } B \in Bodies(\Pi, a), \\ a \in P\ l\ Q,\ Q \text{ contains no decision literals} \end{cases}$

*Backchain True* : $M \longrightarrow M\ l$ if $\begin{cases} a \leftarrow B \in \Pi,\ a \in M,\ l \in B \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in Bodies(\Pi, a) \setminus B \\ l \text{ is unassigned by } M \end{cases}$

*Fail BT*: $M \longrightarrow \bot$ if $\begin{cases} a \leftarrow B \in \Pi,\ a \in M,\ l \in B \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in Bodies(\Pi, a) \setminus B \\ l \in M,\ M \text{ contains no decision literals} \end{cases}$

*Backtrack BT*: $P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l}$ if $\begin{cases} a \leftarrow B \in \Pi,\ a \in P\ l\ Q,\ l' \in B \\ \overline{B'} \cap P\ l\ Q \neq \emptyset \text{ for all } B' \in Bodies(\Pi, a) \setminus B \\ l' \in P\ l\ Q,\ Q \text{ contains no decision literals} \end{cases}$

The graph $\mathrm{SM}_\Pi$ shares the important properties of the graph $\mathrm{AS}_\Pi$. Indeed, Proposition 3 and Corollary 2 hold if one replaces $\mathrm{AS}_\Pi$ with $\mathrm{SM}_\Pi$. Corollary 2 in this form was one of the main results stated in [11][4].

Let $\Pi$ be the program (2). Figure 3(b) presents the module $\mu(\mathrm{AS}_\Pi)$. The module $\mu(\mathrm{SM}_\Pi)$ coincides with the saturated module for $\Pi$ presented in Figure 2. For program $\Pi$, the inference rule *Backchain True* captures the inference that corresponds to the edge from $b$ to $\neg a\ b$, which the transition rules of the graph $\mathrm{AS}_\Pi$ are incapable to capture.

The examples above show that the framework of abstract modules uniformly encompasses different logics. We illustrated this point by means of propositional logic and answer-set programming. Furthermore, it uniformly models diverse reasoning mechanisms (entailment and its proof theoretic specializations). The results also demonstrate that transition systems proposed earlier to represent and analyze SAT and ASP solvers are special cases of general transition systems for abstract modules introduced here.

[4] In [11], Lierler presented the $\mathrm{SM}_\Pi$ graph in a slightly different from: the states of that graph permitted inconsistent states of literals, which in turn allowed to unify the *Fail* and *Backtrack* transition rules for different propagators.

## 4 Abstract Modular System and Solver $\text{AMS}_{\mathcal{A}}$

By capturing diverse logics in a single framework, abstract modules are well suited for studying modularity in declarative formalisms, and principles underlying solvers for modular declarative formalisms. We now define an abstract modular declarative framework that uses the concept of a module as its basic element. We then show how abstract transition systems for modules generalize to the new formalism.

An *abstract modular system* (AMS) is a set of modules. The vocabulary of an AMS $\mathcal{A}$ is the union of the vocabularies of modules of $\mathcal{A}$ (they do not have to have the same vocabulary); we denote it by $\sigma(\mathcal{A})$.

An interpretation $I$ over $\sigma(\mathcal{A})$ (that is, a subset of $\sigma(\mathcal{A})$) is a *model* of $\mathcal{A}$, written $I \models \mathcal{A}$, if $I$ is a model of every module $S \in \mathcal{A}$. An AMS $\mathcal{A}$ *entails* a formula $\varphi$ (over the same vocabulary as $\mathcal{A}$), written $\mathcal{A} \models \varphi$, if for every model $I$ of $\mathcal{A}$ we have $I \models \varphi$. We say that an AMS $\mathcal{A}$ is *sound* if every module $S \in \mathcal{A}$ is sound.

Let $S_1$ be a module presented in Figure 1(a) and $S_2$ be a module in Figure 3(b). The vocabulary of the AMS $\{S_1, S_2\}$ consists of the atoms $a$ and $b$. It is easy to see that the interpretation $\{a, \neg b\}$ is its only model.

For a vocabulary $\sigma$ and a set of literals $M$, by $M|_\sigma$ we denote the maximal subset of $M$ consisting of literals over $\sigma$. For example, $\{\neg a, \neg b\}|_{\{a\}} = \{\neg a\}$.

Each AMS $\mathcal{A}$ determines its *transition system* $\text{AMS}_{\mathcal{A}}$. The set of nodes of $\text{AMS}_{\mathcal{A}}$ consists of the states relative to $\sigma(\mathcal{A})$. The transition rules of $\text{AMS}_{\mathcal{A}}$ comprise the rule *Decide* and the rules *Propagate*$_S$, *Fail*$_S$, and *Backtrack*$_S$, for all modules $S \in \mathcal{A}$. The latter three rules are modified to account for the vocabulary $\sigma(\mathcal{A})$ and are presented in Figure 8.

$$\textit{Propagate}_S: \quad M \longrightarrow M\ l \text{ if } S \text{ has an edge from } M|_{\sigma(S)} \text{ to } M\ l|_{\sigma(S)}$$

$$\textit{Fail}_S: \quad M \longrightarrow \bot \text{ if } \begin{cases} S \text{ has an edge from } M|_{\sigma(S)} \text{ to } \bot, \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_S: \quad P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l} \text{ if } \begin{cases} S \text{ has an edge from } P\ l\ Q|_{\sigma(S)} \text{ to } \bot, \\ Q \text{ contains no decision literals} \end{cases}$$

Fig. 8: The transition rules of the graph $\text{AMS}_{\mathcal{A}}$.

**Theorem 2.** *For every sound AMS $\mathcal{A}$,*

*(a) the graph $\text{AMS}_{\mathcal{A}}$ is finite and acyclic,*
*(b) any terminal state of $\text{AMS}_{\mathcal{A}}$ other than $\bot$ is a model of $\mathcal{A}$,*
*(c) the state $\bot$ is reachable from $\emptyset$ in $\text{AMS}_{\mathcal{A}}$ if and only if $\mathcal{A}$ is unsatisfiable.*

This theorem demonstrates that to decide a satisfiability of a sound AMS $\mathcal{A}$ it is sufficient to find a path leading from node $\emptyset$ to a terminal node. It provides a foundation for the development and analysis of solvers for modular systems.

For instance, let $\mathcal{A}$ be the AMS $\{S_1, S_2\}$. Below is a valid path in the transition graph $\text{AMS}_{\mathcal{A}}$ with every edge annotated by the corresponding transition rule:

$$\emptyset \overset{\textit{Decide}}{\longrightarrow} \neg a^{\Delta} \overset{\textit{Propagate}_{S_2}}{\longrightarrow} \neg\, a^{\Delta}\, b \overset{\textit{Backtrack}_{S_1}}{\longrightarrow} a \overset{\textit{Decide}}{\longrightarrow} a\, \neg b^{\Delta}.$$

The state $a\ \neg b^{\Delta}$ is terminal. Thus, Theorem 2 (b) asserts that $\{a, \neg b\}$ is a model of $\mathcal{A}$. Let us interpret this example. Earlier we demonstrated that module $S_1$ can be regarded as a representation of a propositional theory consisting of a single clause $a$ whereas $S_2$ corresponds to the logic program (2) under the semantics of answer sets. We then illustrated how modules $S_1$ and $S_2$ give rise to particular algorithms for implementing search procedures. The graph $\text{AMS}_{\mathcal{A}}$ represents the algorithm obtained by *integrating* the algorithms supported by the modules $S_1$ and $S_2$ separately.

The results presented above imply, as special cases, earlier results on the logics PC(ID) and SM(ASP), and their solvers [14].

## 5 Learning in Solvers for AMSs.

Nieuwenhuis et al. [20, Section 2.4] defined the *DPLL System with Learning* graph to describe SAT solvers' learning, one of the crucial features of current SAT solvers responsible for rapid success in this area of automated reasoning. The approach of Nieuwenhuis, Oliveras, and Tinelli extends to our abstract setting. Specifically, the graph $\text{AMS}_{\mathcal{A}}$ can be extended with "learning transitions" to represent solvers for AMSs that incorporate learning.

The intuition behind learning in SAT is to allow new propagations by extending the original clause database as computation proceeds. These "learned" clauses provide new "immediate derivations" to a SAT solver by enabling additional applications of *UnitPropagate*. In the framework of abstract modules, immediate derivations are represented by edges. Adding edges to modules captures the idea of learning by supporting new propagations that the transition rule *Propagate* may take an advantage of. We now state these intuitions formally for the case of abstract modular systems.

Let $S$ be a module and $E$ a set of edges between nodes of $S$. By $S^E$ we denote the module constructed by adding to $S$ the edges in $E$. A set $E$ of edges is *$S$-safe* if the module $S^E$ is sound and equivalent to $S$. For an AMS $\mathcal{A}$ and a set of edges $E$ over the vocabulary of $\mathcal{A}$, we define $\mathcal{A}^E = \{S^{E_{|S}} : S \in \mathcal{A}\}$ (where $E_{|S}$ is the set of those edges in $E$ that connect nodes in $S$). We say that $E$ is *$\mathcal{A}$-safe* if $\mathcal{A}$ and $\mathcal{A}^E$ are *equivalent*, and each module $S^E$ in $\mathcal{A}^E$ is sound.

An *(augmented) state* relative to an AMS $\mathcal{A} = \{S_1, \ldots, S_n\}$ is either a distinguished state $\bot$ or a pair of the form $M||\Gamma_1, \ldots, \Gamma_n$ where $M$ is an *ordered* consistent set $M$ of literals over $\sigma$, some possibly annotated by $\Delta$; and $\Gamma_1, \ldots, \Gamma_n$ are sets of edges between nodes of modules $S_1, \ldots, S_n$, respectively. Sometimes we denote $\Gamma_1, \ldots, \Gamma_n$ by $\mathcal{G}$. For any AMS $\mathcal{A} = \{S_1, \ldots, S_n\}$, we define a graph $\text{AMSL}_{\mathcal{A}}$. Its nodes are the augmented states relative to $\mathcal{A}$. The rule *Decide* of the $\text{AMS}_{\mathcal{A}}$ graph extends to $\text{AMSL}_{\mathcal{A}}$ as follows

$$\textit{Decide}: \quad M||\mathcal{G} \longrightarrow M\ l^{\Delta}||\mathcal{G} \ \text{ if } \ l \text{ is unassigned by } M.$$

Figure 9 presents the transition rules of $\text{AMSL}_{\mathcal{A}}$ that are specific to each module $S_i$ in $\mathcal{A}$. We note that the set $E$ of edges in the rule *Learn Local*$_{S_i}$ is required to consist of

edges that run between the nodes of $S_i$. The transition rule

$$\textit{Learn Global}: \quad M||\ldots,\Gamma_j,\ldots \longrightarrow M||\ldots,\Gamma_j \cup E_{|S_i},\ldots \text{ if } E \text{ is } \mathcal{A}\text{-safe}$$

where $E$ is a set of edges between nodes over the vocabulary $\sigma(\mathcal{A})$, concludes the definition of $\text{AMSL}_{\mathcal{A}}$.

$$\textit{Propagate}_{S_i}: \quad M||\mathcal{G} \longrightarrow M\ l||\mathcal{G} \text{ if } S_i^{\Gamma_i} \text{ has an edge from } M \text{ to } M\ l$$

$$\textit{Fail}_{S_i}: \quad M||\mathcal{G} \longrightarrow \bot \text{ if } \begin{cases} S_i^{\Gamma_i} \text{ has an edge from } M \text{ to } \bot, \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_{S_i}: \quad P\ l^{\Delta}\ Q||\mathcal{G} \longrightarrow P\ \bar{l}||\mathcal{G} \text{ if } \begin{cases} S_i^{\Gamma_i} \text{ has an edge from } P\ l\ Q \text{ to } \bot, \\ Q \text{ contains no decision literals} \end{cases}$$

$$\textit{Learn Local}_{S_i}: \quad M||\ldots,\Gamma_i,\ldots \longrightarrow M||\ldots,\Gamma_i \cup E,\ldots \text{ if } E \text{ is } S_i\text{-safe}$$

Fig. 9: Transition rules of $\text{AMSL}_{\mathcal{A}}$ for module $S_i \in S$.

We refer to the transition rules *Propagate*, *Backtrack*, *Decide*, and *Fail* of the graph $\text{AMSL}_{\mathcal{A}}$ as *basic*. We say that a node in the graph is *semi-terminal* if no basic rule is applicable to it. The graph $\text{AMSL}_{\mathcal{A}}$ can be used for deciding whether an AMS $\mathcal{A}$ has an answer set by constructing a path from $\emptyset||\emptyset,\ldots,\emptyset$ to a semi-terminal node.

**Theorem 3.** *For any sound AMS $\mathcal{A}$,*

*(a) there is an integer $m$ such that every path in $\text{AMSL}_{\mathcal{A}}$ contains at most $m$ edges due to basic transition rules,*
*(b) for any semi-terminal state $M||\mathcal{G}$ of $\text{AMSL}_{\mathcal{A}}$ reachable from $\emptyset||\emptyset,\ldots,\emptyset$, $M$ is a model of $\mathcal{A}$,*
*(c) state $\bot$ is reachable from $\emptyset||\emptyset,\ldots,\emptyset$ in $\text{AMSL}_{\mathcal{A}}$ if and only if $\mathcal{A}$ has no models.*

It follows that if we are constructing a path starting in $\emptyset||\emptyset,\ldots,\emptyset$ in a way that guarantees that every sequence of consecutive edges of the path labeled with *Learn Local* and *Learn Global* eventually ends (is finite), then the path will reach some semi-terminal state. As soon as a semi-terminal state is reached the problem of finding a model is solved.

There is an important difference between *Learn Local* and *Learn Global*. The first one allows new propagations within a module but does not change its semantics as the models of the module stay the same (and it is local, other modules are unaffected by it). The application of *Learn Global* while preserving the overall semantics of the system may change the semantics of individual modules by eliminating some of their models (and, being global, affects in principle all modules of the system). SAT researchers have demonstrated that *Learn Local* is crucial for the success of SAT technology both in practice and theoretically. Our initial considerations suggest that under some circumstances, *Learn Global* offers additional substantial performance benefits.

We stress that our discussion of learning does not aim at any specific algorithmic ways in which one could perform learning. Instead, we formulate conditions that learned edges are to satisfy ($S$-safety for learning local to a module $S$, and $\mathcal{A}$-safety for the global learning rule), which ensure the correctness of solvers that implement learning so that to satisfy them. In this way, we provide a uniform framework for correctness proofs of multi-logic solvers incorporating learning.

## 6 Related Work

In an important development, Brewka and Eiter [3] introduced an abstract notion of a *heterogeneous nonmonotonic multi-context system* (MCS). One of the key aspects of that proposal is its abstract representation of a logic and hence contexts that rely on such abstract logics. The independence of contexts from syntax promoted focus on semantic aspect of modularity exhibited by multi-context systems. Since their inception, multi-context systems have received substantial attention and inspired implementations of hybrid reasoning systems including DLVHEX [5] and DMCS [1]. Abstract modular systems introduced here are similar to MCSs as they too do not rely on any particular syntax for logics assumed in modules (a counterpart of a context). What distinguishes them is that they encapsulate some semantic features stemming from inferences allowed by the underlying logic. This feature of abstract modules is essential for our purposes as we utilize them as a tool for studying algorithmic aspects of multi-logic systems. Another difference between AMS and MCS is due to "bridge rules." Bridge rules are crucial for defining the semantics of an MCS. They are also responsible for "information sharing" in MCSs. They are absent in our formalism altogether. In AMS information sharing is implemented by a simple notion of a shared vocabulary between the modules.

Modularity is one of the key techniques in principled software development. This has been a major trigger inspiring research on modularity in declarative programming paradigms rooting in KR languages such as answer set programming, for instance. Oikarinen and Janhunen [21] proposed a modular version of answer set programs called lp-modules. In that work, the authors were primarily concerned with the decomposition of lp-modules into sets of simpler ones. They proved that under some assumptions such decompositions are possible. Järvisalo, Oikarinen, Janhunen, and Niemelä [9], and Tasharrofi and Ternovska [24] studied the generalizations of lp-modules. In their work the main focus was to abstract lp-modules formalism away from any particular syntax or semantics. They then study properties of the modules such as "joinability" and analyze *different ways* to join modules together and the semantics of such a join. We are interested in building simple modular systems using abstract modules – the only composition mechanism that we study is based on conjunction of modules. Also in contrast to the work by Järvisalo et al. [9] and Tasharrofi and Ternovska [24], we define such conjunction for any modules disregarding their internal structure and interdepencies between each other.

Tasharrofi, Wu, and Ternovska [25] developed and studied an algorithm for processing modular model expansion tasks in the abstract multi-logic system concept developed by Tasharrofi and Ternovska [24]. They use the traditional pseudocode method to present the developed algorithm. In this work we adapt the graph-based framework

for designing backtrack search algorithms for abstract modular systems. The benefits of that approach for modeling families of backtrack search procedures employed in SAT, ASP, and PC(ID) solvers were demonstrated by Nieuwenhuis et al. [20], Lierler [11], and Lierler and Truszczynski [14]. Our work provides additional support for the generality and flexibility of the graph-based framework as a finer abstraction of backtrack search algorithms than direct pseudocode representations, allowing for convenient means to prove correctness and study relationships between the families of the algorithms.

## 7 Conclusions

We introduced abstract modules and abstract modular systems and showed that they provide a framework capable of capturing diverse logics and inference mechanisms integrated into modular knowledge representation systems. In particular, we showed that transition graphs determined by modules and modular systems provide a unifying representation of model-generating algorithms, or solvers, and simplify reasoning about such issues as correctness or termination. We believe they can be useful in theoretical comparisons of solver effectiveness and in the development of new solvers. Learning, a fundamental technique in solver design, displays itself in two quite different flavors, local and global. The former corresponds to learning studied before in SAT and SMT and demonstrated both theoretically and practically to be essential for good performance. Global learning is a new concept that we identified in the context of modular systems. It concerns learning *across* modules and, as local learning, promises to lead to performance gains. In the future work we will conduct a systematic study of global learning in abstract modular systems and its impact on solvers for practical multi-logic formalisms.

## References

1. Bairakdar, S.E.D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The dmcs solver for distributed nonmonotonic multi-context systems. In: 12th European Conference on Logics in Artificial Intelligence (JELIA). pp. 352–355 (2010)
2. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsch, T. (eds.) Handbook of Satisfiability, pp. 737–797. IOS Press (2008)
3. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proceedings of National conference on Artificial Intelligence (AAAI). pp. 385–390 (2007)
4. Denecker, M., Lierler, Y., Truszczynski, M., Vennekens, J.: A Tarskian informal semantics for answer set programming. In: Dovier, A., Costa, V.S. (eds.) International Conference on Logic Programming (ICLP). LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
5. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer set programming. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). pp. 90–96 (2005)
6. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium. pp. 1070–1080. MIT Press (1988)

7. Giunchiglia, F.: Contextual reasoning. Epistemologia XVI, 345–364 (1993)
8. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19(20), 503–581 (1994)
9. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 155–168. LPNMR '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04238-6_15
10. Lee, J.: A model-theoretic counterpart of loop formulas. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). pp. 503–508. Professional Book Center (2005)
11. Lierler, Y.: Abstract answer set solvers. In: Proceedings of International Conference on Logic Programming (ICLP). pp. 377–391. Springer (2008)
12. Lierler, Y.: Abstract answer set solvers with backjumping and learning. Theory and Practice of Logic Programming 11, 135–169 (2011)
13. Lierler, Y.: On the relation of constraint answer set programming languages and algorithms. In: Proceedings of the AAAI Conference on Artificial Intelligence. MIT Press (2012)
14. Lierler, Y., Truszczynski, M.: Transition systems for model generators — a unifying approach. Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue 11, issue 4-5 (2011)
15. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer Verlag (1999)
16. Mariën, M., Wittocx, J., Denecker, M., Bruynooghe, M.: SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In: SAT. pp. 211–224 (2008)
17. McCarthy, J.: Generality in Artificial Intelligence. Communications of the ACM 30(12), 1030–1035 (1987), reproduced in [**?**]
18. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
19. Niemelä, I., Simons, P.: Extending the Smodels system with cardinality and weight constraints. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 491–521. Kluwer (2000)
20. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
21. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: 17th European Conference on Artificial Intelligence(ECAI). pp. 412–416 (2006)
22. Rossi, F., van Beek, P., Walsh, T.: Constraint programming. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 181–212. Elsevier (2008)
23. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138, 181–234 (2002)
24. Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: Frontiers of Combining Systems, 8th International Symposium (FroCoS). pp. 259–274 (2011)
25. Tasharrofi, S., Wu, X.N., Ternovska, E.: Solving modular model expansion tasks. CoRR abs/1109.0583 (2011)
26. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of ACM 38(3), 620–650 (1991)

# Negation in the Head of CP-logic Rules

Joost Vennekens

joost.vennekens@cs.kuleuven.be
Dept. Computerscience — Campus De Nayer
KU Leuven

**Abstract.** CP-logic is a probabilistic extension of the logic FO(ID). Unlike ASP, both of these logics adhere to a Tarskian informal semantics, in which interpretations represent objective states-of-affairs. In other words, these logics lack the epistemic component of ASP, in which interpretations represent the beliefs or knowledge of a rational agent. Consequently, neither CP-logic nor FO(ID) have the need for two kinds of negations: there is only one negation, and its meaning is that of objective falsehood. Nevertheless, the formal semantics of this objective negation is mathematically more similar to ASP's negation-as-failure than to its classical negation. The reason is that both CP-logic and FO(ID) have a constructive semantics in which all atoms start out as false, and may only become true as the result of a rule application. This paper investigates the possibility of adding the well-known ASP feature of allowing negation in the head of rules to CP-logic. Because CP-logic only has one kind of negation, it is of necessity this "negation-as-failure like" negation that will be allowed in the head. We investigate the intuitive meaning of such a construct and the benefits that arise from it.

## 1 Introduction

This paper is part of a long-term research project that aims to develop a *Tarskian view* on Answer Set Programming (ASP). Historically, the origins of ASP lie in the seminal papers by Gelfond and Lifschitz on the stable semantics for normal (1988) and extended logic programs (1991). These papers develop an *epistemic view* on logic programs, in which an answer set is seen as an exhaustive enumeration of a rational agent's atomic beliefs. In this view, an atom $A$ belonging to an answer set $X$ means that the agent believes $A$; $A \notin X$ means that $A$ is not believed; and $\neg A \in X$ means that a is believed to be false. A rule such as:

$$A \leftarrow B_1, \ldots, B_n, \texttt{not } C_1, \ldots, \texttt{not } C_m. \tag{1}$$

tells the agent that if he believes all of the $B_i$ and does not believe any of the $C_j$, he should believe $A$. In addition, the agent also obeys the *rationality principle*, believing only what he has reason to believe. The stable model semantics then computes what a perfectly rational agent would believe under all these rules.

While these epistemic intuitions have played a crucial role in the history of ASP, current practice seems to have largely drifted away from them. In particular,

programs written according to the currently prevalent *Generate-Define-Test*-methodology (GDT) (term coined by Lifschitz, 2002) are typically no longer explicitly concerned with the beliefs of an agent. A typical example is the *graph colouring* problem, in which we *generate* the search space of all assignments of colours to nodes, we *define* that two nodes are in conflict if they share an edge and have the same colour, and then *test* that there are no conflicts. Unlike early ASP examples—such as, e.g., Gelfond's example (1991) of interviewing all students for which we do not *know* whether they are eligible for a grant—the statement of the graph colouring problems is not concerned with anyone's knowledge or beliefs, but only with the objective colour of the nodes.

Suppose now that we have an ASP representation of a purely objective GDT problem, such as graph coloring. How should we intuitively interpret this program? Falling back on the papers by Gelfond and Lifschitz, every single statement in the program will be interpreted as an epistemic statement about some agent's knowledge. Obviously, this is a poor match with the objective intuitions behind the problem. Therefore, an alternative informal semantics is needed, which omits this agent, and explains how rules of the program can be interpreted as statements about the real world, in this same way as formulas in classical first-order logic (FO) are. There are now two important and related questions:

- If we view a semantical object such as an answer set as a representation of an objective state of the world, instead of some agent's beliefs, how should we then interpret a rule such as (1)?
- How does this objective interpretation of ASP compare to the classical way of representing such objective information about the world, namely FO?

An extensive study of these two questions has been performed by Denecker and several coauthors. Recent summaries of these results were published by Denecker et al. (2010) and Denecker et al. (2012). A goal of this research program is to reconstruct ASP as a series of conservative extensions of FO. One of its main achievements has been the development of the language of FO(ID) (Denecker and Ternovska, 2007), which extends FO with a construct for representing *inductive definitions*. FO(ID) can be seen as a variant of ASP, which adheres to a strict objective interpretation of its semantical constructs, i.e., a model of an FO(ID) theory does not represent beliefs, but an objective state of the world.

The language of FO(ID) has been further extended in many ways. This paper is concerned with one particular such extension, namely, *CP-logic* (Vennekens et al., 2009), which extends the inductive definition construct of FO(ID) with a means for expressing non-deterministic choice. One application is to represent non-deterministic inductive definitions. For instance, an execution trace of a non-deterministic Turing machine may be defined by means of a rule that states that if the machine reads a character $c$ in a state $s$ at time $\alpha$, it will be in a state $s'$ at time $\alpha + 1$, where $s'$ is *one of* the states that it may transition to from $(s, c)$. CP-logic represents such non-determinism by allowing disjunction in the head of rules. This is similar in syntax to the kind of rules allowed by, for instance, the DLV language. This is, therefore, another way in which one of ASP's features can be conservatively added to the classical framework. However,

to correctly formalise non-deterministic inductive definitions, not the minimal model semantics must be used, but the *possible world semantics* of Sakama and Inoue (1994).

A more important application of CP-logic, however, is to represent *probabilistic causal laws*. Such relations have received a great deal of attention in the AI community, especially since the influential work by Pearl (2000) on this topic. As shown by Vennekens et al. (2010), CP-logic can actually be seen as a refinement of Pearl's theory, which allows for a more compact and modular representation of certain phenomena. As an example, consider three gear wheels, each of which has an attached crank that can be used to turn it. The first gear wheel is connected to the second, which is in turn connected to the third, so that in 90% of the cases, when one turns the other also turns; however, there is some damage to the gear wheels' teeth, which in 10% of the cases prevents this. In CP-logic, we can represent this by means of seven independent probabilistic causal laws:

$$Turns(Gear1) \leftarrow Crank_1. \tag{2}$$
$$Turns(Gear2) \leftarrow Crank_2. \tag{3}$$
$$Turns(Gear3) \leftarrow Crank_3. \tag{4}$$
$$(Turns(Gear1) : 0.9) \leftarrow Turns(Gear2). \tag{5}$$
$$(Turns(Gear2) : 0.9) \leftarrow Turns(Gear1). \tag{6}$$
$$(Turns(Gear2) : 0.9) \leftarrow Turns(Gear3). \tag{7}$$
$$(Turns(Gear3) : 0.9) \leftarrow Turns(Gear2). \tag{8}$$

By contrast, Pearl would represent it in a less modular way, by means of three structural equations, each of which defines precisely when a particular gear wheel will turn :

$$Turns(Gear1) := Crank_1 \vee (Crank_2 \wedge Trans_{1,2}) \vee (Crank_3 \wedge Trans_{3,2} \wedge Trans_{2,1})$$
$$Turns(Gear2) := Crank_2 \vee (Crank_1 \wedge Trans_{1,2}) \vee (Crank_3 \wedge Trans_{3,2})$$
$$Turns(Gear3) := Crank_3 \vee (Crank_2 \wedge Trans_{2,3}) \vee (Crank_1 \wedge Trans_{1,2} \wedge Trans_{2,3})$$

CP-logic has certain similarities to P-log, a probabilistic extension of ASP (Baral et al., 2008). However, it differs by its focus on representing individual probabilistic causal laws, as discussed by Vennekens et al. (2010, 2009).

As this example illustrates, a causal law in CP-logic may cause some atom(s) to become true, and it may also fail to do so. What is currently not possible, however, is that such a laws causes an atom to be false. For instance, suppose that the first gear wheel may be locked, in order to prevent it from turning. The current way to represent this would be to replace rules (2) and (5) by:

$$(Turns(Gear1) : 0.9) \leftarrow Crank_1 \wedge \neg Locked(1).$$
$$(Turns(Gear1) : 0.9) \leftarrow Turns(Gear2) \wedge \neg Locked(1).$$

However, this goes against our desire for a modular representation of the individual causal laws. Our goal in the current paper is to extend CP-logic to allow instead

to keep rules (2) and (5) as they are, and instead add a rule:

$$\neg Turns(Gear1) \leftarrow Locked(1).$$

In other words, we will examine how CP-logic can be extended with the familiar ASP feature of *negation in the head* Gelfond and Lifschitz (1991). Again, the traditional ASP interpretation of a classical negation literal is rooted in the epistemic tradition: whereas `not` $A$ means that $A$ is not believed to be true, a classical negation literal $\neg A$ means that $A$ is believed to be false. Since FO(ID) and CP-logic have no beliefs, the only thing that negation *can* mean in this context is that $A$ is objectively false. Nevertheless, as this paper will show, there is still a place for negation-in-the-head in such a logic. Our two main contributions are therefore as follows:

- By adding this additional feature to CP-logic, we extend its ability to represent causal laws in a modular way, as illustrated by the above example.
- From the point of view of the larger research project, negation-in-the-head is an ASP feature that, until now, could not yet be given a place within the FO(ID)/CP-logic framework and its Tarskian semantics. This paper offers one way in which this gap can be filled.

This paper is structured as follows. First, Section 2 recalls the definition of CP-logic. Section 2.1 elaborates further on the role of negation in the current version of CP-logic, before Section 3 then discusses our proposed extension with negation in the head. Several uses of this new feature are then discussed in Sections 4 to 6. Finally, Section 7 discusses the implementation of this new language feature.

## 2 Preliminaries: CP-logic

A theory in CP-logic consists of a set of rules. These rules are called *causal probabilistic laws*, or *CP-laws* for short, and they are statements of the form:

$$\forall \boldsymbol{x}\ (A_1 : \alpha_1) \vee \cdots \vee (A_n : \alpha_n) \leftarrow \phi. \tag{9}$$

Here, $\phi$ is a first-order formula and the $A_i$ are atoms, such that the tuple of variables $\boldsymbol{x}$ contains all free variables in $\phi$ and the $A_i$. The $\alpha_i$ are non-zero probabilities with $\sum \alpha_i \leq 1$. Such a CP-law expresses that $\phi$ causes some (implicit) non-deterministic event, of which each $A_i$ is a possible outcome with probability $\alpha_i$. If $\sum_i \alpha_i = 1$, then at least one of the possible effects $A_i$ must result if the event caused by $\phi$ happens; otherwise, it is also possible that the event happens without any (visible) effect on the state of the world. For mathematical uniformity, we introduce the notation $r^{=}$ to refer to $r$ itself if the equality holds, and otherwise to the CP-law:

$$\forall \boldsymbol{x}\ (A_1 : \alpha_1) \vee \cdots \vee (A_n : \alpha_n) \vee (\text{—} : 1 - \sum_i \alpha_i) \leftarrow \phi.$$

Here, the dash is a new symbol that explicitly represents the possibility that none of the effects $A_i$ are caused. Whenever we add this dash to some set $X$, it does not change $X$, i.e., $X \cup \{—\} = X$.

The semantics of a theory in CP-logic is defined in terms of its grounding, so from now on we will restrict attention to ground theories, i.e., we assume that for each CP-law, the tuple of variables $\boldsymbol{x}$ is empty. For now, we also assume that the rule bodies $\phi$ do not contain negation.

For a CP-law $r$, we refer to $\phi$ as the *body* of $r$, and to the sequence $(A_i, \alpha_i)_{i=1}^n$ as the *head* of $r$. We denote these objects as $body(r)$ and $head(r)$, respectively.

In CP-laws of form (9), the precondition $\phi$ may be omitted for events that are vacuously caused. If a CP-law has a deterministic effect, i.e., it is of the form $(A : 1) \leftarrow \phi$, then we also write it simply as $A \leftarrow \phi$.

*Example 1.* Suzy and Billy might each decide to throw a rock at a bottle. If Suzy does so, her rock breaks the bottle with probability 0.8. Billy's aim is slightly worse and his rock only hits with probability 0.6. Assuming that Suzy decides to throw with probability 0.5 and that Billy always throws, this domain corresponds to the following set of causal laws:

$$(Throws(Suzy) : 0.5). \quad (10) \qquad (Broken : 0.8) \leftarrow Throws(Suzy). \quad (12)$$

$$Throws(Billy). \quad (11) \qquad (Broken : 0.6) \leftarrow Throws(Billy). \quad (13)$$

In causal modeling, a distinction is commonly made between endogenous properties, whose values are completely determined by the causal mechanisms described by the model, and exogenous properties, whose values are somehow determined outside the scope of the model. Following this convention, the predicates of a CP-theory are also divided into exogenous and endogenous predicates. We define the semantics of a theory in the presence of a given, fixed interpretation $X$ for the exogenous predicates.

A second common assumption (see e.g. Hall, 2007) is that each of the endogenous properties has some default value, which represents its "natural state". In other words, the *default* value of an endogenous property is the value that it has whenever there are no causal mechanisms acting upon it. The effect of the causal mechanisms in the model is then of course precisely to flip the value of some of the properties from its default to a *deviant* value.

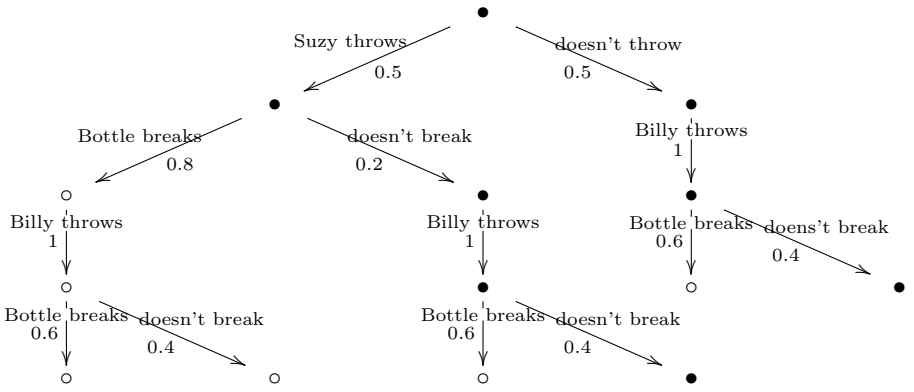
Theories in CP-logic have a straightforward execution semantics. We consider probability trees, in which each node is labeled with an Herbrand interpretation for the endogenous predicates. The root of the tree—i.e., the initial state of our causal process—is labeled with the universally false interpretation $\{\}$. This incorporates our second assumption: w.l.o.g. we force the user to choose his vocabulary in such a way that the default value for each endogenous atom is false. We then constructively extend the tree by applying the following operation as long as possible:

1. Choose a pair $(s, r)$ of a leaf $s$ of the tree and a rule $r$ of the theory, such that $(X \cup \mathcal{I}(s)) \models body(r)$ and there exists no ancestor $s'$ of $s$ such that $(s', r)$ has already been chosen

2. Extend $s$ with children $s_0, \ldots, s_m$, where each $s_i$ corresponds to one of the disjuncts $(h_i : \alpha_i)$ in $head(r^=)$, in the sense that $\mathcal{I}(s_i) = \mathcal{I}_s \cup \{h_i\}$ and the edge from $s$ to $s_i$ is labeled by $\alpha$.

We call a tree $\mathcal{T}$ constructed in this way an *execution model* of the CP-theory under $X$. We define a probability distribution $\pi_{\mathcal{T}}$ over the set of all Herbrand interpretations as: $\pi_{\mathcal{T}}(I) = \sum_{\mathcal{I}(l)=I} \pi_{\mathcal{T}}(l)$, where the sum is taken over all leaves $l$ of $\mathcal{T}$ whose interpretation equals $I$ and the probability $\pi_{\mathcal{T}}(l)$ of such a leaf consists of the product of all probability labels that are encountered on the path to this leaf.

The following picture represents an execution model for the CP-theory of Example 1. The states $s$ in which the bottle is broken (i.e., for which $Broken \in \mathcal{I}(s)$) are represented by an empty circle, and those in which it is still whole by a full one. This pictures does not show the interpretations $\mathcal{I}(s)$; instead, we have just written the effects of each event in natural language as labels on the edges.




The third branch of this execution model consists of five nodes $(s_0, \ldots, s_4)$. The progression of the associated states of the world $(\mathcal{I}(s_0), \ldots, \mathcal{I}(s_4))$ is as follows:

$$(\{\}, \{Throws(Suzy)\}, \{Throws(Suzy)\},$$
$$\{Throws(Suzy), Throws(Billy)\},$$
$$\{Throws(Suzy), Throws(Billy), Broken\}).$$

Note that, in keeping with the Tarskian setting of CP-logic, each interpretation represents an objective state of the world.

Even when starting from the same interpretation $X$ for the exogenous predicates, the same CP-theory may have many execution models, which differ in their selection of a rule to apply in each node (step 1). It was shown by Vennekens et al. (2009) that, because each applicable rule must eventually be applied, the differences between these execution models are irrelevant, as long as we only care about the final states that may be reached. In other words, all execution models $\mathcal{T}$ of the same CP-theory $T$ that start from the same interpretation $X$ generate the same distribution $\pi_{\mathcal{T}}$. We also denote this unique distribution as $\pi_T^X$.

An interesting special case is that in which each rule $r$ is *deterministic*, i.e., it causes a single atom with probability 1. In this case, each execution model is a degenerate tree consisting of a single branch, in which all edges are labeled with probability 1. The successive interpretations in this branch are constructed by adding to the previous interpretation the head of a rule whose body is satisfied. The single leaf of this tree is therefore precisely the least Herbrand model of the set of rules. In this way, positive logic programs and monotone inductive definitions in FO(ID) are embedded in CP-logic.

### 2.1 Negation in CP-logic

Consider again the role that the CP-law

$$(Broken : 0.9) \leftarrow Throws(Suzy)$$

plays in the above execution model. Initially, when the atom $Throws(Suzy)$ is still at its default, this law is dormant. Once $Throws(Suzy)$ has been caused, this law becomes active and will (eventually) be executed, causing $Broken$ with probability 0.9. Now, suppose we had instead assumed that the default is for Suzy to throw unless she decides to refuse:

$$(Broken : 0.9) \leftarrow \neg RefusesThrow(Suzy).$$
$$(RefusesThrow(Suzy) : 0.5).$$

Under the semantics given so far, this first CP-law would be active in *any* state where $RefusesThrow(Suzy)$ has not deviated from its default. For instance, this law would always be active in the initial state. This means that there would be an execution model in which this law first causes the bottle to break and then, afterwards, Suzy decides to refuse the throw. Such execution models are not very meaningful, or useful.

For this reason, when allowing negation, an additional condition is imposed on the execution models of a CP-theory. The basic idea is to read $\neg A$ not simply as "$A$ is currently at its default vaue", but instead as "$A$ will not deviate from its default". Under this interpretation, the law will only become active once our causal process is far enough along to be able to say with certainty that no deviation will occur. For the above example, this would mean that the first CP-law can only become active *after* the second one has taken place and has failed to cause $RefusesThrow(Suzy)$.

This idea is formalized by means of concepts from three-valued logic, where atoms can be unknown (**u**) in addition to true (**t**) or false (**f**). Given a three-valued interpretation $\nu$, that assigns one of these three truth values to each atom, the standard Kleene truth tables can be used to assign a corresponding truth value $\nu(\phi)$ to each formula $\phi$. A two-valued interpretation $I$ is said to be approximated by a three-valued interpretation $\nu$ if it can be constructed from it by switching atoms from **u** to **t** or **f**. If $I$ is approximated by $\nu$, then for each formula $\phi$, the truth value $\nu(\phi)$ also approximates the truth of $\phi$ according to $I$; that is, if $\nu(\phi) = \mathbf{t}$ then $I \models \phi$ and if $\nu(\phi) = \mathbf{f}$ then $I \not\models \phi$.

Now, for each state $s$ of an execution model, we construct an overestimate of the set of atoms that might still be caused in the part of the tree following $s$. First, the set of events that could potentially happen in this state itself is $Pot(s) = \{r \in \mathcal{R}(s) \mid \mathcal{I}(s) \models body(r)\}$, where $\mathcal{R}(s)$ denotes the set of all rules that have not yet happened in the ancestors of $s$. For each child $s'$ of $s$, $\mathcal{I}(s')$ will therefore differ from $\mathcal{I}(s)$ by including at most one atom $A \notin \mathcal{I}(s)$ from the head of one of the rules $r \in Pot(s)$. Therefore, if we construct a three-valued interpretation $\nu_0$ that labels all such atoms $A$ as $\mathbf{u}$ and coincides with $\mathcal{I}(s)$ on all other atoms, then we end up with an approximation of each $\mathcal{I}(s')$ for which $s'$ is a child of $s$. Now, if an event $r$ is to happen in one of these children $s'$ of $s$, then it must be the case that that $\mathcal{I}(s') \models body(r)$, which implies that $\nu_1(body(r)) \neq \mathbf{f}$. We now derive a $\nu_2$ from $\nu_1$ by turning into $\mathbf{u}$ all atoms $A$ for which $\nu_1(A) = \mathbf{f}$ and $A$ appears in the head of an $r$ for which $\nu_0(body(r)) \neq \mathbf{f}$. This $\nu_2$ is then an approximation of all $\mathcal{I}(s'')$ for which $s''$ is a grandchild of $s$. We can now iterate this principle and construct a sequence $(\nu_1, \nu_2, \ldots)$ of three-valued interpretations, where each $\nu_i$ approximates all the $\mathcal{I}(t)$ for which $t$ is a descendant of $s$, separated from $s$ by at most $i-1$ intermediary nodes. This process will make more and more atoms $\mathbf{u}$, until eventually it reaches a fixpoint, which we denote as $\mathcal{U}(s)$. This fixpoint approximates all the $\mathcal{I}(t)$ for which $t$ is a descendant of $s$. Therefore, if an atom is $\mathbf{f}$ in $\mathcal{U}(s)$, then it will not be caused anywhere below $s$.

To illustrate, consider the rightmost branch $(s'_0, s'_1, \ldots, s'_3)$ of the execution model shown in Section 2. The associated three-valued interpretations are as follows, where we abbreviate $Throws$ and $Broken$ by $T$ and $B$, and $Billy$ and $Suzy$ by $By$ and $Sy$.

| Node $s$ | $\mathcal{U}(s)$ | | |
|---|---|---|---|
| | **t** | **u** | **f** |
| $s'_0$ | $\{\}$ | $\{T(Sy), T(By), B\}$ | $\{\}$ |
| $s'_1$ | $\{\}$ | $\{T(By), B\}$ | $\{T(Sy)\}$ |
| $s'_2$ | $\{T(By)\}$ | $\{B\}$ | $\{T(Sy)\}$ |
| $s'_3$ | $\{T(By)\}$ | $\{\}$ | $\{T(Sy), B\}$ |

The following additional condition is now imposed on the execution models of a CP-theory:

> *For a rule $r$ to be allowed to happen in a node $s$, it is not enough that simply $\mathcal{I}(s) \models body(r)$; in addition, it must also be the case that the truth value of $body(r)$ according to $\mathcal{U}(s)$ is* **t** *instead of* **u**.

Therefore, if the CP-theory of the above example contained an additional rule with body $\neg Throws(Suzy)$, this could be applied from state $s'_1$ onwards in the above branch, whereas a rule with body $\neg Broken$ would have to wait until $s'_3$.

With this additional condition, it now becomes possible for execution models to become stuck, in that sense that, in some leaf $l$, there remain some rules $r$ such that $\mathcal{I}(l) \models body(r)$, yet $r$ cannot happen because $body(r)$ is $\mathbf{u}$ in $\mathcal{U}(s)$. This can happen only when the CP-theory contains loops over negation. Such theories

are viewed as unsound, and no semantics is defined for them. An important class of sound theories are those which are stratified, but there also exist useful sound theories outside of this class (see Vennekens et al. (2009) for a discussion).

Again, an interesting special case is when all rules of the CP-theory are deterministic. In this case, the CP-theory syntactically coincides with a normal logic program, and all of its execution models end in a single leaf $l$, such that $\mathcal{U}(l)$ is the well-founded model of this program. If the CP-theory is sound, $\mathcal{U}(l) = \mathcal{I}(l)$ is the two-valued well-founded model and therefore also the unique stable model of the program. In this way, normal logic programs with a two-valued well-founded model are embedded in CP-logic. While the limitation to two-valued well-founded models may seem restrictive, in practice this is often mitigated by the fact predicates may be declared as exogenous, which has the same effect as "opening them up" with a loop over negation. Also in FO(ID), definitions whose well-founded model is not two-valued are considered inconsistent, so CP-logic is indeed a true generalization of FO(ID)'s inductive definition construct.

## 3 Negation in the head

A CP-theory represents a set of causal mechanisms, that are activated one after the other, and together construct the final state of the domain. Each such causal mechanism has the same kind of effect: for some set of atoms, it causes at most one of these atoms to deviate from their default value **f** to the deviant value **t**. If multiple causal mechanisms affect the same atom, the result is simple: there are no additive effects and the outcome is simply that the atom is **t** if and only if at least one mechanism causes it. If subsequent rules end up "causing" an effect that is already **t**, then this changes absolutely nothing.

It is to this setting that we now want to add negation-in-the-head. We will call such a negated literal in the head a *negative effect* literal. To be more precise, from now on, we allow rules of the form:

$$\forall \boldsymbol{x} \quad (L_1 : \alpha_1) \vee \cdots \vee (L_n : \alpha_n) \leftarrow \phi.$$

Here, $\phi$ is again a first-order logic formula with $\boldsymbol{x}$ as free variables and the $\alpha_i \in [0, 1]$ are again such that $\Sigma\alpha_i \leq 1$. Each of the $L_i$ is now either a *positive effect literal* $A$ (i.e., an atom) or a *negative effect literal* $\neg A$.

While the goal of this extension is of course to be able to represent such phenomena as the locking of the gear wheel described in the introduction, let us first take a step back and consider, in the abstract, which possible meanings this construct could reasonably have. Clearly, if for some atom $A$ only positive effect literals are caused, the atom should end up being true, just as it always has. Similarly, if only negative effect literals $\neg A$ are caused, the atom $A$ should be false. However, this does not even depend on the negative effect literals being present: because false is the default value in CP-logic, an atom will already be false whenever there are no positive effect literals for it, even if there are no negative effect literals either.

The only question, therefore, is what should happen if, for some $A$, both a positive and a negative effect literal are caused. One alternative could be that the result would somehow depend on the relative strength of the negative and positive effects, e.g., whether the power of aspirin to prevent a fever is "stronger" than the power of flu to cause it. However, such a semantics would be a considerable departure from the original version of CP-logic, in which cumulative effects are strictly ignored. In other words, CP-logic currently makes no distinction whatsoever between a headache that is simultaneously caused by five different conditions and a headache that has just a single cause. This design decision was made to avoid a logic that, in addition to probabilities, would also need to keep track of the degree to which a property holds. A logic combining probabilities with such fuzzy truth degrees would, in our opinion, become quite complex and hard to understand.

In this paper, we want to preserve the relative simplicity of CP-logic, and we will therefore again choose not to work with degrees of truth. Therefore, only two options remain: when both effect literals $A$ and $\neg A$ are caused, the end result must be that $A$ is either true of false. This basically means that, in the presence of both kinds of effect literals, we will have to choose to ignore one kind. It is obvious what this choice should be: the negative effect literals already have no impact on the semantics when there are only positive effect literals or when there are no positive effect literals, so if they would also have no impact when positive and negative effect literals are both present, then they would have never have any impact at all and we would have introduced a completely superfluous language construct. Therefore, the only reasonable choice is to give negative effect literals precedence over positive ones, that is, an atom $A$ will be true if and only if it is caused at least once and no negative effect literal $\neg A$ is caused.

This can be formally defined by a minor change to the existing semantics of CP-logic. Recall that, in the current semantics, each node $s$ of an execution model has an associated interpretation $\mathcal{I}(s)$, representing the current state of the world, and an associated three-valued interpretation $\mathcal{U}(s)$, representing an overestimate of all that could still be caused in $s$. We now add to this a third set, namely a set of atoms $\mathcal{N}(s)$, containing all atoms for which a negative effect literal has already been caused. The sets $\mathcal{I}(s)$ and $\mathcal{N}(s)$ evolve throughout an execution model as follows:

- In the root of the tree, $\mathcal{I}(s) = \mathcal{N}(s) = \{\}$
- When a *negative* effect literal $\neg A$ is caused in a node $s$, the execution model adds a child $s'$ to $s$ such that:
  - $\mathcal{N}(s') = \mathcal{N}(s) \cup \{A\}$;
  - $\mathcal{I}(s') = \mathcal{I}(s) \setminus \{A\}$.
- When a *positive* effect literal $A$ is caused in a node $s$, the execution model adds a child $s'$ to $s$ such that:
  - $\mathcal{N}(s') = \mathcal{N}(s)$;
  - if $A \in \mathcal{N}(s)$, then $\mathcal{I}(s') = \mathcal{I}(s)$, else $\mathcal{I}(s') = \mathcal{I}(s) \cup \{A\}$.

Note that, throughout the execution model, we maintain the property that $\mathcal{N}(s) \cap \mathcal{I}(s) = \{\}$.

The overestimate $\mathcal{U}(s)$ is again constructed as the limit of a sequence of three-valued interpretations $\nu_i$. To go from such a $\nu_i$ to $\nu_{i+1}$, we make $\nu_{i+1}(A) = \mathbf{u}$ for all atoms $A$ satisfying both of the following conditions:

- as before, $\nu_i(A) = \mathbf{f}$ and the positive effect literal $A$ appears in the head of a rule $r \in \mathcal{R}(s)$ with $\nu_i(body(r)) \neq \mathbf{f}$;
- but now also $A \notin \mathcal{N}(s)$.

In this way, $\mathcal{U}(s)$ always assigns $\mathbf{t}$ to all atoms in $\mathcal{I}(s)$ and $\mathbf{f}$ to all those in $\mathcal{N}(s)$.

## 4 Encoding interventions

One of the interesting uses of negation-in-the-head is related to the concept of interventions, introduced by Pearl (2000). Let us briefly recall this notion. Pearl works in the context of *structural models*. Such a model is built from a number of random variables. For simplicity, we only consider boolean variables, i.e., atoms. These are again divided into exogenous and endogenous atoms. A structural model now consists of one equation $X := \varphi$ for each endogenous atom $X$, which defines that $X$ is true if and only if the boolean formula $\varphi$ holds. This set of equations should be acyclic (i.e., if we order the variables by defining that $X < Y$ if $X$ appears in the equation defining $Y$, then this $<$ should be a strict order), in order to ensure that an assignment of values to the exogenous atoms induces a unique assignment of values to the endogenous ones.

A crucial property of causal models is that they can not only be used to predicts the normal behaviour of a system, but also to predict what would happen if outside factors unexpectedly intervene with its normal operation. For instance, consider the following simple model of which students must repeat a class:

$$Fail := \neg Smart \wedge \neg Effort. \qquad Repeat := Fail \wedge Required.$$

Under the normal operation of this "system", only students who are not smart can fail classes and be forced to repeat them. Suppose now that we catch a student cheating on an assignment and decide to fail him for the class. This action was not foreseen by the causal model, so it does not follow from the normal behaviour. In particular, failing the student may cause him to have to repeat the class, but if the student is actually smart, then failing him will not make him stupid. Pearl shows that we can model our action of failing the student by means of an *intervention*, denoted $do(Fail = \mathbf{t})$. This is a simple syntactic transformation, which removes and replaces the original equation for $Fail$:

$$Fail := \mathbf{t}. \qquad Repeat := Fail \wedge Required.$$

According to this updates set of equations, the student fails and may have to repeat the class, but he has not been made less smart.

In the context of CP-logic, let us consider the following simple medical theory:

$$(HighBloodPressure : 0.6) \leftarrow BadLifeStyle. \tag{14}$$

$$(HighBloodPressure : 0.9) \leftarrow Genetics. \tag{15}$$

$$(Fatigue : 0.3) \leftarrow HighBloodPressure. \tag{16}$$

Here, $BadLifeStyle$ and $Genetics$ are two exogenous predicates, which are both possible causes for $HighBloodPressure$. Suppose now that we observe a patient who suffers from $Fatigue$. Given our limited theory, this patient must be suffering from $HighBloodPressure$, caused by at least one of its two possible causes.

Now, suppose that a doctor is wondering whether it is a good idea to prescribe this patient some pills that cure high blood pressure. Again, the proper way to answer such a question is by means of an *intervention*, that first prevents the causal mechanisms that normally determine someone's blood pressure and then substitutes a new "mechanism" that just makes $HighBloodPressure$ false. This can be achieved by simply removing the two rules (14) and (15) from the theory. This is an instance of a general method, developed by Vennekens et al. (2010), of performing Pearl-style interventions in CP-logic. The result is that probability of $Fatigue$ drops to zero, i.e., $P(Fatigue \mid do(\neg HighBloodPressure)) = 0$.

In this way, we can evaluate the effect of prescribing the pills *without* actually having these pills in our model. This is a substantial difference to the way in which reasoning about actions is typically done in the field of knowledge representation, where formalisms such as situation or event calculus require an explicit enumeration of all available actions and their effects. Using an intervention, by contrast, we can envisage the effects of actions that we never even considered when writing our model.

Eventually, however, we may want to transform the above *descriptive* theory into a *prescriptive* one that tells doctors how to best treat a patient, given his or her symptoms. In this case, we would need rules such as this:

$$BPMedicine \leftarrow Fatigue. \tag{17}$$

Obviously, this requires us to introduce the action $BPMedicine$ of prescribing the medicine, which previously was implicit in our intervention, as an explicit action in our vocabulary. Negation-in-the-head allows us to syntactically express the effect of this new action: $\neg HighBloodPressure \leftarrow BPMedicine$.

This transformation can be applied in general, as the following theorem shows.

**Theorem 1.** *Let $T$ be a CP-theory over a propositional vocabulary $\Sigma$. For an atom $A \in \Sigma$, let $T'$ be the theory $T \cup \{r\}$ with $r$ the rule $\neg A \leftarrow B$ and $B$ an exogenous atom not in $\Sigma$. For each interpretation $X$ for the exogenous atoms of $T'$, if $B \in X$, then $\pi^X_{T'} = \pi^X_{do(T,\neg A)}$ and if $B \notin X$, then $\pi^X_{T'} = \pi^X_T$.*

This theorem shows that negation-in-the-head allows CP-theories to "internalize" the intervention of *doing* $\neg A$. The result is a theory $T'$ in which the intervention can be switched on or off by simply choosing the appropriate interpretation for the exogenous predicate that now explicitly represents this intervention. Once the intervention has been syntactically added to the theory in this way, additional rules such as (17) may of course be added to turn it from an exogenous to an endogenous property.

It is important to note that this is a fully modular and elaboration tolerant encoding of the intervention, i.e., the original CP-theory is left untouched and the rules that describe the effect of the intervention-turned-action are simply added to it. This is something that we can only achieve using negation-in-the-head.

## 5 Representing defaults

An interesting test case for logic programs has always been the representation of defaults. The typical example concerns the default $\delta = \frac{Bird(x) : Flies(x)}{Flies(x)}$ together with the background knowledge: $\forall x\ Penguin(x) \Rightarrow \neg Flies(x)$. In an extended logic program, the two kinds of negation can be exploited to represent the default in an elegant way:

$$Flies(x) \leftarrow Bird(x) \wedge \texttt{not}\ \neg Flies(x). \qquad \neg Flies(x) \leftarrow Penguin(x).$$

In a normal logic program or deterministic CP-theory, defaults are typically represented using an *abnormality* predicate.

$$Flies(x) \leftarrow Bird(x) \wedge \neg Ab_\delta(x). \qquad Ab_\delta(x) \leftarrow Penguin(x).$$

Using CP-logic's new negation-in-the-head, the abnormality predicate can be omitted.

$$Flies(x) \leftarrow Bird(x). \tag{18}$$

$$\neg Flies(x) \leftarrow Penguin(x). \tag{19}$$

However, we do now lose the ability to distinguish between defeasible and non-defeasible rules, since negative effect literals can always be added to block any effect. In fact, this is necessary because of our desire to use negation-in-the-head to syntactically represent interventions (Section 4). It is after all a key property of Pearl's interventions that any causal relation in the model should, in principle, be open to intervention.

Even though, as this section shows, it is possible to use CP-logic to represent certain defaults, it is important to remember that it is not intended as a default logic. In particular, rule (18) should not actually be read as saying that birds normally fly. Instead, it says that, for each $x$, $x$ being a bird causes it to be able to fly. Similarly, rule (19) says that being a penguin is a cause for being unable to fly. Note also that this is not a generally applicable methodology for representing defaults. For instance, if we wanted to state that penguins with jetpacks are an exception to rule (19), we would still have to introduce an abnormality predicate.

## 6 Probabilities and defaults

An interesting consequence of adding negation-in-the-head to CP-logic is that we can combine the encoding of defaults as in the previous section with uncertainty. For instance, let us suppose that there is, in general, a 5% change with which being a bird *does not* cause one to be able to fly. This may be the result, for instance, of a birth defect or some accident. This could be represented as follows:

$$(Flies(x) : 0.95) \leftarrow Bird(x). \tag{20}$$

$$\neg Flies(x) \leftarrow Penguin(x). \tag{21}$$

The first rule describes the normal situation for birds, whereas the second rule still serves to give an exception to the general rule. Note that, even for penguins, the causal mechanism underlying the first rule still happens, i.e., the rule is still fired, but it just fails to produce the outcomes of flying. Intuitively, we can think of this as the penguins still being born and being raised by their parents—i.e., they go through the same process of growing up that any bird goes through. It is just that, whereas this process causes the ability to fly for 95% of the normal birds, it never has this outcome for penguins. Of course, since learning to fly is actually the *only* possible effect of the first rule, the fact that this rule is still fired for penguins has no effect on anything.

The following example shows that this is not always the case.

$$(Wound(x) : 0.7) \vee (HoleInWall : 0.3) \leftarrow Shoot(x). \tag{22}$$

$$\neg Wound(x) \leftarrow Superhero(x). \tag{23}$$

Here, this first rule states that shooting a gun at someone might produce two possible effects: either the person ends up being wounded or the shot misses and causes instead a hole in the wall. The second rule adds an exception: if $x$ happens to be a superhero, then $x$ cannot be wounded. So, firing a gun at a superhero never causes $Wound(x)$, but with probability 0.3 still causes a hole in the wall.

This example also reveals a further way in which CP-logic is at heart a *causal* logic and not a logic of defaults. While we have so far been getting away with reading a rule such as (23) as expressing an exception to a default, this is not what it actually says: what this rule states is that being a superhero causes one to become "unwoundable". This does not only apply to wounds that would be caused by rule (22), but to all wounds. Therefore, if the CP-theory were to contain other causes for wounds, such as $(Wound(x) : 0.9) \leftarrow FallFromBuilding(x)$, then superheroes are automatically also protected against these.

## 7 Implementation

To implement the feature of negation-in-the-head, a simple transformation to regular CP-logic may be used. This transformation is based on the way in which Denecker and Ternovska (2007) encode causal ramifications in their inductive definition modelling of the situation calculus.

For a CP-theory $T$ in vocabulary $\Sigma$, let $\Sigma_\neg$ consist of all atoms $A$ for which a negative effect literal $\neg A$ appears in $T$. For each atom $A \in \Sigma_\neg$, we introduce two new atoms, $C_A$ and $C_{\neg A}$. Intuitively, $C_A$ means that there is a cause for $A$, and $C_{\neg A}$ means that there is a cause for $\neg A$. Let $\tau_A$ be the following transformation:

- Replace all positive effect literals $A$ in the heads of rules by $C_A$
- Replace all negative effect literals $\neg A$ in the heads of rules by $C_{\neg A}$
- Add this rule: $A \leftarrow C_A \wedge \neg C_{\neg A}$

Let $\tau_\neg(T)$ denote the result of applying to $T$, in any order, all the transformations $\tau_A$ for which $A \in \Sigma_\neg$. It is clear that $\tau_\neg(T)$ is a regular CP-theory, i.e., one

without negation-in-the-head. As the following theorem shows, this reduction preserves the semantics of the theory.

**Theorem 2.** *For each interpretation $X$ for the exogenous predicates, the projection of $\pi^X_{\tau_\neg(T)}$ onto the original vocabulary $\Sigma$ of $T$ is equal to $\pi^X_T$.*

When comparing the transformed theory $\pi_{\tau_\neg}(T)$ to the original theory $T$, we see that the main benefit of having negation-in-the-head lies in its *elaboration tolerance*: there is no need to know before-hand for which atoms we later might wish to add negative effect literals, since we can always add these later, without having to change to original rules. Both in the example of syntactically representing an intervention (Section 4) and that of representing exceptions to defaults (Section 5), this feature may be useful.

## 8 Conclusion

This paper is part of a long-term research project which aims to develop a Tarskian alternative to ASP: instead of relying on ASP's original epistemic intuitions, our goal is to have a language in which every expression can be interpreted as an objective statement about the real world. The first motivation for this is *simplicity*: many problems that are solved using present-day ASP systems and the GDT-methodology do not have an inherent epistemic component, so it would just be simpler if we could understand such programs in terms of what they say about the real world directly, instead of having to make a detour through the beliefs of some (irrelevant) rational agent. A second motivation is the *unity of science*: a huge effort has gone into both theoretical and practical research on classical logic. Its roots in Non-monotonic Reasoning have made ASP an antithesis to the classical approach, in which the desire to express objective knowledge is abandoned in favor of epistemic knowledge. Even though applications of ASP-solvers and SAT-solvers are often quite similar in practice, the "official" reading of ASP programs and classical theories is therefore radically different. The second goal is to bridge this gap.

An important part of this research project was the development of the language FO(ID), which showed how normal logic programs could be interpreted as *inductive definitions* and added in a meaningful way to classical logic. An extension of this work was the development of the language CP-logic, which allows non-deterministic and probabilistic causal processes to be expressed. In this paper, we have investigated the useful ASP feature of negation-in-the-head. We presented a meaningful interpretation of this feature in the context of CP-logic and discussed possibles uses of it. Finally, we also showed a simple transformation that reduces it to regular CP-logic.

## References

C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming* 9(1):57-144, 2008.

M. Denecker and E. Ternovska. Inductive situation calculus. *Artificial Intelligence*, 171(5-6):332–360, 2007.

M. Denecker and J. Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In *LPNMR*, volume 4483 of *LNCS*, pages 84–96. Springer, 2007.

M. Denecker, J. Vennekens, H. Vlaeminck, J. Wittocx, and M. Bruynooghe. Answer set programming's contributions to classical logic. An analysis of ASP methodology. In *MG-65: Symposium on Constructive Mathematics in Computer Science*, 2010.

M. Denecker, Y. Lierler, M. Truszczynski, and J. Vennekens. A tarskian informal semantics for asp. In *Technical Communications of the 28th International Conference on Logic Programming*, 2012.

M. Gelfond. Strong introspection. In *AAAI*, pages 386–391, 1991.

M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

N. Hall. Structural equations and causation. *Philosophical Studies*, 132(1): 109–136, 2007.

V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.

C. Sakama and K. Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13 (1):145–172, 1994.

J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009.

J. Vennekens, M. Denecker, and M. Bruynooghe. Embracing events in causal modelling: Interventions and counterfactuals in CP-logic. In *JELIA*, pages 313–325, 2010.

# Author Index