WORKSHOP PROCEEDINGS

ICLP 2009 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2009)

Edited by Wolfgang Faber and Joohyung Lee

July 14, 2009

Preface

These are working notes of the workshop on *Answer Set Programming and Other Computing Paradigms (ASPOCP) 2009*, collocated with the *25th International Conference on Logic Programming (ICLP) 2009* in Pasadena, California, USA.

Since its introduction in the 1990s, 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 new method of computing answer sets using SAT solvers and techniques adapted from SAT. While so far this has been the most studied relationship, identifying links between ASP and other computing paradigms, such as constraint satisfaction, quantified Boolean formulas (QBF), first-order logic (FOL), or databases, to name just a few, is the subject of active research.

The contributions brought about by these studies are manifold: 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, and FOL theorem provers. New and improved languages are proposed, inspired by language constructs found in related paradigms. In a somewhat orthogonal way, languages or tasks in other research areas are reduced to ASP, one of the main benefits being that a computational engine is thereby automatically provided. Furthermore, language and solver integration is facilitated, allowing for multi-paradigm problem-solving; currently the integration of ASP with description logics (in the realm of the Semantic Web) and constraint satisfaction are the main focus of this type of activity.

This workshop aims at facilitating the discussion about crossing the boundaries of current ASP techniques, in combination with or inspired by other computing paradigms. It is the second workshop of its kind after ASPOCP 2008, which was also collocated with ICLP in December 2008 in Udine, Italy. Despite the relatively short period between the first workshop and this second edition, and despite the submission phase having been in a very busy period of the year, we are happy to have received 6 submissions, of which 5 were accepted for presentation. We thank the contributors for their efforts to provide material of high quality, we thank the program committee members and reviewers for their valuable help to guarantee and improve the quality of the workshop, and last but not least the ICLP officials for making this workshop possible and for their smooth cooperation.

Wishing you all an informative and enjoyable workshop,

Wolfgang Faber, University of Calabria, Italy
Joohyung Lee, Arizona State University, USA

Programme Chairs

Wolfgang Faber, University of Calabria, Italy
Joohyung Lee, Arizona State University, USA

Programme Committee

Chitta Baral, Arizona State University, USA
Gerhard Brewka, University of Leipzig, Germany
Pedro Cabalar, University of A Coruña, Spain
Nicola Leone, University of Calabria, Italy
Vladimir Lifschitz, University of Texas at Austin, USA
Fangzhen Lin, Hong Kong University of Science and Technology, China
Thomas Lukasiewicz, University of Oxford, UK
Ilkka Niemelä, Helsinki University of Technology, Finland
Torsten Schaub, University of Potsdam, Germany
Mirosław Truszczyński, University of Kentucky, USA
Dirk Vermeir, Vrije Universiteit Brussel, Belgium
Stefan Woltran, Vienna University of Technology, Austria
Jia-Huai You, University of Alberta, Canada
Yan Zhang, University of Western Sydney, Australia
Yuanlin Zhang, Texas Tech University, USA

Additional Reviewers

Susanna Cozza
Jianmin Ji
Guohua Liu
Marco Manna
Max Ostrowski

Contents

A Module-Based Framework for Multi-Language Constraint Modeling*

Matti Järvisalo, Emilia Oikarinen, Tomi Janhunen, and Ilkka Niemelä

Helsinki University of Technology TKK
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
matti.jarvisalo@tkk.fi,emilia.oikarinen@tkk.fi,
tomi.janhunen@tkk.fi,ilkka.niemela@tkk.fi

Abstract. We develop a module-based framework for constraint modeling where it is possible to combine different constraint modeling languages and exploit their strengths in a flexible way. In the framework a constraint model consists of modules with clear input/output interfaces. When combining modules, apart from the interface, a module is a black box whose internals are invisible to the outside world. Inside a module a chosen constraint language (approaches such as CP, ASP, SAT, and MIP) can be used. This leads to a clear modular semantics where the overall semantics of the whole constraint model is obtained from the semantics of individual modules. The framework supports multi-language modeling without the need to develop a complicated joint semantics and enables the use of alternative semantical underpinnings such as default negation and classical negation in the same model. Furthermore, computational aspects of the framework are considered and, in particular, possibilities of benefiting from the known module structure in solving constraint models are studied.

1 Introduction

There are several constraint-based approaches to solving combinatorial search and optimization problems: constraint programming (CP), answer set programming (ASP), mixed integer programming (MIP), linear programming (LP), Boolean satisfiability checking (SAT) and its extension to satisfiability modulo theories (SMT). Each has its particular strengths: for example, CP systems support global constraints, ASP recursive definitions and default negation, LP constraints on real-valued variables, and SAT efficient solver technology. In larger applications it is often necessary to exploit the strengths of several languages and to reuse and combine available components. For example, in scheduling problems involving a large amount data and constraints, multi-language modeling can be very useful (as also exemplified in this paper in Sect. 5).

In this work we develop a module-based framework for modeling complex problems with constraints using a combination of different modeling languages. Rather than taking one language as a basis and extending it, we develop a framework for multi-language modeling where different languages are treated on equal terms. The starting

* This work is financially supported by Academy of Finland under the project *Methods for Constructing and Solving Large Constraint Models* (grant #122399).

point is to use modules with clear input/output (I/O) interfaces. When combining modules, apart from the interface, a module is a black box whose internals are invisible to the outside world. Inside a module a chosen constraint language (for example, CP, ASP, MIP) with its normal semantics can be used. In this way a clear modular semantics is obtained: the overall semantics of the whole constraint model (consisting of modules) is obtained by "composing" the semantics of individual modules.

We see substantial advantages of this approach for modeling. The clean module interfaces enable support for multi-language modeling without the need to develop a complicated joint semantics capturing arbitrary combinations of special constraints available in different languages. It is also possible to use alternative semantical underpinnings such as default negation and classical negation in the same model. The module-based approach brings the benefits of modular programming to developing constraints models and enables to create libraries to enhance module reuse. It also improves elaboration tolerance and facilitates maintaining and updating a constraint model. Moreover, extending the approach with further languages is conceptually straightforward.

Computational aspects of the framework are also promising. Module interfaces and separation of inputs and outputs can be exploited in decision methods, for example, with more top-down solution techniques where the overall output of the constraint model can be used to identify the relevant parts of the model. The module-based approach allows optimizing the computational efficiency of a model in a structured way: a module can be replaced by another (more optimized) version without altering the solutions of the model as long as the I/O relation of the module is not changed. Similarly, the framework supports modular testing, validation, and debugging of constraint models.

This module-based framework for multi-language modeling seems to be a novel approach. Several approaches to adding modularity to ASP languages [1–5] have been proposed. However, in these approaches modular multi-language modeling is not directly supported although the combination of propositional ASP and SAT modules is studied in [5]. A large number of extended modeling languages have also been previously proposed. On one hand, ASP languages have been extended with constraints or other externally defined relations (see [6–11] for examples). On the other hand, Prolog systems have been extended with ASP features [12–15]. Extended modeling languages have been developed also for constraint programming, including ESRA [16], ESSENCE [17], and Zinc [18]. However, none of the approaches supports modular multi-language modeling where different languages are treated on equal terms. Instead, they can all be seen as extensions of a given basic language with features from other languages.

The rest of this paper is organized as follows. As preliminaries, we first give a generic definition of a constraint and related notation (Sect. 2). Then constraint modules, the basic building blocks of module systems, are introduced (Sect. 3). The language of module systems, based on composing constraint modules, is discussed in Sect. 4. Then, in Sect. 5 we discuss how the framework can be instantiated in practice: A larger application is considered in order to illustrate the issues arising in using a multi-language modeling approach, and the required language interface for constructing a multi-language module system is sketched. Before conclusions (Sect. 7), com-

putational aspects, and especially, possibilities of benefiting from the explicit modular constraint model description when solving such a model are highlighted (Sect. 6).

2 Constraints

In this section we introduce necessary concepts and notation related to the generic concept of constraints applied in this work. These serve as basic building blocks for constraint modules which are then combined to form complex constraint models.

Let $\mathcal{X}$ be a set of variables. For each variable $x \in \mathcal{X}$, we associate a set of *values* $D(x)$, called the *domain* of x. Given a set $X \subseteq \mathcal{X}$ of variables, an assignment over X is a function

$$\tau : X \to \bigcup_{x \in X} D(x),$$

which maps variables in X to values in their domains. A *constraint* $\mathcal{C}$ over a set of variables X is characterized by a set $\mathsf{Solutions}(\mathcal{C})$ of assignments over X, called the *satisfying assignments* of $\mathcal{C}$. We denote by $\mathsf{Vars}(\mathcal{C})$ the set X of variables.

It is important to notice that, since the satisfying assignments solely characterize the constraint, this generic way of describing constraints does not specify how a constraint should be implemented, i.e. the modeling language and semantics used for realizing the constraint declaratively remain unspecified.

Example 1. Let $\mathcal{C}$ be a constraint over a set of Boolean variables $\{a, b\}$, i.e., $D(a) = D(b) = \{\mathbf{t}, \mathbf{f}\}$, characterized by $\mathsf{Solutions}(\mathcal{C}) = \{\tau_1, \tau_2\}$, where $\tau_1 = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}\}$ and $\tau_2 = \{a \mapsto \mathbf{f}, b \mapsto \mathbf{t}\}$. Now, $\mathcal{C}$ can be implemented, for example, as a *normal logic program* $\{a \leftarrow {\sim}b.\ b \leftarrow {\sim}a\}$ or as a *disjunctive logic program* $\{a \vee b \leftarrow\}$ in ASP, or as a *conjunctive normal form (CNF) formula* $\{a \vee \neg b, \neg a \vee b\}$ in SAT.

Given an assignment τ and a set of variables X, the *projection* $\pi_X(\tau)$ of τ on X is the assignment that maps each variable $x \in X$ for which $\tau(x)$ is defined to $\tau(x)$. For instance, the projection $\pi_{\{a\}}(\tau_1)$ for τ_1 from Example 1 is the assignment $\pi_{\{a\}}(\tau_1) = \{a \mapsto \mathbf{t}\}$ over the set $\{a\}$.

Given a constraint $\mathcal{C}$, and an assignment τ over a set X of variables, the *restriction* $\mathcal{C}[\tau]$ of $\mathcal{C}$ to τ is characterized by

$$\mathsf{Solutions}(\mathcal{C}[\tau]) = \{\tau' \in \mathsf{Solutions}(\mathcal{C}) \mid \pi_{\mathsf{Vars}(\mathcal{C}) \cap X}(\tau') = \pi_{\mathsf{Vars}(\mathcal{C}) \cap X}(\tau)\}.$$

For instance, let $\tau_3 = \{b \mapsto \mathbf{f}\}$ be an assignment over $\{b\}$. Now, the restriction $\mathcal{C}[\tau_3]$ of $\mathcal{C}$ from Example 1 is a constraint characterized by $\{\tau_1\} \subseteq \mathsf{Solutions}(\mathcal{C})$, i.e., $\mathsf{Solutions}(\mathcal{C}[\tau_3]) = \{\tau_1\}$.

Given two constraints $\mathcal{C}$ and $\mathcal{C}'$, an assignment τ over $\mathsf{Vars}(\mathcal{C})$ is *compatible* with an assignment τ' over $\mathsf{Vars}(\mathcal{C}')$ if $\pi_{\mathsf{Vars}(\mathcal{C}) \cap \mathsf{Vars}(\mathcal{C}')}(\tau) = \pi_{\mathsf{Vars}(\mathcal{C}) \cap \mathsf{Vars}(\mathcal{C}')}(\tau')$. The union $\tau \cup \tau'$ of two compatible assignments, τ and τ' over X and X', respectively, is the assignment over $X \cup X'$ mapping each $x \in X$ to $\tau(x)$ and each $x \in X' \setminus X$ to $\tau'(x)$.

Example 2. Let $\mathcal{C}'$ be a constraint over a set of Boolean variables $\{b, c\}$ characterized by $\mathsf{Solutions}(\mathcal{C}') = \{\tau'\}$ such that $\tau' = \{b \mapsto \mathbf{f}, c \mapsto \mathbf{f}\}$. Consider $\mathcal{C}$ from Example 1.

The assignment τ_1 is compatible with τ', because $\{a, b\} \cap \{b, c\} = \{b\}$ and $\tau_1(b) = \mathbf{f} = \tau'(b)$. On the other hand, τ_2 is not compatible with τ', because $\tau_2(b) = \mathbf{t} \neq \tau'(b)$. The union $\tau_1 \cup \tau' = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}, c \mapsto \mathbf{f}\}$ is an assignment over the set $\{a, b, c\}$.

3 Constraint Modules

The view to constructing complex constraint models proposed in this work is based on expressing such models as *module systems*. Module systems are built from *constraint modules* which are combined together in a controlled fashion. In this section we introduce the generic concept of a constraint module. Constraint modules are based on a chosen constraint, with the addition of an explicit I/O interface. Our definition for a constraint module is generic in the sense that it does not insist on a specific implementation of the constraint on the declarative level. The aim here is to allow implementing the constraint using different declarative languages, offering the implementer of a module the possibility to choose the constraint language and the semantics.

Definition 1. *A constraint module $\mathcal{M}$ is a triple $\langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$, where*

- *$\mathcal{C}$ is a constraint; and*
- *$\mathcal{I}$ and $\mathcal{O}$ define the* I/O interface *of $\mathcal{M}$:*
  - *$\mathcal{I} \subseteq \mathsf{Vars}(\mathcal{C})$ is the input specification of $\mathcal{M}$,*
  - *$\mathcal{O} \subseteq \mathsf{Vars}(\mathcal{C})$ is the output specification of $\mathcal{M}$, and*
  - *$\mathcal{I} \cap \mathcal{O} = \emptyset$.*

A module $\mathcal{M}$ is thus a constraint with a fixed I/O interface. In analogy to the characterization of a constraint, a module $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$ is characterized by a set $\mathsf{Solutions}(\mathcal{M})$ of assignments over $\mathcal{I} \cup \mathcal{O}$ called the *satisfying assignments of the module.* Given a constraint module $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$ and an assignment τ_I over $\mathcal{I}$, the set of *consistent outputs* of $\mathcal{M}$ w.r.t. τ_I is

$$\mathsf{SolutionOut}(\mathcal{M}, \tau_I) := \{\pi_{\mathcal{O}}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{C}[\tau_I])\}.$$

The satisfying assignments of a module are obtained by considering all possible input assignments.

Definition 2. *Given a constraint module $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$, the set $\mathsf{Solutions}(\mathcal{M})$ of satisfying assignments of $\mathcal{M}$ is the union of the sets $\{\tau_O \cup \tau_I \mid \tau_O \in \mathsf{SolutionOut}(\mathcal{M}, \tau_I)\}$ for all assignments τ_I over $\mathcal{I}$.*

Those variables in $\mathsf{Vars}(\mathcal{C})$ which are not in $\mathcal{I} \cup \mathcal{O}$ are *local to* $\mathcal{M}$; the assignments in $\mathsf{Solutions}(\mathcal{M})$ do not assign values to them. Notice that the possibility of local variables enables *encapsulation* and *information hiding*. A module offers through its I/O interface to the user a black-box implementation of a specific constraint. The idea behind this abstract way of defining a module is that, looking from the outside of a module when using the module as a part of a constraint model, the user is interested in the input-output relationship, i.e., the functionality of the module. This can be highlighted by making explicit the conditions under which two modules are considered equivalent.

Definition 3. *Two constraint modules,* $\mathcal{M}_1 = \langle \mathcal{C}_1, \mathcal{I}_1, \mathcal{O}_1 \rangle$ *and* $\mathcal{M}_2 = \langle \mathcal{C}_2, \mathcal{I}_2, \mathcal{O}_2 \rangle$*, are* equivalent*, denoted by* $\mathcal{M}_1 \equiv \mathcal{M}_2$*, if and only if* $\mathcal{I}_1 = \mathcal{I}_2$*,* $\mathcal{O}_1 = \mathcal{O}_2$*, and* $\mathsf{Solutions}(\mathcal{M}_1) = \mathsf{Solutions}(\mathcal{M}_2)$.

Example 3. Consider $\mathcal{M} = \langle \mathcal{C}, \{a\}, \{b\} \rangle$, where a and b are Boolean variables, and let $\mathsf{Solutions}(\mathcal{M}) = \{\tau_1, \tau_2\}$ where $\tau_1 = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}\}$ and $\tau_2 = \{a \mapsto \mathbf{f}, b \mapsto \mathbf{t}\}$. Since τ_1 and τ_2 are the same as in Example 1, $\mathcal{M}$ can be implemented using any of the implementations of the constraint described in Example 1.

Moreover, the set of variables used in implementing $\mathcal{C}$ is not limited to $\{a, b\}$. For instance, a *logic program module* [4] $\langle P, I, O \rangle = \langle \{c \leftarrow \sim a.\ b \leftarrow c\}, \{a\}, \{b\} \rangle$ is an implementation of $\mathcal{C}$ such that $\mathsf{Solutions}(\mathcal{C}) = \{\tau_3, \tau_4\}$ where $\tau_3 = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}, c \mapsto \mathbf{f}\}$ and $\tau_4 = \{a \mapsto \mathbf{f}, b \mapsto \mathbf{t}, c \mapsto \mathbf{t}\}$.[1] Now, there are two possible assignments over $\{a\}$. If $\tau_I = \{a \mapsto \mathbf{t}\}$ we obtain $\mathsf{SolutionOut}(\mathcal{M}, \tau_I) = \{\pi_{\{b\}}(\tau_3)\}$ since $\mathsf{Solutions}(\mathcal{C}[\tau_I]) = \{\tau_3\}$ as $\tau_3(a) = \tau_I(a) = \mathbf{t}$. For the other possible input assignment $\tau'_I = \{a \mapsto \mathbf{f}\}$, we obtain $\mathsf{SolutionOut}(\mathcal{M}, \tau'_I) = \{\pi_{\{b\}}(\tau_4)\}$. Finally, note that $\tau_I \cup \pi_{\{b\}}(\tau_3) = \tau_1$ and $\tau'_I \cup \pi_{\{b\}}(\tau_4) = \tau_2$. Thus, $\mathsf{Solutions}(\mathcal{M}) = \{\tau_1, \tau_2\}$.

4 Module Systems

In this section we discuss how larger module systems are built from individual constraint modules. The idea is that module systems are constructed by connecting smaller module systems through the I/O interfaces offered by such systems. In other words, similarly as constraint modules, a module system has an I/O interface, and constraint modules are seen as primitive module systems. We will start by introducing a formal language for expressing such systems and then introduce the semantics for module systems which are *well-formed*.

Definition 4 (The language of module systems).

1. *All constraint modules are module systems.*
2. *If* $\mathcal{M}$ *is a module system and* X *is a set of variables, then* $\pi_X(\mathcal{M})$ *is a module system.*
3. *If* $\mathcal{M}$ *and* $\mathcal{M}'$ *are module systems, then* $(\mathcal{M} \triangleright \mathcal{M}')$ *is a module system.*

Notice that Definition 4 is purely syntactical. Our next goal is to define the semantics for more complex module systems as we have already defined the semantics of individual constraint modules. This is achieved by formalizing the semantics of operators π_X and $\triangleright$; intuitively, π_X offers a way of filtering the output of a module system, whereas $\triangleright$ is used for composing two module systems into one.

Let us denote by $\mathsf{Input}(\mathcal{M})$ and $\mathsf{Output}(\mathcal{M})$ the input and output of a module system $\mathcal{M}$, respectively. We start by defining the conditions under which two module systems are *composable* and *independent*.

[1] Notice that unlike other formalisms mentioned so far, the logic program modules in [4] already facilitate I/O interfaces, and their semantics differs from the standard stable model semantics since input variables have a classical interpretation.

Definition 5 (Composable and independent module systems). *Two module systems $\mathcal{M}_1$ and $\mathcal{M}_2$ are* composable *if* $\mathsf{Output}(\mathcal{M}_1) \cap \mathsf{Output}(\mathcal{M}_2) = \emptyset$. *Module system $\mathcal{M}_1$ is* independent *from module system $\mathcal{M}_2$ if* $\mathsf{Input}(\mathcal{M}_1) \cap \mathsf{Output}(\mathcal{M}_2) = \emptyset$.

The composability property is used to ensure that if two module systems interfere with each others' output, they cannot be put together. The independence property allows us to ensure that two modules are not in cyclic dependency. Note that the independence of $\mathcal{M}_1$ from $\mathcal{M}_2$ does not imply that $\mathcal{M}_2$ is independent from $\mathcal{M}_1$.

When composing module systems, we have to take into account their dependencies.

Definition 6 (Module composition). *Given two composable module systems $\mathcal{M}_1$ and $\mathcal{M}_2$, their composition $\mathcal{M}_1 \triangleright \mathcal{M}_2$ is defined if and only if $\mathcal{M}_1$ is independent from $\mathcal{M}_2$. Now $\mathcal{M}_1 \triangleright \mathcal{M}_2$ is a module system that has the following I/O-interface:*

- $\mathsf{Input}(\mathcal{M}_1 \triangleright \mathcal{M}_2) = \mathsf{Input}(\mathcal{M}_1) \cup (\mathsf{Input}(\mathcal{M}_2) \setminus \mathsf{Output}(\mathcal{M}_1))$
- $\mathsf{Output}(\mathcal{M}_1 \triangleright \mathcal{M}_2) = \mathsf{Output}(\mathcal{M}_1) \cup \mathsf{Output}(\mathcal{M}_2)$

and

$$\mathsf{Solutions}(\mathcal{M}_1 \triangleright \mathcal{M}_2) = \{(\tau_1 \cup \tau_2) \mid \tau_1 \in \mathsf{Solutions}(\mathcal{M}_1), \tau_2 \in \mathsf{Solutions}(\mathcal{M}_2), \text{ and } \tau_2 \text{ is compatible with } \tau_1\}.$$

Notice that in the composition $\mathcal{M}_1 \triangleright \mathcal{M}_2$, module systems $\mathcal{M}_1$ and $\mathcal{M}_2$ interact though the variables in $\mathsf{Output}(\mathcal{M}_1) \cap \mathsf{Input}(\mathcal{M}_2)$.

Example 4. Let $\mathcal{M} = \langle \mathcal{C}, \{a\}, \{n\} \rangle$ and $\mathcal{M}' = \langle \mathcal{C}', \{n\}, \{m\} \rangle$ be constraint modules where a is a Boolean variable, $D(n) = D(m) = \{1, 2, 3\}$, $\mathsf{Solutions}(\mathcal{M}) = \{\tau_1\}$, and $\mathsf{Solutions}(\mathcal{M}') = \{\tau_2, \tau_3\}$ where $\tau_1 = \{a \mapsto \mathbf{f}, n \mapsto 3\}$, $\tau_2 = \{n \mapsto 1, m \mapsto 1\}$, and $\tau_3 = \{n \mapsto 3, m \mapsto 2\}$. Since $\mathcal{M}$ is independent from $\mathcal{M}'$, their composition $\mathcal{M} \triangleright \mathcal{M}'$ is defined. Notice that $n \in \mathsf{Output}(\mathcal{M}) \cap \mathsf{Input}(\mathcal{M}')$ provides the connection between $\mathcal{M}$ and $\mathcal{M}'$, i.e., $n \in \mathsf{Output}(\mathcal{M})$ is the input for $\mathcal{M}'$ because $\mathsf{Input}(\mathcal{M}') = \{n\}$. Furthermore, $\mathsf{Input}(\mathcal{M} \triangleright \mathcal{M}') = \{a\}$, $\mathsf{Output}(\mathcal{M} \triangleright \mathcal{M}') = \{n, m\}$, and $\mathsf{Solutions}(\mathcal{M} \triangleright \mathcal{M}') = \{\tau_1 \cup \tau_3\}$, because τ_1 is not compatible with τ_2 and τ_1 is compatible with τ_3.

As a special case, the *empty module* $\mathcal{E}$ is a constraint module such that $\mathsf{Input}(\mathcal{E}) = \mathsf{Output}(\mathcal{E}) = \emptyset$ and $\mathsf{Solutions}(\mathcal{E}) = \{\tau_e\}$, where τ_e is the *empty assignment*. Note that, given any module system $\mathcal{M}$, both $\mathcal{E} \triangleright \mathcal{M}$ and $\mathcal{M} \triangleright \mathcal{E}$ are defined, and $\mathcal{E} \triangleright \mathcal{M} \equiv \mathcal{M} \triangleright \mathcal{E} \equiv \mathcal{M}$.

Definition 7 (Projecting output of a module system). *Given a module system $\mathcal{M}$ and set of variables $\mathcal{O}$, the module system $\pi_{\mathcal{O}}(\mathcal{M})$ is defined if and only if $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$. Now $\pi_{\mathcal{O}}(\mathcal{M})$ is a module system that has the following I/O interface:* $\mathsf{Input}(\pi_{\mathcal{O}}(\mathcal{M})) = \mathsf{Input}(\mathcal{M})$, $\mathsf{Output}(\pi_{\mathcal{O}}(\mathcal{M})) = \mathcal{O}$, *and*

$$\mathsf{Solutions}(\pi_{\mathcal{O}}(\mathcal{M})) = \{\pi_{\mathcal{O} \cup \mathsf{Input}(\mathcal{M})}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{M})\}.$$

Example 5. Consider the module system $\mathcal{M} \triangleright \mathcal{M}'$ from Example 4 and assume that we are not interested in the values assigned to n. Thus, we consider the projection $\mathcal{M}_\pi = \pi_{\{m\}}(\mathcal{M} \triangleright \mathcal{M}')$. Now $\mathsf{Input}(\mathcal{M}_\pi) = \{a\}$, $\mathsf{Output}(\mathcal{M}_\pi) = \{m\}$, and $\mathsf{Solutions}(\mathcal{M}_\pi) = \{\tau_\pi\}$ where $\tau_\pi = \pi_{\{a,m\}}(\tau_1 \cup \tau_3) = \{a \mapsto \mathbf{f}, m \mapsto 2\}$.

We are interested in so called *well-formed* module systems that respect the conditions for applying $\triangleright$ (independence) and π_X (projection is focused on output).

Definition 8 (Well-formed module system). *A module system is* well-formed *if each composition and projection operation is defined in the sense of Definitions 6 and 7.*

Determining whether an arbitrary module system is well-formed consists of a syntactic check on the compositionality and compatibility of the I/O interfaces ($\triangleright$) and subset relation (π). From now on we use the term *module system* to refer to a well-formed module system. The graph formed by taking into account the input-output dependencies of parts of a module system is *directed* and *acyclic*, and is referred to as the *module dependency graph*. More precisely, the module dependency graph of a given module system $\mathcal{M}$ has the set of constraint modules appearing in $\mathcal{M}$ as the set of vertices. There is a edge from a constraint module $\mathcal{M}_1$ to module $\mathcal{M}_2$ if and only if at least one output variable of $\mathcal{M}_1$ is an input variable of $\mathcal{M}_2$.

By definition, the semantics of a well-formed module system is *compositional*: compatible solutions for individual parts form a solution for the whole system and a solution for the module system gives solutions for the individual parts.

Remark 1. Operators for $\triangleright$ and π_X provide flexible ways for building complex module systems. Additional operators useful in practice can be defined as combinations of these basic operators. For instance, by combining composition with projection we obtain $\mathcal{M}_1 \blacktriangleright \mathcal{M}_2$ defined as $\pi_{\mathsf{Output}(\mathcal{M}_2)}(\mathcal{M}_1 \triangleright \mathcal{M}_2)$. One could also be interested in a non-deterministic choice of solutions for $\mathcal{M}_1$ and $\mathcal{M}_2$ (denoted $\mathcal{M}_1 \cup \mathcal{M}_2$) or common solutions for $\mathcal{M}_1$ and $\mathcal{M}_2$ (denoted $\mathcal{M}_1 \cap \mathcal{M}_2$). In order to define $\mathcal{M}_1 \cup \mathcal{M}_2$ and $\mathcal{M}_1 \cap \mathcal{M}_2$, we cannot assume that $\mathcal{M}_1$ and $\mathcal{M}_2$ are composable. However, even these operators can be expressed in terms of composition and projection using an additional renaming scheme for variables.

5 Module Systems in Practice

We now outline how the framework for module systems developed in the previous section can be instantiated in practice. In Section 5.1 we present an example application from the university timetabling domain in order to point out issues arising in a multi-language modeling scenario. The example illustrates advantages of using the module-based framework in such applications. The modularity of the framework supports a structured modeling approach which enhances the development, maintenance, and updating of the constraint model. Moreover, the approach makes it possible to use multiple modeling languages in a clean modular way taking advantage of their best features. For example, ASP languages provide an intuitive way of expressing recursive definitions, defaults, and exceptions. On the other hand, CP languages offer a wide selection of *global constraints* for specific modeling purposes. In Section 5.2 we sketch the required language interface for constructing a multi-language module system.

5.1 Modular Representation for the Timetabling Domain

For illustrating multi-language modeling, we describe components involved in a modular constraint model for *university timetabling*, variants of which have previously been formalized, e.g., as a Boolean satisfiability, as a constraint satisfaction problem (CSP) [19], and as a logic program [20, Appendix].

Designing a feasible weekly schedule for events related to courses in a university curriculum is a challenging task. The problem is not just about allocation time and space resources; the interdependencies of courses and the respective events give rise to a rich body of constraints. When modeling the domain, one needs to express the mutual exclusion of events as regards, e.g., placing any two events in the same lecture hall at the same time. A straightforward representation of such a constraint with clauses or rules may require quadratic space. In contrast, a very concise encoding of this aspect of the domain can be obtained using global constraints such as *all-different* or *cumulative* typically supported by constraint programming systems which also offer powerful propagation techniques for such constraints. On the other hand, there are features which are cumbersome to describe in CP. For example, exceptions like the temporary unavailability of a particular lecture hall in a timetable are easy to represent with non-monotonic rules such as those used in ASP. Moreover, rules provide a flexible way of defining new relations on the basis of existing ones.

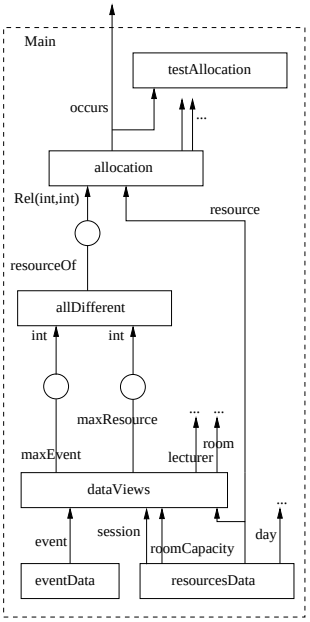


Fig. 1. Example of a Module System

The structure of a modular constraint model for the university timetabling domain is depicted in Fig. 1. The individual constraint modules involved in the model serve the following purposes. The two ASP modules at the bottom define relations specific to a particular problem instance. The first module, *eventData*, defines which events are involved in the problem. The second, *resourcesData*, formalizes the time and space resources available for scheduling. For the sake of simplicity, it is assumed that the five days of the week subject to scheduling are divided into three 4-hour sessions: morning, afternoon, and evening. An individual *resource* is conceptualized as a pair $\langle r, s \rangle$ where r is a room and s is a session. The ASP module on top of these two modules, *dataViews*, defines a number of subsidiary relations, such as ROOM(r) (available rooms) and LECTURER(l) (involved lecturers), on the basis of the basic relations provided by modules *eventData* and *resourcesData* which together determine the problem instance in question. In particular, the dimensions of a problem instance are determined dynamically. To this end, the relations MAXEVENT(n) and MAXRESOURCE(m) are supposed to hold (only) for the respective numbers of events n and resources m.

After suitable type conversions—represented by the circles in Fig. 1—these two parameters serve as input for the CP module *allDifferent* whose purpose is to assign different resources (represented by integers in the range $1 \ldots m$) to all events (represented by an array of integers indexed by $1 \ldots n$). Through such a conversion, a constraint library implementation of *allDifferent* which works only on integer-valued variables can be directly used. The resulting array of assignments of integers, RESOURCEOF, is then converted to a relation for events e and resources r and the ASP module *allocation* is used to restore the representation of resources as integers back to pairs of rooms and sessions. The outcome relation OCCURS(e, r, s) denotes the fact that an event e takes place in room r during session s. The topmost module *testAllocation* ensures that the given allocation of resources to events, i.e., the relation OCCURS(e, r, s) meets further criteria of interest. For instance, one could insist on the property that sessions related with a particular lecture hall are always reserved in a continuous manner, i.e., no gaps are allowed between reservations in the respective schedule.

5.2 Language Interface for Combining Constraint Modules

Referring to the theory developed in Sect. 3 and 4, we distinguish two types of module declarations. An individual constraint module is written in a particular constraint language accompanied by appropriate I/O interface specifications. The language of each constraint module is declared using an identifier "SAT", "ASP", "CP", etc. A module system is effectively a definition of the interconnections between submodules encapsulated by it. Since module systems are not confined to a particular constraint language the identifier "SYSTEM" is used. In addition, simple *type converters* are declared when needed, as outlined above.

In practice, a module system is not described as an expression (recall Definition 4) using explicitly composition and projection operators. Instead, it is very useful to give primitive constraint module descriptions as schemata which can be reused by instantiating them with appropriate input and output variables. To support this we follow an approach which handles module instantiation and composition simultaneously. Modules are instantiated using a declaration `[outputlist]= modulename(inputlist);`

```
#module ASP dataViews
  (Rel(int, string, string, int, string, int) event,
   Rel(int, string, int) resource,
   Rel(int) session,
   Rel(string,int) roomCapacity)
  [Rel(int) maxEvent,
   Rel(int) maxResource,
   Rel(string) room,
   Rel(string) lecturer]

% Determine problem dimensions
eventId(I) :- event(I,CC,T,D,L,C).
maxEvent(I) :- eventId(I), not eventId(I+1).
resourceId(I) :- resource(I,R,S).
maxResource(I) :- resourceId(I), not resourceId(I+1).

% Rooms and personnel
room(R) :- resource(I,R,S).
lecturer(L) :- event(I,CC,l,D,L,C), L!=noname.

...

#endmodule
```

```
#module SYSTEM main()

% Data (problem instance)
[event] = eventData();
[day,session,resource,roomCapacity] =
  resourcesData();

% Different views of data
[maxEvent, maxResource, room, lecturer] =
  dataViews(event,resource,session,roomCapacity);

% Allocating resources
[resourceOf] =
  allDifferent(indexOfTrueElement(maxEvent),
               indexOfTrueElement(maxResource));

% Recover rooms and sessions from resources
[occurs] = allocation(resource,
                      arrayToRel(resourceOf));

% Checking the feasibility allocation
[] = testAllocation(occurs);

#solve[occurs]

#endmodule
```

Fig. 2. Examples of a constraint module and a module system as illustrated in Fig. 1

where `modulename` is the name of the module being instantiated, and `inputlist` and `outputlist` are the lists of input and output variables, respectively. This allows for writing a module composition $\mathcal{M}_1 \triangleright \mathcal{M}_2$ as suitable module instantiations: $[x_1, x_2, \ldots] = \mathcal{M}_1(\ldots)$; $[\ldots] = \mathcal{M}_2(x_1, x_2, \ldots)$; where appropriate output variables of $\mathcal{M}_1$ are used as input variables of $\mathcal{M}_2$. A module system is described as a sequence $\mathcal{M}_1; \mathcal{M}_2; \ldots; \mathcal{M}_n$; of such instantiation declarations which is acyclic, i.e., output variables of $\mathcal{M}_i$ cannot be used as input variables for any $\mathcal{M}_j$, $j \leq i$. This guarantees that the set of declarations can be seen as a well-formed composition $\mathcal{M}'_1 \triangleright (\mathcal{M}'_2 \triangleright (\ldots \triangleright \mathcal{M}'_n) \ldots)$ where $\mathcal{M}'_i$s are the corresponding instantiated constraint modules. The projection operator is handled implicitly in the instantiation of modules. For the top level of a module system we provide an explicit projection operator as the `#solve`$[\cdot]$ directive for defining the actual output variables of the whole module system.

A simplified example of a constraint module and a module system is given in Fig. 2. Each module description begins with a header line. The keyword "`#module`" is followed by (i) the language identifier, e.g., `SAT`, `ASP`, `CP`, or `SYSTEM`, (ii) the name of the module, and (iii) the specification of input and output variables enclosed in parentheses "`(...)`" and brackets "`[...]`", respectively. The types of variables are declared using elementary types (`int`, `string`, ...) and type constructors such as `Rel`.[2] Local variables (if any) and their types are declared with lines that begin with the keyword `#type`. A module description ends with a line designated by a keyword `#endmodule`. The module instantiation declarations need to be well-typed, i.e., the given input and output variables must conform to the module interfaces. The top-level module is distinguished by the reserved name `main` and the `#solve` directive for defining the output variables of the whole module system can be used only there.

[2] Description of a complete typing mechanism is beyond the scope of this paper. For now, we aim at type specifications which allow for static type checking.

6 Computational Aspects and Benefits of the Modular Approach

In this section we consider computational aspects related to module systems. First we analyze how certain computational properties of individual constraint modules are related to those of more complex module systems. Then we show how the structure of a module system can be exploited when one is interested in finding a satisfying assignment for a subset of the output variables of the module system.

We describe computational properties of a constraint module under the terms *checkable*, *solvable*, and *finite output for fixed input*, defined as follows.

Definition 9. *A constraint module* $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$

- *is checkable if and only if given any assignment* τ *over the variables in* $\mathcal{I} \cup \mathcal{O}$*, it can be decided whether* $\tau \in \mathsf{Solutions}(\mathcal{M})$*;*
- *is solvable if and only if there is a computable function that, given any assignment* τ *over the variables in* $\mathcal{I}$*, returns* $\tau' \in \mathsf{SolutionOut}(\mathcal{M}, \tau)$ *if such an assignment exists, and reports unsatisfiability otherwise; and*
- *has* FOFI *(finite output for fixed input) if and only if (i) the set* $\mathsf{SolutionOut}(\mathcal{M}, \tau)$ *is finite for any assignment* τ *over the variables in* $\mathcal{I}$*, and (ii) there is a computable function that, given any assignment* τ *over the variables in* $\mathcal{I}$*, outputs* $\mathsf{SolutionOut}(\mathcal{M}, \tau)$.

The knowledge about a specific property for constraint modules $\mathcal{M}$ and $\mathcal{M}'$ is not necessarily enough to guarantee that the property holds for a module system obtained by composition and/or projection operations from $\mathcal{M}$ and $\mathcal{M}'$. Clearly, if $\mathcal{M}$ and $\mathcal{M}'$ are checkable constraint modules, then $\mathcal{M} \triangleright \mathcal{M}'$ is checkable, too. Solvability of $\mathcal{M}$ and $\mathcal{M}'$ does not, however, imply that $\mathcal{M} \triangleright \mathcal{M}'$ is solvable. For instance, let $\mathcal{M} = \langle \mathcal{C}, \emptyset, \{a\} \rangle$ and $\mathcal{M}' = \langle \mathcal{C}', \{a\}, \{b\} \rangle$ be solvable constraint modules such that $\mathsf{Solutions}(\mathcal{M}) = \{\{a \mapsto 1\}, \{a \mapsto 2\}\}$, $\mathsf{Solutions}(\mathcal{M}') = \{\{a \mapsto 2, b \mapsto 2\}\}$, and $\mathcal{M} \triangleright \mathcal{M}'$ is defined. Assume that the computable function for $\mathcal{M}$ always returns $\tau = \{a \mapsto 1\}$. Now, $\mathsf{SolutionOut}(\mathcal{M}', \tau) = \emptyset$, and this would lead us to think that $\mathsf{Solutions}(\mathcal{M} \triangleright \mathcal{M}') = \emptyset$. But this is in contradiction with $\mathsf{Solutions}(\mathcal{M} \triangleright \mathcal{M}') = \{a \mapsto 2, b \mapsto 2\}$. If we in addition assume that $\mathcal{M}$ and $\mathcal{M}'$ have the FOFI property, then $\mathcal{M} \triangleright \mathcal{M}'$ is solvable and, moreover, has the FOFI property.

For projection, the situation is slightly different. If $\mathcal{M}$ is a checkable constraint module, then $\pi_{\mathcal{O}}(\mathcal{M})$ is not necessarily checkable for $\mathcal{O} \subset \mathsf{Output}(\mathcal{M})$. Given an assignment τ over $\mathsf{Input}(\mathcal{M}) \cup \mathcal{O} \subset \mathsf{Input}(\mathcal{M}) \cup \mathsf{Output}(\mathcal{M})$, we cannot decide whether $\tau \in \mathsf{Solutions}(\pi_{\mathcal{O}}(\mathcal{M}))$ because we do not know the assignment for variables in $\mathsf{Output}(\mathcal{M}) \setminus \mathcal{O}$. If, in addition, $\mathcal{M}$ is solvable, then using the projection τ' of τ to $\mathsf{Input}(\mathcal{M})$ we can compute $\tau'' \in \mathsf{SolutionOut}(\mathcal{M}, \tau')$ and the projection of τ'' to $\mathcal{O}$. Thus $\pi_{\mathcal{O}}(\mathcal{M})$ is solvable (and checkable).

The discussion above is summarized in the following proposition.

Proposition 1. *Let* $\mathcal{M}$ *and* $\mathcal{M}'$ *be constraint modules s.t.* $\mathcal{M} \triangleright \mathcal{M}'$ *is defined, and* $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$*. If* $\mathcal{M}$ *and* $\mathcal{M}'$ *are checkable, then* $\mathcal{M} \triangleright \mathcal{M}'$ *is checkable. If* $\mathcal{M}$ *is solvable, then* $\pi_{\mathcal{O}}(\mathcal{M})$ *is solvable. If* $\mathcal{M}$ *and* $\mathcal{M}'$ *have FOFI, then* $\mathcal{M} \triangleright \mathcal{M}'$ *and* $\pi_{\mathcal{O}}(\mathcal{M})$ *have FOFI.*

Next we underline conditions under which parts of a module system can be neglected when solving a constraint model based on that system. Namely, we define the *cone-of-influence reduction* for module systems, which is based on the concepts of *total module systems* and *don't care variables*.

Definition 10. *A constraint module $\mathcal{M}$ is* total *if* $\mathsf{SolutionOut}(\mathcal{M}, \tau) \neq \emptyset$ *for all assignments τ over* $\mathsf{Input}(\mathcal{M})$.

If $\mathcal{M}_1$ and $\mathcal{M}_2$ are total module systems such that $\mathcal{M}_1 \triangleright \mathcal{M}_2$ is defined, then $\mathcal{M}_1 \triangleright \mathcal{M}_2$ is total. Furthermore $\pi_{\mathcal{O}}(\mathcal{M})$ is total for any total $\mathcal{M}$ and $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$.

Seen as a black-box entity, testing totality from the outside is hard even on the level of constraint modules. However, if the declarative implementation of the module is known, there are easy-to-test syntactic conditions guaranteeing totality. For example, in Boolean circuit satisfiability, we know that if no gate of a circuit is constrained to a specific truth value, any module implemented with such a Boolean circuit is total. In practice, when implementing reusable modules for inclusion in a module library, the totality of a module could be explicitly declared in the module interface specification.

Definition 11. *Given a constraint module $\mathcal{M}$, $x \in \mathsf{Input}(\mathcal{M})$, and $y \in \mathsf{Output}(\mathcal{M})$, we say that x is a* $\mathcal{M}$-don't care *w.r.t. y, if for any assignment τ over* $\mathsf{Input}(\mathcal{M}) \setminus \{x\}$,

$$\{\pi_{\{y\}}(\tau') \mid \tau' \in \mathsf{SolutionOut}(\mathcal{M}, \tau \cup \tau_1)\} = \{\pi_{\{y\}}(\tau') \mid \tau' \in \mathsf{SolutionOut}(\mathcal{M}, \tau \cup \tau_2)\}$$

for all pairs of assignments τ_1, τ_2 for x.

As in the case of totality, in general checking whether a given input variable is a don't care is hard when constraint modules are seen as black-box entities. But again, if the declarative implementation of the module is known, there are easy-to-test syntactic conditions which guarantee that a variable is a don't care. For example, if a CNF formula can be split into two disjoint components, i.e., sets of clauses which do not share variables. A similar check can be done, e.g., for ASP programs and CSP instances.

In addition to totality and don't cares, we use the concept of *relevant I/O variables*. Let $\mathsf{CM}(\mathcal{M})$ denote the set of constraint modules appearing in a module system $\mathcal{M}$. For instance, if $\mathcal{M} = \pi_{\mathcal{O}}(\mathcal{M}_1 \triangleright \mathcal{M}_2)$ then $\mathsf{CM}(\mathcal{M}) = \{\mathcal{M}_1, \mathcal{M}_2\}$.

Definition 12. *Given a module system $\mathcal{M}$ and $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$, the set of relevant I/O variables in $\mathcal{M}$ w.r.t. $\mathcal{O}$, denoted by* $\mathsf{Rel}(\mathcal{M}, \mathcal{O})$*, is the smallest set S of variables that fulfills the following conditions:*

- $\mathcal{O} \subseteq S$.
- $\mathsf{Input}(\mathcal{M}') \subseteq S$ *for each non-total* $\mathcal{M}' \in \mathsf{CM}(\mathcal{M})$.
- *If $y \in S$, then for each total* $\mathcal{M}' \in \mathsf{CM}(\mathcal{M})$ *such that* $y \in \mathsf{Output}(\mathcal{M}')$, $\{x \in \mathsf{Input}(\mathcal{M}') \mid x$ *is not* $\mathcal{M}'$*-don't care w.r.t.* $y\} \subseteq S$.

The cone-of-influence reduction is used in disregarding parts of a module system in the case we are only interested in the values assigned to a subset of the output of the system.

Definition 13. *Given a module system $\mathcal{M}$ and a set X of variables, the* module system reduction $\mathcal{M}|_X$ *is defined as follows.*

– *If $\mathcal{M}$ is a constraint module, then*
$\mathcal{M}|_X = \begin{cases} \mathcal{E} \text{ (the empty module)} & \text{, if } \mathsf{Output}(\mathcal{M}) \cap X = \emptyset \text{ and } \mathcal{M} \text{ is total} \\ \mathcal{M} & \text{, otherwise.} \end{cases}$
– *If $\mathcal{M}$ is of the form $\mathcal{M}_1 \triangleright \mathcal{M}_2$, then $\mathcal{M}|_X = (\mathcal{M}_1|_X \triangleright \mathcal{M}_2|_X)$.*
– *If $\mathcal{M}$ is of the form $\pi_O(\mathcal{M}')$, then $\mathcal{M}|_X = \pi_{\mathsf{Output}(\mathcal{M}'|_X) \cap O}(\mathcal{M}'|_X)$.*

Given a module system $\mathcal{M}$ and a set of variables $\mathcal{O}$, the cone-of-influence reduction *of $\mathcal{M}$ w.r.t. $\mathcal{O}$ is the module system $\mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})}$.*

For finding a satisfying assignment for $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$ of a module system $\mathcal{M}$, one needs to consider only the *subsystem* $\mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})}$ of $\mathcal{M}$.

Proposition 2. *Given a module system $\mathcal{M}$ and a set of variables $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$, then $\{\pi_{\mathcal{O}}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{M})\} = \{\pi_{\mathcal{O}}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})})\}$.*

Example 6. Consider the module system $\mathcal{M} = (\mathcal{M}_1 \triangleright \mathcal{M}_2) \triangleright (\mathcal{M}_3 \triangleright \mathcal{M}_4)$ illustrated in Fig. 3. Thus, $\mathsf{Input}(\mathcal{M}) = \{a, b, c\}$ and $\mathsf{Output}(\mathcal{M}) = \{d, e, f, g\}$. The constraint module $\mathcal{M}_2$ represented with gray in Fig. 3 is not total, while the other constraint modules in $\mathsf{CM}(\mathcal{M})$, i.e., $\mathcal{M}_1$, $\mathcal{M}_3$, and $\mathcal{M}_4$, are total. Assume that, in addition, it is known that e and f are $\mathcal{M}_4$-don't cares w.r.t. g. Assume that we are only interested in finding a satisfying assignment for $\mathcal{O} = \{g\}$. By Proposition 2 we can exploit the cone-of-influence reduction. The set of relevant I/O variables $\mathsf{Rel}(\mathcal{M}, \mathcal{O}) = X = \{a, b, c, d, g\}$ because $\mathcal{O} \subseteq \mathsf{Rel}(\mathcal{M}, \mathcal{O})$, $\mathsf{Input}(\mathcal{M}_2) \subseteq \mathsf{Rel}(\mathcal{M}, \mathcal{O})$, d is not $\mathcal{M}_4$-don't care w.r.t. $g \in \mathsf{Output}(\mathcal{M}_4)$, and a and b are not $\mathcal{M}_1$-don't cares w.r.t. $d \in \mathsf{Output}(\mathcal{M}_1)$. Using the set of relevant I/O variables, the cone-of-influence reduction of $\mathcal{M}$ w.r.t. $\mathcal{O}$ is

$$\begin{aligned} \mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})} &= (\mathcal{M}_1 \triangleright \mathcal{M}_2)|_X \triangleright (\mathcal{M}_3 \triangleright \mathcal{M}_4)|_X \\ &= (\mathcal{M}_1|_X \triangleright \mathcal{M}_2|_X) \triangleright (\mathcal{M}_3|_X \triangleright \mathcal{M}_4|_X) \\ &= (\mathcal{M}_1 \triangleright \mathcal{M}_2) \triangleright (\mathcal{E} \triangleright \mathcal{M}_4) \\ &= (\mathcal{M}_1 \triangleright \mathcal{M}_2) \triangleright \mathcal{M}_4. \end{aligned}$$

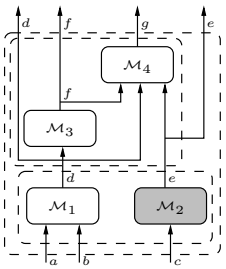


Fig. 3. A module system

7 Conclusions

We develop a generic framework for module-based constraint modeling using multiple modeling languages within the same model. Within the framework, constraint models are constructed as module systems which are composed of constraint modules each having a clean input/output interface specification. This approach has many interesting properties. First of all, individual constraint modules can be implemented using a constraint language most suitable for modeling the constraint in question. The approach paves the way for reusable constraint module libraries and also allows for multiple modelers to implement parts of a constraint model in parallel. Our framework supports modular multi-language modeling by treating different constraint languages on equal terms whereas previous approaches can be seen as extensions of a given basic language with features from other languages. The modular construction of constraint models as module systems yields in itself a structured view to the model which can be exploited when solving the model. We describe a system level cone-of-influence reduction, which allows parts of the module system to be disregarded when solving a constraint model, without the need to consider properties specific to the constraint languages employed in implementing the individual constraint modules.

One of the main goals of this paper is to provide foundations for developing methods for solving constraint satisfaction problems expressed using the multi-language framework. A number of interesting approaches can be studied. In a hybrid system individual constraint modules (or parts of the module system modeled using the same constraint language) are solved using a solver aimed at the language and the multiple language specific solvers interact to compute solutions to the whole constraint model. In a translation-based approach all parts of the model are mapped into a single constraint language for which highly efficient off-the-shelf solvers are available. For example, there is interesting recent work on bit-blasting more general CSP models into SAT [21]. Another interesting paradigm is the extension of SAT to Satisfiability Module Theories (SMT), into which e.g. stable model computation can be very compactly encoded [22]. Additionally, the modular structure of module systems poses interesting research topics such as harnessing the I/O interfaces in developing novel decision heuristics and devising techniques to instantiate and ground module schemata lazily.

References

1. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Proc. LPNMR'97. Volume 1265 of LNCS., Springer (1997) 290–309
2. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Proc. ICLP'06. Volume 4079 of LNCS., Springer (2006) 376–390
3. Balduccini, M.: Modules and signature declarations for A-Prolog: Progress report. In: Proc. SEA'07. (2007) 41–55
4. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory and Practice of Logic Programming **8**(5–6) (2008) 717–761
5. Janhunen, T.: Modular equivalence in general. In: Proc. ECAI'08, IOS Press (2008) 75–79

6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Proc. IJCAI'05, Professional Book Center (2005) 90–96
7. Elkabani, I., Pontelli, E., Son, T.: Smodels^{a} - a system for computing answer sets of logic programs with aggregates. In: Proc. LPNMR'05. Volume 3662 of LNCS., Springer (2005) 427–431
8. Gebser, M., et al.: Clingcon. Available at `http://www.cs.uni-potsdam.de/clingcon/` (2009)
9. Tari, L., Baral, C., Anwar, S.: A language for modular answer set programming: Application to ACC tournament scheduling. In: Proc. ASP'05. Volume 142 of CEUR Workshop Proceedings., CEUR-WS.org (2005)
10. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Proc. ICLP'05. Volume 3668 of LNCS., Springer (2005) 52–66
11. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence **53**(1-4) (2008) 251–287
12. Castro, L., Swift, T., Warren, D.: Xasp. Available at `http://xsb.sourceforge.net/` (2009)
13. El-Khatib, O., Pontelli, E., Son, T.: Integrating an answer set solver into Prolog: ASP-PROLOG. In: Proc. LPNMR'05. Volume 3662 of LNCS., Springer (2005) 399–404
14. Pontelli, E., Son, T., Baral, C.: A logic programming based framework for intelligent web services composition. In: Managing Web Services Quality: Measuring Outcomes and Effectiveness. IDEA Group Publishing (2008)
15. Elkhatib, O.: Towards a Programming Environment for Answer Set Programming. PhD thesis, Department of Computer Science, New Mexico State University (2007)
16. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Proc. LOPSTR'03. Volume 3018 of LNCS., Springer (2004) 214–232
17. Frisch, A., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: ESSENCE: A constraint language for specifying combinatorial problems. Constraints **13**(3) (2008) 268–306
18. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., de la Banda, M.G., Wallace, M.: The design of the Zinc modelling language. Constraints **13**(3) (2008) 229–267
19. Goltz, H.J., Matzke, D.: University timetabling using constraint logic programming. In: Proc. PADL'99. Volume 1551 of LNCS., Springer (1999) 320–334
20. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. Annals of Mathematics and Artificial Intelligence **51**(2-4) (2007) 195–228
21. Huang, J.: Universal Booleanization of constraint models. In: Proc. CP'08. Volume 5202 of LNCS., Springer (2008) 144–158
22. Niemelä, I.: Stable models and difference logic. Annals of Mathematics and Artificial Intelligence **53**(1–4) (2008) 313–329

Representing Constraint Satisfaction Problems in Answer Set Programming

Marcello Balduccini

Intelligent Systems, OCTO
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract In this paper we describe an approach for integrating answer set programming (ASP) and constraint programming, in which ASP is viewed as a specification language for constraint satisfaction problems. ASP programs are written in such a way that their answer sets encode the desired constraint satisfaction problems; the solutions of those problems are then found using constraint satisfaction techniques. Differently from other methods of integrating ASP and constraint programming, our approach has the advantage of allowing the use of off-the-shelf, unmodified ASP solvers and constraint solvers, and of global constraints, which substantially increases the practical applicability of the approach to industrial-size problems.

1 Introduction

Answer Set Programming (ASP) [1,2,3] is a declarative programming paradigm with roots in the research on non-monotonic logic and on the semantics of default negation of Prolog.

In recent years, ASP has been applied successfully to solving complex problems (e.g. [4,5]), and the underlying language has been extended in various directions to broaden its applicability even further (e.g. [6,7]).

Particular interest has been recently devoted to the integration of ASP with Constraint Logic Programming (CLP) (see [8,9] and the *clingcon* system[1]), aimed at combining the ease of knowledge representation of ASP with the powerful support for numerical computations of CLP. Such approaches are based on an extension of the ASP language, and on the use of answer set and constraint solvers modified to work together. Although the combination of ASP and CLP showed substantial performance improvements over ASP alone, the restriction of using ad-hoc ASP and CLP solvers limits the practical applicability of the approach. In fact, programmers can no longer select the solvers that best fit their needs (most notably, SMODELS, DLV, SWI-Prolog and SICStus Prolog), as is instead commonly done in ASP. Another limit for the practical applicability of the approach is the lack of specific support for global constraints. Without global constraints, applications' performance is often heavily impacted by the combinatorial

[1] http://www.cs.uni-potsdam.de/clingcon/

explosion of the underlying search space, even for relatively small (compared to the intended application domain) problem instances.

In this paper we describe an approach for integrating ASP and constraint programming, in which ASP is viewed as a specification language for constraint satisfaction problems. ASP programs are written in such a way that their answer sets encode the desired constraint satisfaction problems; the solutions to those problems are found using constraint satisfaction techniques. Both the answer sets and the solutions to the constraint problems can be computed with arbitrary off-the-shelf solvers, as long as a (relatively simple) translation procedure is defined from the ASP encoding of the constraint problems to the input language of the constraint solver selected. Moreover, our approach allows the use of the global constraints available in the selected constraint solver. Compared to the other approaches to the integration of ASP and CLP, our technique allows programmers to exploit the full power of the state-of-the-art solvers when tackling industrial-size problems.

The paper is organized as follows. We start by giving background notions of ASP and constraint satisfaction. In Section 3, we describe our encoding of constraint satisfaction problems in ASP and define its semantics. In Section 4 we explain how to compute the solutions to the constraint problems encoded by the answer sets of ASP programs. Section 5 compares our approach with existing research on integrating ASP and CLP. In Section 6, we draw conclusions.

2 Background

The syntax and semantics of ASP are defined as follows. Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual. A literal is either an atom a or its strong (also called classical or epistemic) negation $\neg a$. The sets of atoms and literals formed from Σ are denoted by $atoms(\Sigma)$ and $literals(\Sigma)$ respectively.

A *rule* is a statement of the form:[2]

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

where h and l_i's are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe h. We call h the *head* of the rule, and $\{l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n\}$ the *body* of the rule. Given a rule r, we denote its head and body by $head(r)$ and $body(r)$ respectively.

A *program* is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules over Σ. Often we denote programs by just the second element of the pair, and let the signature be defined implicitly. In that case, the signature of Π is denoted by $\Sigma(\Pi)$.

[2] For simplicity we focus on non-disjunctive programs. Our results extend to disjunctive programs in a natural way.

A set A of literals is *consistent* if no two complementary literals, a and $\neg a$, belong to A. A literal l is *satisfied* by a consistent set of literals A if $l \in A$. In this case, we write $A \models l$. If l is not satisfied by A, we write $A \not\models l$. A set $\{l_1, \ldots, l_k\}$ of literals is satisfied by a set of literals A ($A \models \{l_1, \ldots, l_k\}$) if each l_i is satisfied by A.

Programs not containing default negation are called *definite*. A consistent set of literals A is *closed* under a definite program Π if, for every rule of the form (1) such that the body of the rule is satisfied by A, the head belongs to A.

Definition 1. *A consistent set of literals A is an* answer set *of definite program Π if A is closed under all the rules of Π and A is set-theoretically minimal among the sets closed under all the rules of Π.*

The *reduct* of a program Π with respect to a set of literals A, denoted by Π^A, is the program obtained from Π by deleting:

- Every rule, r, such that $l \in A$ for some expression of the form not l from the body for r;
- All expressions of the form not l from the bodies of the remaining rules.

We are now ready to define the notion of answer set of a program.

Definition 2. *A consistent set of literals A is an* answer set *of program Π if it is an answer set of the reduct Π^A.*

To simplify the programming task, variables are often allowed to occur in ASP programs. A rule containing variables (called a *non-ground* rule) is then viewed as a shorthand for the set of its *ground instances*, obtained by replacing the variables in it by all the possible ground terms. Similarly, a non-ground program is viewed as a shorthand for the program consisting of the ground instances of its rules.

Let us now turn our attention to Constraint Programming. In this paper we follow the traditional definition of constraint satisfaction problem. The one that follows is adapted from [10]. A *Constraint Satisfaction Problem (CSP)* is a triple $\langle X, D, C \rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of domains, such that D_i is the domain of variable x_i (i.e. the set of possible values that the variable can be assigned), and C is a set of constraints.[3] Each constraint $c \in C$ is a pair $c = \langle \sigma, \rho \rangle$ where σ is a list of variables and ρ is a subset of the Cartesian product of the domains of such variables.

An *assignment* is a pair $\langle x_i, a \rangle$, where $a \in D_i$, whose intuitive meaning is that variable x_i is assigned value a. A *compound assignment* is a set of assignments to distinct variables from X. A *complete assignment* is a compound assignment to all the variables in X.

[3] Strictly speaking, the use of the same index i across sets X and D in the above definition of the set of domains would require X and D to be ordered. However, as the definition of CSP is insensitive to the particular ordering chosen, we follow the approach, common in the literature on constraint satisfaction, of simply considering X and D sets and abusing notation slightly in the definition of CSP.

A constraint $\langle \sigma, \rho \rangle$ specifies the acceptable assignments for the variables from σ. We say that such assignments *satisfy* the constraint. A *solution* to a CSP $\langle X, D, C \rangle$ is a complete assignment satisfying every constraint from C.

Constraints can be represented either *extensionally*, by specifying the pair $\langle \sigma, \rho \rangle$, or *intensionally*, by specifying an expression involving variables, such as $x < y$. In this paper we focus on constraints represented intensionally. A *global constraint* is a constraint that captures a relation between a non-fixed number of variables [11], such as $sum(x, y, z) < w$ and *all_different*$(x_1, \ldots, x_k)$.

One should notice that the mapping of an intensional constraint specification into a pair $\langle \sigma, \rho \rangle$ depends on the *constraint domain*. For example, the expression $1 \leq x < 2$ corresponds to the constraint $\langle \langle x \rangle, \{\langle 1 \rangle\} \rangle$ if the finite domain is considered, while it corresponds to $\langle \langle x \rangle, \{\langle v \rangle \mid v \in [1, 2)\} \rangle$ in a continuous domain. For this reason, and in line with the CLP Schema [12,13], in this paper we assume that a CSP includes the specification of the intended constraint domain.

3 Representing constraint problems in ASP

Our approach consists in writing ASP programs whose answer sets encode the desired constraint satisfaction problems (CSPs). The solutions to the CSPs are then computed using constraint satisfaction techniques.

CSPs are encoded in ASP using the following three types of statements.

– A constraint domain declaration is a statement of the form:

$$cspdomain(\mathcal{D})$$

where $\mathcal{D}$ is a constraint domain such as fd, q, or r. Informally, the statement states that the CSP is over the specified constraint domain, thereby fixing an interpretation for the intensionally specified constraints.

– A constraint variable declaration is a statement of the form:

$$cspvar(x, l, u)$$

where x is a ground term denoting a variable of the CSP (CSP variable or constraint variable for short), and l and u are numbers from the constraint domain. The statement informally states that the domain of x is $[l, u]$.[4]

– A constraint statement is a statement of the form:

$$required(\gamma)$$

where γ is an expression that intensionally represents a constraint on (some of) the variables specified by the $cspvar$ statements. Intuitively the statement says that

[4] As an alternative, the domain of the variables could also be specified using constraints. We use a separate statement for similarity with CLP languages.

the constraint intensionally represented by γ is required to be satisfied by any solution to the CSP. For the purpose of specifying global constraints, we allow γ to contain expressions of the form $[\delta/k]$. If δ is a function symbol, the expression intuitively denotes the sequence of all variables formed from function symbol δ and with arity k, ordered lexicographically.[5] For example, if given CSP variables $v(1), v(2), v(3)$, $[v/1]$ denotes the sequence $\langle v(1), v(2), v(3)\rangle$. If δ is a relation symbol and $k \geq 1$, the expression intuitively denotes the sequence $\langle e_1, e_2, \ldots, e_n\rangle$ where e_i is the last element of the i^{th} k-tuple satisfying relation δ, according to the lexicographic ordering of such tuples. For example, given a relation r' defined by $r'(a,3), r'(b,1), r'(c,2)$ (that is, by tuples $\langle a,3\rangle$, $\langle b,1\rangle$, $\langle c,2\rangle$), the expression $[r'/2]$ denotes the sequence $\langle 3,1,2\rangle$.

Example 1. The following sets of statements encode simple CSPs:

$$
\begin{array}{l}
A_1 = \\
\{cspdomain(\mathit{fd}), \\
\ cspvar(v(1),1,3), cspvar(v(2),2,5), cspvar(v(3),1,4), \\
\ required(v(1)+v(2) \leq 4), required(v(2)-v(3)>1), \\
\ required(sum([v/1]) \geq 4)\}
\end{array}
$$

$$
\begin{array}{l}
A_2 = \\
\{cspdomain(\mathit{fd}), \\
\ cspvar(start(j1),1,100), cspvar(start(j2),25,100), \\
\ cspvar(start(j3),30,80), cspvar(start(j4),45,150), \\
\ required(serialized([start/1],[duration/2]))\}
\end{array}
$$

In the rest of this paper, we consider signatures that contain:

- relations $cspdomain$, $cspvar$, $required$;
- constant symbols for the constraint domains $\mathcal{FD}$, $\mathcal{Q}$, and $\mathcal{R}$
- suitable symbols for the variables, functions and relations used in the CSP;
- the numerical constants needed to encode the CSP.

Let A be a set of atoms formed from relations $cspdomain$, $cspvar$, and $required$. We say that A is a *well-formed CSP definition* if:

- A contains exactly one constraint domain declaration;
- The same CSP variable does not occur in two or more constraint variable declarations of A;
- Every CSP variable that occurs in a constraint statement from A also occurs in a constraint variable declaration from A.

[5] The choice of a particular order is due to the fact that global constraints that accept multiple lists often expect the elements in the same position throughout the lists to be in a certain relation. More sophisticated techniques for the specification of lists are possible, but, according to our analysis of the use of global constraints in constraint satisfaction, this method should work well in most cases.

Example 2. The following is not a well-formed CSP definition:

$$\{cspdomain(\mathit{fd}), cspvar(x, 1, 2), required(x < y)\}.$$

On the other hand, the sets of atoms from Example 1 are well-formed CSP definitions.

Let A be a well-formed CSP definition. The CSP *defined* by A is the triple $\langle X, D, C\rangle$ such that:

- $X = \{x_1, x_2, \ldots, x_k\}$ is the set of all CSP variables from the constraint variable declarations in A;
- $D = \{D_1, D_2, \ldots, D_k\}$ is the set of domains of the variables from X. The domain D_i of variable x_i is given by arguments l and u of the constraint variable declaration of x_i in A, and consists of the segment between l and u in the constraint domain specified by the constraint domain declaration from A.
- C is a set containing a constraint γ' for each constraint statement $required(\gamma)$ of A. Constraint γ' is obtained by:
  1. Replacing the expressions of the form $[f/k]$, where f is a function symbol, by the list of variables from X formed by f and of arity k, ordered lexicographically;
  2. Replacing the expressions of the form $[r/k]$, where r is a relation symbol and $k \geq 1$, by the sequence $\langle e_1, \ldots, e_n\rangle$, where, for each i, $r(t_1, t_2, \ldots, t_{k-1}, e_i)$ is the i^{th} element of the sequence, ordered lexicographically, of atoms from A formed by relation r;
  3. Interpreting the resulting intensionally specified constraint w.r.t. the constraint domain specified by the constraint domain declaration from A.

Example 3. Set A_1 from Example 1 defines the CSP $\langle X_1, D_1, C_1\rangle$:

- $X_1 = \{v(1), v(2), v(3)\}$
- $D_1 = \{\{1, 2, 3\}, \{2, 3, 4, 5\}, \{1, 2, 3, 4\}\}$
- $C_1 = \left\{ \begin{array}{l} v(1) + v(2) \leq 4,\ v(2) - v(3) > 1, \\ sum(v(1), v(2), v(3)) \geq 4 \end{array} \right\}$

Consider A_2 from Example 1 and

$$\begin{array}{l} I = \{duration(j1, 20), duration(j2, 10), \\ \qquad duration(j3, 50), duration(j4, 60)\}. \end{array}$$

Set $A_2 \cup I$ defines the CSP $\langle X_2, D_2, C_2\rangle$:

- $X_2 = \{start(j1), start(j2), start(j3), start(j4)\}$
- $D_2 = \{\{1, 2, \ldots, 100\}, \{25, \ldots, 100\}, \ldots\}$
- $C_2 =$
  $\{serialized([start(j1), start(j2), start(j3), start(j4)],$
  $[20, 10, 50, 60])\}$

Let A be a set of literals. We say that A *contains* a well-formed CSP definition if the set of atoms from A formed by relations $cspdomain$, $cspvar$, and $required$ is a well-formed CSP definition. We also say that a CSP is defined by a set of literals A if it is defined by the well-formed CSP definition contained in A. Notice that, if a set A of literals does not contain a well-formed CSP definition, A does not define any CSP. For simplicity, in the rest of the discussion we omit the term "well-formed" and simply talk about CSP definitions.

Definition 3. *A pair $\langle A, \alpha \rangle$ is an extended answer set of program Π iff A is an answer set of Π and α is a solution to the CSP defined by A.*

Example 4. Consider set A_1 from Example 1. An extended answer set of A_1 is:

$$\langle A_1, \{(v(1), 1), (v(2), 3), (v(3), 1)\}\rangle.$$

Example 5. Consider the program:

$$P_1 = \begin{cases} index(1).\ index(2).\ index(3).\ index(4). \\ \\ cspdomain(\mathit{fd}). \\ \\ cspvar(v(I), 1, 10) \leftarrow index(I). \\ \\ required(v(I1) - v(I2) \geq 3) \leftarrow \\ \qquad index(I1),\ index(I2), \\ \qquad I2 = I1 + 1. \end{cases}$$

An extended answer set of P_1 is:

$$\begin{array}{l} \langle \{index(1), \ldots, index(4), cspdomain(\mathit{fd}), \\ \quad cspvar(v(1), 1, 10), \ldots, cspvar(v(4), 1, 10), \\ \quad required(v(1) - v(2) \geq 3), \ldots, \\ \quad required(v(3) - v(4) \geq 3)\}, \\ \{(v(1), 10), (v(2), 7), (v(3), 4), (v(4), 1)\}\rangle \end{array}$$

Example 6. Consider the riddle:
"There are either 2 or 3 brothers in the Smith family. There is a 3 year difference between one brother and the next (in order of age). The age of the eldest brother is twice the age of the youngest. The youngest is at least 6 years old."

A program, P_2, that finds the solutions to the riddle is:

```
% There are either 2 or 3 brothers in the Smith family.
num_brothers(2) ← not num_brothers(3).
num_brothers(3) ← not num_brothers(2).

index(1).index(2).index(3).

is_brother(B) ←
        index(B), index(N),
        num_brothers(N),
        B ≤ N.

eldest_brother(1).
```

$$
\begin{array}{l}
youngest\_brother(B) \leftarrow \\
\qquad index(B), \\
\qquad num\_brothers(B).
\end{array}
$$

$$cspdomain(fd).$$

$$cspvar(age(B), 1, 80) \leftarrow index(B), is\_brother(B).$$

% 3 year difference between one brother and the next.

$$
\begin{array}{l}
required(age(B1) - age(B2) = 3)) \leftarrow \\
\qquad index(B1), index(B2), \\
\qquad is\_brother(B1), is\_brother(B2), \\
\qquad B2 = B1 + 1.
\end{array}
$$

% The eldest brother is twice as old as the youngest.

$$
\begin{array}{l}
required(age(BE) = age(BY) * 2) \leftarrow \\
\qquad index(BE), index(BY), \\
\qquad eldest\_brother(BE), \\
\qquad youngest\_brother(BY).
\end{array}
$$

% The youngest is at least 6 years old.

$$
\begin{array}{l}
required(age(BY) \geq 6) \leftarrow \\
\qquad index(BY), \\
\qquad youngest\_brother(BY).
\end{array}
$$

An extended answer set of P_2 is:

$$
\begin{array}{l}
\langle \{num\_brothers(3), \\
\quad cspvar(age(1), 1, 80), \ldots, cspvar(age(3), 1, 80), \ldots\}, \\
\quad \{(age(1), 12), (age(2), 9), (age(3), 6)\}\rangle,
\end{array}
$$

which states that there are 3 brothers, of age 12, 9, and 6 respectively. Notice that there is no extended answer set containing $num\_brothers(2)$.

4 Computing Extended Answer Sets

To compute the extended answer sets of a program, we combine the use of answer set solvers and constraint solvers. The algorithm is as follows:

Algorithm Alg_1
Input: program Π
Output: the set of extended answer sets of Π

1. $\mathcal{E} := \emptyset$
2. Let $\mathcal{A}$ be the set of answer sets of Π containing a CSP definition.
3. For each $A \in \mathcal{A}$:
   (a) Select a constraint solver $solve_{\mathcal{D}}$ for the constraint domain $\mathcal{D}$ specified by the constraint domain declaration from A.
   (b) Translate the CSP definition from A into an encoding $\chi_A^{\mathcal{D}}$ suitable for $solve_{\mathcal{D}}$.

(c) Let $\mathcal{S} = \{\alpha_1, \ldots, \alpha_k\}$ be the set of solutions returned by $solve_{\mathcal{D}}(\chi_A^{\mathcal{D}})$.
(d) For each $\alpha \in \mathcal{S}$, $\mathcal{E} := \mathcal{E} \cup \langle A, \alpha \rangle$.

4. Return $\mathcal{E}$.

As can be seen from step (3b), the algorithm relies on the correctness of the translation from the CSP definition to the encoding for the constraint solver. More precisely:

Definition 4. *A translation algorithm $Trans$ from CSP definitions to encodings suitable for a constraint solver $solve$ is* correct *if α is a solution to the CSP defined by A iff α is one of the answers returned by $solve(Trans(A))$.*

The following theorems deal with the soundness and completeness of Alg_1. Their proofs are not difficult, and are omitted to save space.

Theorem 1. *Let Π be a program and $Trans$ be a translation algorithm as above. If $\langle A, \alpha \rangle \in Alg_1(\Pi)$ and $Trans$ is correct, then $\langle A, \alpha \rangle$ is an extended answer set of Π.*

Theorem 2. *Let Π be a program and $Trans$ be a translation algorithm as above. If $\langle A, \alpha \rangle$ is an extended answer set of Π, $Trans$ is correct, and the solver selected for A at step (3a) of the algorithm is complete for $\chi_A^{\mathcal{D}}$, then $\langle A, \alpha \rangle \in Alg_1(\Pi)$.*

A convenient way to compute the solutions of the CSPs at step (3c) is to use the constraint solvers embedded in CLP systems. Therefore, we describe an algorithm to translate from a CSP definition to a CLP program. The translation algorithm assumes that the constraint variables that occur in the CSP definition being translated are *legal ground terms* for the CLP system, or that a suitable mapping to legal terms has implicitly taken place, and that the CLP system can handle all the constraint domains of interest. The algorithm is based on the CLP Schema [12,13].

Algorithm ψ
Input: a CSP definition A
Output: a CLP program P

1. $P := \emptyset$
2. $\nu := \emptyset$ { *CLP variables for the encoding of the CSP* }
3. $\theta := \emptyset$ { *body of the top-level clause of P* }
4. Retrieve atom $cspdomain(\mathcal{D})$ from A.
5. Add to P a directive:[6]

$$:- use\_module(library(cs))$$

   where cs is a suitable constraint solver for constraint domain $\mathcal{D}$ (e.g. clpfd, clpr).
6. For each $cspvar(x, l, u) \in A$:
   (a) $\nu := \nu \cup \{V_x\}$, where V_x is a fresh CLP variable.
   (b) $\theta := \theta \cup \{V_x \geq l, V_x \leq u\}$.
7. For each $required(\gamma_1) \in A$:
   (a) Obtain γ_2 from γ_1 by replacing every expression of the form $[f/k]$ in γ_1, where f is a function symbol, with the list of all the CSP variables of the form $f(t_1, t_2, \ldots, t_k)$ declared in A, ordered lexicographically.

[6] We use the syntax of SICStus [14]. The translation for other CLP systems is similar.

(b) Obtain γ_3 from γ_2 by replacing every expression of the form $[r/k]$ in γ_2, where r is a relation symbol and $k \geq 1$, by the list $[e_1, \ldots, e_n]$, where, for each i, $r(t_1, t_2, \ldots, t_{k-1}, e_i)$ is the i^{th} element of the list, ordered lexicographically, of atoms from A formed by r.
(c) Obtain γ_4 from γ_3 by replacing every occurrence of a CSP variable x in γ_3 by the corresponding V_x from ν.
(d) $\theta := \theta \cup \{\gamma_4\}$.
8. $\theta := \theta \cup \{labeling(\nu)\}$.[7]
9. $\lambda := \{(x, V_x) \mid x \in \nu\}$.
10. $P := P \cup \{solve(\lambda) :- \theta\}$.
11. Return P.

Example 7. Consider the CSP definition

$$A_3 = \begin{cases} cspdomain(\textit{fd}). \\ cspvar(x, 1, 5).\ cspvar(y, 1, 5). \\ required(x < y). \end{cases}$$

Its translation into CLP is:

$$\psi(A_3) = \begin{cases} :- use\_module(library(\textit{clpfd})). \\ \\ solve([(x, V_x), (y, V_y)]) :- \\ \qquad V_x \geq 1, V_x \leq 5, \\ \qquad V_y \geq 1, V_y \leq 5, \\ \qquad V_x < V_y, \\ \qquad labeling([V_x, V_y]). \end{cases}$$

Example 8. The CSP definition

$$A_4 = \begin{cases} cspdomain(\textit{fd}). \\ cspvar(var(a), 1, 5).\ cspvar(var(b), 2, 8). \\ duration(var(a), 4).\ duration(var(b), 2). \\ required(serialized([var/1], [duration/2])). \end{cases}$$

is translated by ψ into:

$$\psi(A_4) = \begin{cases} :- use\_module(library(\textit{clpfd})). \\ \\ solve([(var(a), V_x), (var(b), V_y)]) :- \\ \qquad V_x \geq 1, V_x \leq 5, \\ \qquad V_y \geq 2, V_y \leq 8, \\ \qquad serialized([V_x, V_y], [4, 2]), \\ \qquad labeling([V_x, V_y]). \end{cases}$$

The following theorem ensures the correctness of the translation.

[7] To simplify the notation, we allow sets to occur in CLP programs, although, strictly speaking, they would have to be encoded as Prolog lists.

Theorem 3. *The translation algorithm ψ is* correct.

Proof. *(Sketch)*
According to Definition 4, we have to prove that, if A is a CSP definition, then α is a solution to the CSP defined by A iff there exists a derivation from goal $solve(S)$ in $\psi(A)$ that succeeds with substitution $S|_\alpha$.

Left-to-right. Let α be a solution to the CSP defined by A and let us prove that there exists a derivation from goal $solve(S)$ in $\psi(A)$ that succeeds with substitution $S|_\alpha$. Because the value assigned by α to each variable x is by definition within the domain of the variable, the conditions added to the body of the clause for $solve$ in step (6b) are satisfied. Also, by definition α satisfies every constraint from the CSP defined by A. Hence, it is not difficult to see that the conditions added to the body of the clause in step (7d) are satisfied. Because all the conditions are satisfied by α, the call to predicate $labeling$ is bound to succeed, and the derivation is indeed successful.

Right-to-left. Now let α be such that goal $solve(S)$ in $\psi(A)$ succeeds with substitution $S|_\alpha$, and let us prove that α is a solution to the CSP defined by A. First of all, notice that, if goal $solve(S)$ succeeds with substitution $S|_\alpha$, then all the conditions in the body of the clause for $solve$ must be satisfied. Since the call to $labeling$ succeeds, all the V_x variables are instantiated in α. Hence, α is a compound assignment, and moreover it is a complete assignment by construction. Because the conditions added by step (6b) are all satisfied, all the variables are guaranteed to have values within the respective domains as specified in A. Finally, because the conditions added by step (7d) are all satisfied, it is not difficult to conclude that the constraints from A are satisfied by α. Hence, α is a solution to the CSP defined by A.

Q.E.D.

5 Related Work

The *clingcon* system[8] integrates the answer set solver Clingo and the constraint solver Gecode. The system thus differs significantly from ours in that programmers cannot arbitrarily select the most suitable ASP and constraint solvers for the task at hand. As the system is very recent, too little documentation is currently available about the system for a thorough analysis.

The approach proposed by Mellarkod, Gelfond and Zhang [15,9] is based on an extension, $\mathcal{AC}(\mathcal{C})$, of the syntax and semantics of ASP and CR-Prolog allowing the use of CSP-style constraints in the body of the rules. The assignment of values to the constraint variables is denoted by means of special atoms occurring in the body of the rules. Such atoms are treated as abducibles, and their truth determined by solving a suitable CSP. For example, an $\mathcal{AC}(\mathcal{C})$ program solving the same problem as the CSP defined in

[8] http://www.cs.uni-potsdam.de/clingcon/

Example 5 is:

$$
\begin{array}{l}
val(1).\ val(2).\ \ldots\ val(10).\\
\#csort(val).\\
\\
index(1).\ index(2).\ index(3).\ index(4).\\
var(v(I)) \leftarrow index(I).\\
\#mixed\ has\_value(var, val).\\
\\
\leftarrow V1 - V2 < 3,\\
\quad has\_value(v(I1), V1), has\_value(v(I2), V2),\\
\quad index(I1), index(I2), I2 = I1 + 1.
\end{array}
$$

Because constraint-related atoms (e.g. $V1 - V2 < 3$ above) are allowed to occur in the body of the rules, in a sense this approach allows feeding the results of solving CSPs back into the ASP computation, allowing for further inference. For example in [9] the authors show an ASP program containing the rules:

$$
\begin{array}{l}
acceptable\_time(T) \leftarrow 10 \leq T \leq 20.\\
acceptable\_time(T) \leftarrow 100 \leq T \leq 120.\\
\\
\neg occurs(A, S) \leftarrow at(S, T), \text{not}\ \ acceptable\_time(T).
\end{array}
$$

where the application of (one of the ground instances of) the last rule depends on the solutions to the constraints in the first two rules.

In practice, though, many problems can be solved by first generating a partial answer using (non-extended) ASP, and then completing the answer by solving a CSP. *It is remarkable that most of the examples of use of $\mathcal{AC}(\mathcal{C})$ from [9] are structured in this way.*

Problems that combine planning and scheduling are particularly good candidates for this approach. In Example 2 from [9] the authors solve such a problem by dividing the program in a planning and a scheduling part. The planning part is written with the usual ASP planning techniques, while the scheduling part introduces the assignment of (wall-clock) times to the steps of the plan and imposes suitable constraints. The problem can be easily solved using our approach by replacing the scheduling part from [9] with the following rules:

$$
\begin{array}{l}
cspdomain(\mathit{fd}).\\
\\
cspvar(at(S), 0, 1440).\\
\\
required(at(S0) \leq at(S1)) \leftarrow\\
\qquad next(S1, S0).\\
\\
required(at(S) - at(0) \leq 60) \leftarrow\\
\qquad goal(S).\\
\\
required(at(S0) - at(S1) \leq -20) \leftarrow\\
\qquad next(S1, S0),\\
\qquad occurs(go\_to(john, home), S0),\\
\qquad holds(at\_loc(john, \mathit{office}), S0).
\end{array}
$$

In fact, there is a whole subclass of $\mathcal{AC}(\mathcal{C})$ programs that can be automatically translated into "equivalent" ASP programs encoding suitable CSPs, as we show next.

Let us consider the class of $\mathcal{AC}_0$ programs without consistency restoring rules (from now on simply called $\mathcal{AC}_0$ programs). We will show that these programs can be translated into ASP programs whose extended answer sets correspond to the answer sets of the original programs.

We start by defining a translation from $\mathcal{AC}_0$ programs to ASP programs. For simplicity, we assume that all mixed predicates in Π have a single constraint parameter (extending to multiple constraint parameters is not difficult). Given a mixed atom $m(t_1, t_2, \ldots, t_{k-1}, t_k)$ where t_i's are terms, we call *functional representation* of the mixed atom the expression $m(t_1, t_2, \ldots, t_{k-1})$. For example, the functional representation of $at(S0, T0)$ is $at(S0)$.

Let Π be an $\mathcal{AC}_0$ program. The ASP-translation, Π', of Π is obtained from Π by:

- Adding a fact $cspdomain(\mathcal{D})$, where $\mathcal{D}$ is an appropriate constraint domain.
- Replacing each declaration $\#mixed\ m(p_1, p_2, \ldots, p_{k-1}, p_k)$ by a rule:

$$cspvar(m(X_1, X_2, \ldots, X_{k-1}), l, u) \leftarrow \\ p_1(X_1), p_2(X_2), \ldots, p_{k-1}(X_{k-1}).$$

  where l and u are the lower and upper bounds of constraint sort p_k and X_i's are variables.[9]
- Replacing each denial in the middle part of Π of the form:

$$\leftarrow \Gamma, c \tag{2}$$

  where c is the constraint literal, by a rule $required(\neg c') \leftarrow \Gamma'$, where c' is obtained by replacing every variable in c by the functional representation of the corresponding mixed atom and Γ' is the set of regular literals from Γ. Notice that $\neg c'$ can be typically simplified by replacing the comparison symbol in c' appropriately. For example, the denial, d_1:

$$\leftarrow goal(S), at(0, T1), at(S, T2), T2 - T1 > 60$$

  is replaced by the rule:

$$required(at(S) - at(0) \leq 60) \leftarrow goal(S).$$

In the rest of the discussion, the expression $c(d)$ will denote the constraint, c, from the body of a denial d of the form (2). For example, given denial d_1 above, $c(d_1)$ is $T2 - T1 > 60$.

Notice that the semantics of $\mathcal{AC}_0$, differently from that of ASP programs, is defined for possibly non-ground programs. In fact, because of the intended use of constraint atoms,

[9] The bounds can be easily extracted from the definition of sort p_k in Π. If the domain does not consist of a single interval, extra constraints can be added to the CSP definition, but we will assume a single interval for simplicity.

one would typically expect variables to be used for the constraint parameters of mixed literals and the corresponding arguments in the constraints. Therefore, *we assume that in Π variables are used for the constraint parameters of mixed predicates*.

We say that a *partial-ground rule* is a rule where the only variables occur as constraint parameters of the mixed predicates and as arguments of the constraint atoms. The *partial grounding* of a rule r is obtained by grounding all the variables of r, except those that occur as constraint parameters of mixed predicates. The set of partial-ground rules obtained this way is denoted by $pground(r)$.

We say that a ground denial d of the form (2) is *constraint-blocked* w.r.t. set of ground literals A if Γ is satisfied by A. The following is an important property of constraint-blocked ground denials.

Proposition 1. *Let Π be an $\mathcal{AC}_0$ program. For every answer set A of Π and every ground mixed-part denial d, if d is constraint-blocked w.r.t. A, then its constraint $c(d)$ is not satisfied.*

We say that a partial-ground denial d of the form (2) is *constraint-blocked* w.r.t. A if some ground instance of d is constraint-blocked w.r.t. A.

Theorem 4. *Every $\mathcal{AC}_0$ program Π can be translated into an ASP program whose extended answer sets are in one-to-one correspondence with the answer sets of Π.*

Proof. *Because of space restrictions, we omit the complete proof. However, we believe that it is useful to show the mapping that the proof is based upon.*

Let Π be an $\mathcal{AC}_0$ program and Π' be its translation, as described above. We define a mapping μ_Π from a set A of literals from the signature of Π into a pair $\langle A', \alpha \rangle$, where A' is a set of literals and α is an association of values to CSP variables. The mapping is defined as follows:

- *For each ground mixed atom $m(t_1, t_2, \ldots, t_{k-1}, t_k)$ from A, α maps the CSP variable denoted by $m(t_1, t_2, \ldots, t_{k-1})$ to value t_k.*
- *$A' \supseteq (A \setminus mixed(\Pi))$.*
- *A' includes an atom $cspdomain(\mathcal{D})$, where $\mathcal{D}$ is an appropriate constraint domain.*
- *For every ground mixed atom $m(t_1, \ldots, t_{k-1}, t_k) \in A$, A' includes an atom $cspvar(m(t_1, \ldots, t_{k-1}), l, u)$ where l and u are the lower and upper bounds of the constraint sort of the last argument of predicate m.*
- *For every denial d from the middle part of Π and every partial-grounding d^* of d that is constraint-blocked w.r.t. A, A' includes the atom $required(\neg c')$, where c' is obtained from the constraint atom c of d^* by replacing every variable in c by the functional representation of the corresponding mixed atom from $body(d^*)$.*

6 Conclusions

In this paper we have shown how ASP and constraint satisfaction can be integrated by viewing ASP as a specification language for constraint satisfaction problems. This approach allows a useful integration of the two paradigms without the need to extend

the language of ASP, without the need for ad-hoc solvers, and with support for global constraints. The last two features seem particularly appealing for the development of industrial-size applications. The paper also contains results showing that an important subclass of $\mathcal{AC}(\mathcal{C})$ programs can be automatically rewritten using our method.

Although space restrictions prevented us from discussing it here, our experiments have also shown that our technique produces programs that are significantly more compact and easy to understand than similar programs written in CLP alone, but with comparable performance. We plan to discuss this further in an extended paper.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP-88. (1988) 1070–1080
2. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing (1991) 365–385
3. Marek, V.W., Truszczynski, M. The Logic Programming Paradigm: a 25-Year Perspective. In: Stable models and an alternative logic programming paradigm. Springer Verlag, Berlin (1999) 375–398
4. Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence (2006)
5. Baral, C., Chancellor, K., Tran, N., Joy, A., Berens, M.: A Knowledge Based Approach for Representing and Reasoning About Cell Signalling Networks. In: Proceedings of the European Conference on Computational Biology, Supplement on Bioinformatics. (2004) 15–22
6. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In Doherty, P., McCarthy, J., Williams, M.A., eds.: International Symposium on Logical Formalization of Commonsense Reasoning. AAAI 2003 Spring Symposium Series (Mar 2003) 9–18
7. Brewka, G., Niemela, I., Syrjanen, T.: Logic Programs wirh Ordered Disjunction. **20**(2) (2004) 335–357
8. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an Integration of Answer Set and Constraint Solving. In: Proceedings of ICLP 2005. (2005)
9. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Annals of Mathematics and Artificial Intelligence (2008) (to appear).
10. Smith, B.M.: 11. Handbook of Constraint Programming. In: Modelling. Elsevier (2006) 377–406
11. Katriel, I., van Hoeve, W.J.: 6. Handbook of Constraint Programming. In: Global Constraints. Elsevier (2006) 169–208
12. Jaffar, J., Lassez, J.L.: Constraint Logic Programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, ACM Press (Jan 1987) 111–119
13. Marriott, K., Stuckey, P.J., Wallace, M.: 12. Handbook of Constraint Programming. In: Constraint Logic Programming. Elsevier (2006) 409–452
14. SICStus: Sicstus Prolog Web Site (2008) http://www.sics.se/isl/sicstuswww/site/.
15. Mellarkod, V.S., Gelfond, M.: Enhancing ASP Systems for Planning with Temporal Constraints. In: LPNMR 2007. (May 2007) 309–314

A Logic of Fixpoint Definitions

Ping Hou and Marc Denecker

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{Ping.Hou,Marc.Denecker}@cs.kuleuven.be

Abstract. We introduce the fixpoint definitions, which is a reformulation of fixpoint logic constructs. We define the logic FO(FD), an extension of first order logic with fixpoint definitions. We illustrate the relation between fixpoint definitions and non-monotone inductive definitions in FO(ID), which is developed as an integration of ASP and classical logic. We investigate the satisfiability problem, SAT(FD), of the propositional fragment of FO(FD). We also demonstrate how to extend existing SAT solvers to become SAT(FD) solvers.

1 Introduction

It is well-known that, in general, inductive definitions cannot be expressed in first order logic (FO). Yet, inductively defined concepts are often useful in practice. The lack of expressive power of FO logic to represent recursion (including recursion through negation) has motivated its extension with fixpoint constructs. For example, the *μ-calculus* [12, 22], a class of temporal logics with explicit fixpoints, provides a conceptually advantageous framework for specifying and reasoning about Real-Time Systems [10]. In the context of databases, query languages have been extended with fixpoint constructs to represent inductively definable concepts [3, 1]. Also, description logics have been extended with such fixpoint constructs [4]. These logics can be viewed as well-behaved fragments of FO logic with fixpoints [18, 19]. The availability of explicit least and greatest fixpoint constructs in these logics allows inductive and coinductive concepts to be expressed in a natural way. Also in these logics, several forms of induction have been modelled: e.g. monotone induction, partial fixpoint induction, inflationary fixpoint induction.

The language FO(ID) [6, 7] is a knowledge representation language that uses the well-founded semantics of logic programming (LP) [23] to extend classical first order logic with a new "inductive definition" primitive. In the resulting formalism, all kinds of definitions regularly found in mathematical practice – e.g., monotone inductive definitions, non-monotone inductive definitions over a well-ordered set, and iterated inductive definitions – can be represented in a uniform way. FO(ID) represents these definitions in the same way as they typically appear in mathematical texts, i.e., as an enumeration of a set of cases in which the defined relation(s) holds; in FO(ID), each of these cases is represented by a rule.

The origins of FO(ID) lie in the area of logic programming. In particular, FO(ID) is very closely related to the Answer Set Programming (ASP) paradigm. The precise relation between FO(ID) and ASP is discussed in detail in [14]. One of the main contributions that FO(ID) provides to this area is to show how a tight integration of logic programming rules into classical logic can be achieved in a conceptually clean way.

The work in this paper is inspired by work on FO(ID) to integrate LP-style rules into fixpoint logic constructs. In this paper, we introduce a concept of *fixpoint definitions (FDs)*, which is a reformulation of fixpoint logic constructs by applying the rule-based *syntactic sugar*. The fixpoint definitions use the format of LP-like rules which will enable us more easily to link it with logic programming notation and the notation for inductive definitions in FO(ID). We define the logic FO(FD), which is an extension of first order logic with fixpoint definitions. In the resulting logic, almost all kinds of inductions can be expressed as well. We investigate the connections between the fixpoint definitions and non-monotone inductive definitions in FO(ID) by presenting equivalence preserving transformations of non-monotone inductive definitions to fixpoint definitions. It turns out that all kinds of inductive definitions in FO(ID) can be expressed in FO(FD). Meanwhile, due to the allowance of greatest fixpoints in FO(FD), co-induction can be represented in FO(FD). Thus, some concepts, e.g., infinite structures and co-recursion [2], which can not be defined in FO(ID) through the well-founded way, can be handled naturally in FO(FD).

On the computational level, the SAT problem, deciding the satisfiability of propositional logic (PC) theories, is a major research topic. An important research direction is to develop SAT solvers for extension of PC, e.g., SMT [17]. The use of extended languages leads to broader applicability of SAT-like systems, facilitates the modelling of applications, and may substantially reduce the size of encodings. All these benefits also hold for PC(FD), the propositional fragment of FO(FD). This paper presents a SAT solver for PC(FD). The satisfiability problem of PC(FD) is called the SAT(FD) problem.

Our approach to the SAT(FD) problem is by encoding the FDs into propositional logic formulae [20]. In this work, we show how SAT(FD) solvers can be built by extending SAT solvers with an additional propagation mechanism suitable for reasoning on fixpoint definitions. The obvious advantage from this approach is that the SAT(FD) solver benefits from any improvement made to the underlying SAT solver. In particular, it has the same performance on pure propositional problems as the underlying solver. A further advantage is that by separating the two propagation mechanisms (one for propositional theories and one for FDs), we also strongly simplify the description of a SAT(FD) solver. According to our knowledge, SAT(FD) is the first model generator for fixpoint logics, which can be viewed as one of the main contributions of this paper.

The paper is organized as follows. In Section 2, we introduce the fixpoint definitions and the logic FO(FD). We present the inductive definitions of FO(ID) and illustrate the relationship between non-monotone inductive definitions and fixpoint definitions in Section 3. We introduce SAT(FD) and present a high level

overview of the requirements for the algorithm of SAT(FD) in Section 4. We present the detailed SAT(FD) algorithm in Section 5. We finish with conclusions, related and further work.

2 Fixpoint Definitions and FO(FD)

In this section, we introduce the fixpoint definitions (FD), which is basically a new formalism of fixpoint constructs using rule-like syntactic sugar. We present the logic FO(FD), which is the extension of first order logic with fixpoint definitions.

We assume familiarity with classical logic. A vocabulary Σ consists of a set of predicate and function symbols. Propositional symbols and constants are 0-ary predicate symbols, respectively function symbols. Terms and FO formulae are defined as usual, and are built inductively from variables, constant and function symbols and logical connectives and quantifiers. Note that predicate symbols occurring in a fixpoint definition are viewed as predicate constants but not predicate variables.

A *rule* is an expression of the form $\forall \bar{x}(P(\bar{x}) \leftarrow \Psi[\bar{x}])$, where $\Psi[\bar{x}]$ is a first order formula. The *defined predicate* of the rule is P. $\Psi[\bar{x}]$ is known as the *body* of the rule. The connective $\leftarrow$ is called *definitional implication* and is to be distinguished from material implication $\supset$. We say that a predicate symbol occurs positively (negatively) in a formula if it occurs in the scope of an even (odd) numbers of negations. A rule is *positive* in a set σ of predicate symbols if these symbols occur only positively in Ψ.

For a set $\mathcal{R}$ of rules, we denote $def(\mathcal{R})$ as the set of defined predicates of its rules, and we denote $open(\mathcal{R})$ as the set of all other symbols occurring in $\mathcal{R}$.

Without loss of generality, we assume from now on, that rule sets contain for each of its defined predicates exactly one rule of the form $\forall \bar{x}(P(\bar{x}) \leftarrow \varphi_P[\bar{x}])$. Indeed, any set of rules $\{\forall \bar{x}(P(\bar{x}) \leftarrow \varphi_1[\bar{x}]), \ldots, \forall \bar{x}(P(\bar{x}) \leftarrow \varphi_n[\bar{x}])\}$ can be transformed into a single rule $\forall \bar{x}(P(\bar{x}) \leftarrow \varphi_1[\bar{x}] \vee \ldots \vee \varphi_n[\bar{x}])$.

Let Σ be a vocabulary, I a Σ-interpretation and $\bar{\sigma}$ a tuple of symbols not necessarily in Σ. $I[\bar{\sigma} : \bar{v}]$ is a $\Sigma \cup \bar{\sigma}$-interpretation, which is the same as I except symbols $\bar{\sigma}$ are interpreted by values $\bar{v}$ within the domain of I. Given a Σ-interpretation I and $\Sigma' \subseteq \Sigma$, the restriction of I to the symbols of Σ' is denoted $I|_{\Sigma'}$. Given a Σ-interpretation I and a Σ'-interpretation I' with $\Sigma \cap \Sigma' = \emptyset$, the $\Sigma \cup \Sigma'$-interpretation mapping each $P \in \Sigma$ to $I(P)$ and each $P' \in \Sigma'$ to $I'(P')$ is denoted by $I + I'$.

With a set of rules $\mathcal{R}$ and a Σ-interpretation I interpreting at least all open symbols and no defined symbols ($\Sigma \cap def(\mathcal{R}) = \emptyset$ and $open(\mathcal{R}) \subseteq \Sigma$), there is a standard way of associating an operator $\Gamma_I^{\mathcal{R}}$ on the set of $def(\mathcal{R})$-interpretations with the domain of I. For two such interpretations J, K, we define $\Gamma_I^{\mathcal{R}}(J) = K$ if for every $P \in def(\mathcal{R})$, $K(P) = \{\bar{d} | I + J \models \varphi_P[\bar{d}]\}$.

If each defined symbol in $def(\mathcal{R})$ has only positive occurrences in the body of a rule in $\mathcal{R}$, the operator $\Gamma_I^{\mathcal{R}}$ is monotone with respect to the standard truth order on interpretations and hence, it has least and greatest fixpoints in this set

denoted $lfp(\Gamma_I^{\mathcal{R}})$, respectively $gfp(\Gamma_I^{\mathcal{R}})$. Importantly, if $I(P) \leq I'(P)$ for every symbol $P \in open(\mathcal{R})$ with only positive occurrences in rule bodies of $\mathcal{R}$, then $lfp(\Gamma_I^{\mathcal{R}}) \leq lfp(\Gamma_{I'}^{\mathcal{R}})$ and $gfp(\Gamma_I^{\mathcal{R}}) \leq gfp(\Gamma_{I'}^{\mathcal{R}})$.

Definition 1. *We define a* least fixpoint definition *(LFD), respectively* greatest fixpoint definition *(GFD) over vocabulary Σ by simultaneous induction, as a finite expression $\mathcal{D}$ of the form*

$$\left\lfloor \mathcal{R}, \Delta_1, \ldots, \Delta_m, \nabla_1, \ldots, \nabla_n \right\rfloor, \text{ respectively } \left\lceil \mathcal{R}, \Delta_1, \ldots, \Delta_m, \nabla_1, \ldots, \nabla_n \right\rceil$$

with $0 \leq n, m$ such that:

1. *$\mathcal{R}$ is a set of rules over Σ.*
2. *Each Δ_i is a least fixpoint definition and each ∇_j is a greatest fixpoint definition.*

To express the remaining conditions, we need some auxiliary concepts and notations. For such an expression $\mathcal{D}$, we say that a predicate P is locally defined *in $\mathcal{D}$ if $P \in def(\mathcal{R})$, and that P is* defined *in $\mathcal{D}$ if P is locally defined in $\mathcal{D}$ or defined in any of its subdefinitions $\Delta_1, \ldots, \nabla_n$. The set of defined predicates of $\mathcal{D}$ is denoted $def(\mathcal{D})$. A symbol is* open *in $\mathcal{D}$ if it occurs in $\mathcal{D}$ and is not defined in it. The set of open symbols of $\mathcal{D}$ is denoted $open(\mathcal{D})$.*

3. *Every defined symbol of $\mathcal{D}$ has only positive occurrences in bodies of rules in $\mathcal{D}$.*
4. *Each symbol $P \in def(\mathcal{D})$ has exactly one local definition in $\mathcal{D}$. Formally, $\{def(\mathcal{R}), def(\Delta_1), \ldots, def(\nabla_n)\}$ is a partition of $def(\mathcal{D})$.*
5. *For every subdefinition $\mathcal{D}'$ of $\mathcal{D}$, $open(\mathcal{D}') \subseteq open(\mathcal{D}) \cup def(\mathcal{R})$. In particular, a symbol defined in another subdefinition $\mathcal{D}'' \neq \mathcal{D}'$, does not occur in $\mathcal{D}'$.*

A fixpoint definition *is either a least fixpoint definition or a greatest fixpoint definition.* □

Both LFD and GFD expressions are defined as trees with nodes containing rule sets and two types of children, namely LFD's and GFD's. Each defined predicate of the expression is locally defined in exactly one node of this graph. Moreover, a predicate locally defined in one node, has only positive occurrences in rule bodies in the node, its ancestors and its descendants, and does not occur at all in a sibling of the node or the siblings descendants.

Example 1. The following expression is a fixpoint definition of the sets of even and odd numbers on the structure of the natural numbers with zero and the successor function: $\left\lfloor \begin{array}{l} \forall x(E(x) \leftarrow x = 0 \vee \exists y(x = s(y) \wedge O(y))) \\ \forall x(O(x) \leftarrow \exists y(x = s(y) \wedge E(y))) \end{array} \right\rfloor$

Example 2. We represent a transition graph on a set of vertices representing states by a binary predicate T. Let R represent a property of states, i.e., it is a unary predicate on vertices. We want P to represent the set of states which have an (infinite) path which passes an infinite number of times through a state satisfying R. This set is defined by: $\left\lceil \begin{array}{l} \forall x\ (P(x) \leftarrow Q(x)) \\ \left\lfloor \begin{array}{l} \forall x\ (Q(x) \leftarrow R(x) \wedge \exists y(T(x,y) \wedge P(y))) \\ \forall x(Q(x) \leftarrow \exists y(T(x,y) \wedge Q(y))) \end{array} \right\rfloor \end{array} \right\rceil$.

An FO(FD) formula is either an FO formula or a fixpoint definition. An FO(FD) theory is a set of FO(FD) formulae without free variables.

The semantics of the FO(FD) is an integration of standard FO semantics with fixpoint semantics of definitions.

We firstly define the semantics of LFD's and GFD's. Given an expression $\mathcal{D}$ which might be a LFD or an GFD, and a Σ_o-interpretation I interpreting at least all open symbols of $\mathcal{D}$ and no defined ones. We define an operator $\Gamma_I^{\mathcal{D}}$ on the set of $def(\mathcal{D})$-interpretations with domain U_I. This operator is monotone with respect to the standard truth order on interpretations and hence, it has least and greatest fixpoints in this set. We define $\Gamma_I^{\mathcal{D}}(J)$ inductively as the interpretation $K + K'$ where

- K is the $(def(\mathcal{D}) \setminus def(\mathcal{R}))$-interpretation such that, for $J' = I + J|_{def(\mathcal{R})}$:

$$K|_{def(\Delta_i)} = lfp(\Gamma_{J'}^{\Delta_i})$$
$$K|_{def(\nabla_j)} = gfp(\Gamma_{J'}^{\nabla_j})$$

  Observe that J' interprets all open symbols in every subdefinition of φ.
- K' is the $def(\mathcal{R})$-interpretation $\Gamma_K^{\mathcal{R}}(J|_{def(\mathcal{R})})$.

Let $\mathcal{D}$ be a fixpoint definition and I a Σ-interpretation such that Σ contains all symbols in $\mathcal{D}$. If $\mathcal{D}$ is a LFD, then I satisfies $\mathcal{D}$ iff $I|_{def(\mathcal{D})} = lfp(\Gamma^{\mathcal{D}}_{I|_{open(\mathcal{D})}})$. If $\mathcal{D}$ is a GFD, then I satisfies $\mathcal{D}$ iff $I|_{def(\mathcal{D})} = gfp(\Gamma^{\mathcal{D}}_{I|_{open(\mathcal{D})}})$. As usual, this is denoted $I \models \mathcal{D}$. A Σ-interpretation I satisfies an FO(FD) theory T if I satisfies every $\varphi \in T$.

Example 3. In Example 2, the i'th iteration of the operator $\mathcal{D}$ computes P as the set of states with a path containing at least i states satisfying R. The fixpoint consists of nodes with a path containing infinitely many states satisfying R.

3 Generalized Inductive Definitions

In this section, we present the notation for inductive definitions in the logic FO(ID) [6, 7]. We also illustrate the relationship between fixpoint definitions and non-monotone inductive definitions.

Definition 2. *A* (generalized) inductive definition (GID) *is a finite set of rules. Its sets of defined symbols def(D), respectively open symbols open(D) are defined as usual.*

We do not insist on defined predicate to occur positively in rule bodies in a general inductive definition, meaning that non-monotone inductive definitions are allowed. A model of a GID is a two-valued well-founded model [8, 7].

Example 4. Consider the following non-monotone inductive definition of even and odd numbers over the structure of the natural numbers with zero and the successor function:

$$\left\{ \begin{array}{l} \forall x(Even(x) \leftarrow x = 0 \vee \exists y(x = s(y) \wedge \neg Even(y))) \\ \forall x(Odd(x) \leftarrow \exists y(x = s(y) \wedge Even(y))) \end{array} \right\}.$$

Indeed, an arbitrary (non-monotone) inductive definition can be transformed into an equivalent (up to the original vocabulary) least fixpoint definition. We demonstrate it as follows.

Let D be a generalized inductive definition. For each of its defined predicates P, we introduce a new predicate symbol of the same arity $P^\neg$. For each formula φ, let $\overline{\varphi}$ denote the formula obtained by substituting each negative occurrence $P(\bar{t})$ of a defined predicate P by $\neg P^\neg(\bar{t})$. Using this concept, define two sets of rules, $\mathcal{R}_D = \{\forall \bar{x}(P(\bar{x}) \leftarrow \overline{\varphi_P}) | P \in \mathit{def}(D)\}$ and $\mathcal{R}_D^d = \{\forall \bar{x}(P^\neg(\bar{x}) \leftarrow \overline{\neg\varphi_P}) | P \in \mathit{def}(D)\}$. Now define $\Delta_D = \left\lfloor \mathcal{R}_D, \left\lceil \mathcal{R}_D^d \right\rceil \right\rfloor$. Note that this is a well-formed expression.

Example 5.

$$\left\{ \begin{array}{l} \forall x(Even(x) \leftarrow x = 0 \vee \exists y(x = s(y) \wedge \neg Even(y))) \\ \forall x(Odd(x) \leftarrow \exists y(x = s(y) \wedge Even(y))) \end{array} \right\}$$

translates into

$$\left\lfloor \begin{array}{l} \forall x(Even(x) \leftarrow x = 0 \vee \exists y(x = s(y) \wedge Even^\neg(y))) \\ \forall x(Odd(x) \leftarrow \exists y(x = s(y) \wedge Even(y))) \\ \left\lceil \begin{array}{l} \forall x(Even^\neg(x) \leftarrow x \neq 0 \wedge \forall y(x = s(y) \supset Even(y))) \\ \forall x(Odd^\neg(x) \leftarrow \forall y(x = s(y) \supset Even^\neg(y))) \end{array} \right\rceil \end{array} \right\rfloor .$$

Theorem 1. *Let D be a generalized inductive definition over Σ. Let Δ_D be a fixpoint definition over $\Sigma \cup \{P^\neg \mid P \in \mathit{def}(D)\}$ obtained from D by the translation mentioned above. Then there is a one-to-one mapping between the models I of D and the models I' of Δ_D with $I'|_\Sigma = I$ and $I'(P^\neg) = \neg I(P)$ for every $P^\neg$.*

4 Theory of SAT(FD)

4.1 PC(FD) and SAT(FD)

In this section, we introduce PC(FD), the propositional fragment of FO(FD), and SAT(FD), the satisfiability problem of PC(FD). We assume familiarity with propositional logic.

A propositional vocabulary Σ is a set of propositional atoms. A literal is an atom p or its negation $\neg p$. An atom p is called a *positive* literal, $\neg p$ a negative one. For a literal l, we identify $\neg\neg l$ with l. For a set S of literals, we denote by $\neg S$ the set $\{\neg l \mid l \in S\}$, and by $\widehat{S}$ the set $S \cup \neg S$.

A propositional logic theory is a set of propositional formulae. Without loss of generality, we assume from now on, that propositional logic theories are in conjunctive normal form (CNF): all propositional formulae are disjunctions of literals, called *clauses*. A propositional logic theory T is satisfiable if it has a model: an interpretation I that satisfies every clause of T, denoted $I \models T$. SAT is the problem of deciding whether a given theory is satisfiable.

A propositional fixpoint definition is a fixpoint definition such that all symbols occurring in it are propositional symbols.

Example 6. Consider the propositional fixpoint definition $\mathcal{D} = \left[\begin{array}{l} p \leftarrow q \vee r \\ q \leftarrow p \\ \left[\begin{array}{l} r \leftarrow p \\ s \leftarrow t \vee a \\ t \leftarrow s \end{array}\right] \end{array}\right]$.
It is obvious that a is the only open atom in this fixpoint definition. There are only two interpretations satisfying $\mathcal{D}$, namely, $I_1 = \{a \mapsto \mathbf{f}, p \mapsto \mathbf{f}, q \mapsto \mathbf{f}, r \mapsto \mathbf{f}, s \mapsto \mathbf{t}, t \mapsto \mathbf{t}\}$ and $I_2 = \{a \mapsto \mathbf{t}, p \mapsto \mathbf{f}, q \mapsto \mathbf{f}, r \mapsto \mathbf{f}, s \mapsto \mathbf{t}, t \mapsto \mathbf{t}\}$. The construction of I_1 is illustrated as follows: $I_1^1 = \{a \mapsto \mathbf{f}, p \mapsto \mathbf{f}, q \mapsto \mathbf{f}, r \mapsto \mathbf{t}, s \mapsto \mathbf{t}, t \mapsto \mathbf{t}\}$, $I_1^2 = \{a \mapsto \mathbf{f}, p \mapsto \mathbf{f}, q \mapsto \mathbf{f}, r \mapsto \mathbf{f}, s \mapsto \mathbf{t}, t \mapsto \mathbf{t}\}$, which is the limit of the iterations and thus, $I_1 = I_1^2$.

A *PC(FD) theory* is a set of propositional formulae and propositional fixpoint definitions. A propositional fixpoint definition $\mathcal{D}$ is in *definitional normal form* (DefNF) if for any $p \in \Sigma$, the fixpoint definition contains at most one rule $p \leftarrow \varphi_p$, and either $\varphi_p = \bigvee B_p$ or $\varphi_p = \bigwedge B_p$, where B_p is a set of literals called the *body literals*. A PC(FD) theory is in DefNF if it contains only one propositional fixpoint definition, which is in DefNF, and its set of formulae is in conjunctive normal form (CNF). There exists a linear transformation from an arbitrary PC(FD) theory T over Σ to a DefNF theory T' over $\Sigma' \supset \Sigma$ such that there is a one-to-one correspondence between models M of T and models M' of T'. Hence without loss of generality, we can from now on assume that PC(FD) theories are in DefNF.

An interpretation I satisfies a PC(FD) theory if it satisfies every formula and every definition of the theory. The SAT(FD) problem is the satisfiability problem for PC(FD) theories.

SAT solving is the practice of answering the SAT problem. Although other solving techniques exist, the current state of the art solvers are based on DPLL [5] augmented with the two watched literal scheme (2WL) and with clause learning [16, 21]. The propagation method that drives this search is *unit propagation*, whereby a literal is interpreted true if it occurs as the only non-false literal in a clause. For more details we refer the reader to [16].

Definition 3. *The* completion *of a rule $p \leftarrow \varphi_p$ is given by the clausal form of $p \equiv \varphi_p$. The* completion *of a propositional fixpoint definition $\mathcal{D}$, denoted by $comp(\mathcal{D})$, is the union of the completions of all rules in $\mathcal{D}$.*

An important property, though, is that $I \models \mathcal{D}$ implies $I \models comp(\mathcal{D})$. The converse is not true, $\mathcal{D}$ has fewer models than $comp(\mathcal{D})$. Hence, $\mathcal{D}$ can cause extra constraints that result in more propagations than in $comp(\mathcal{D})$. Our work is to extend a SAT solver with these propagations.

4.2 Justifications

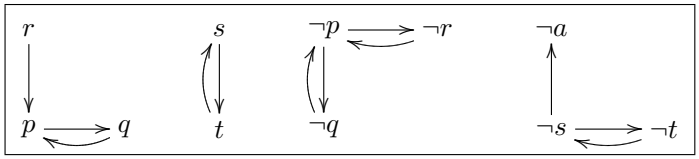
In this section, we introduce the notion of a *justification*, and use it to provide an alternative characterization of the semantics of fixpoint definitions. We then

return to the satisfiability for PC(FD), following an approach suggested by this new characterization.

For a directed graph $G = (V, E)$ and an element $v \in V$, we denote by $Ch_G(v)$ the set $\{w \mid (v, w) \in E\}$. If V is a set of literals, we call a cycle in G *positive* or *negative* if it contains respectively only positive or negative literals.

Definition 4 (Justification). *A justification J for $\mathcal{D}$ is a directed graph $(\widehat{\Sigma}, E)$ such that: (a) for every conjunctive rule $c \leftarrow c_1 \wedge \ldots \wedge c_n \in \mathcal{D}$, $Ch_J(c) = \{c_1, \ldots, c_n\}$ and $Ch_J(\neg c) = \{\neg c_i\}$ for some $i \in [1, n]$, and (b) for every disjunctive rule $d \leftarrow d_1 \vee \ldots \vee d_n \in \mathcal{D}$, $Ch_J(\neg d) = \{\neg d_1, \ldots, \neg d_n\}$ and $Ch_J(d) = \{d_i\}$ for some $i \in [1, n]$, and (c) for every $l \notin \widehat{def(\mathcal{D})}$, $Ch_J(l)$ is empty. The* sub-justification *of J starting in a literal l, denoted $Sub(J, l)$, is the subgraph of J consisting of all paths starting in l.*

Example 7. Consider Example 6. For the disjunctive rule defining p one can choose $Ch_J(p)$ as either q or r. Similarly $Ch_J(s)$ is either t or a. This results in four possible justifications for $\mathcal{D}$, one of which is shown as follows.



Let J be a justification of $\mathcal{D}$. It is easy to see that for each cycle in J, there exists a unique subdefinition $\mathcal{D}'$ of $\mathcal{D}$ (possibly $\mathcal{D}' = \mathcal{D}$) such that all literals occurring in the cycle are defined in $\mathcal{D}'$ and at least one literal in the cycle is locally defined in $\mathcal{D}'$. We call such a cycle *$\mathcal{D}'$-cycle*. Note a cycle in a justification of $\mathcal{D}$ is either positive or negative, due to the positivity of rule bodies. A path in J is called a *$\mathcal{D}'$-path* if all literals occurring in the path are defined in $\mathcal{D}'$ and at least one literal in the path is locally defined in $\mathcal{D}'$.

To formalize the notion that a defined atom is assigned a truth value in accordance with a justification, we introduce the notion of support.

Definition 5 (Support). *Let J be a justification for the fixpoint definition $\mathcal{D}$ and I a Σ-interpretation. Then J* supports *I if for each $l \in \widehat{def(\mathcal{D})}$, $I(l) = I(\bigwedge Ch_J(l))$.*

Definition 6 (Witness). *Let J be a justification for $\mathcal{D}$ and I a Σ-interpretation. Then J is a* $\mathcal{D}$-witness for *I iff (a) J supports I, and (b) for each LFD subdefinition $\mathcal{D}'$ of $\mathcal{D}$, J contains no positive $\mathcal{D}'$-cycle, and for each GFD subdefinition $\mathcal{D}'$ of $\mathcal{D}$, J contains no negative $\mathcal{D}'$-cycle. The second condition is called J is* cycle-safe.

Define $\circlearrowright = \{p \mid$ either p is locally defined in an LFD subdefinition $\mathcal{D}'$ of $\mathcal{D}$ and for any justification J for D, $Sub(J, p)$ contains a positive $\mathcal{D}'$-cycle or p is locally defined in a GFD subdefinition $\mathcal{D}'$ of $\mathcal{D}$ and for any justification

J for D, $Sub(J, \neg p)$ contains a negative $\mathcal{D}'$-cycle }. Those atoms in $\circlearrowleft$ which are locally defined in LFD subdefinitions, respectively locally defined in GFD subdefinitions, have to be false, respectively, true in any model. Defining $\mathcal{D}^{\emptyset}$ as the fixpoint definition obtained by replacing in $\mathcal{D}$ the rule $p \leftarrow \varphi_p$ by $p \leftarrow \bot$ for each $p \in\circlearrowleft$ which is locally defined in some LFD subdefinition and the rule $p \leftarrow \varphi_p$ by $p \leftarrow \top$ for each $p \in\circlearrowleft$ which is locally defined in some GFD subdefinition.

Example 8. Consider Example 6. Then p, q are locally defined in the least fixpoint definition $\mathcal{D}$ and p, q are part of a positive $\mathcal{D}$-cycle in any justification while s, t are locally defined in a GFD subdefinition $\mathcal{D}'$ of $\mathcal{D}$ and $\neg s, \neg t$ are part of a negative $\mathcal{D}'$-cycle in any justification. Hence, $\mathcal{D}^{\emptyset} = \left| \begin{array}{l} p \leftarrow \bot \\ q \leftarrow \bot \\ \left\lceil \begin{array}{l} r \leftarrow p \\ s \leftarrow \top \\ t \leftarrow \top \end{array} \right\rceil \end{array} \right|$.

The following proposition follows easily:

Proposition 1. $\mathcal{D} \equiv \mathcal{D}^{\emptyset}$.

A witness for an interpretation I reflects a reasoning for the truths in I and the following theorem holds:

Theorem 2. *Let I be a Σ-interpretation and $\mathcal{D}$ a fixpoint definition on Σ. $I \models \mathcal{D}$ iff there exists a $\mathcal{D}^{\emptyset}$-witness for I.*

Example 9. Continuing from Examples 6, 7, 8, we search a model I of $\mathcal{D}$ that extends $I' = \{a \mapsto \mathbf{t}\}$. The only justification for $\mathcal{D}^{\emptyset}$ does not contain any cycles, hence it can be a $\mathcal{D}^{\emptyset}$-witness for I. For the justification to be a $\mathcal{D}^{\emptyset}$-witness of I, it should support I, hence we have that $I(p) = I(q) = I(r) = \mathbf{f}$ and $I(s) = I(t) = \mathbf{t}$, i.e., $I = I_2$.

In other words, if we have a partial interpretation I', we can strengthen it (propagate) by making p, q, r false and s, t true, based on the above reasoning.

The above theorem suggests the following structure for a SAT(FD) algorithm:

- initialize to find $\mathcal{D}^{\emptyset}$,
- apply SAT on $comp(\mathcal{D}^{\emptyset})$ (and on the propositional part of the theory),
- maintain a witness for the interpretation found by the SAT solver.

5 SAT(FD) Algorithm

The algorithm of SAT(FD) presented here is based on the algorithm of SAT(ID) solver from [15]. However, due to the difference between the justification semantics of fixpoint definitions and that of general inductive definitions, the algorithm of SAT(FD) varies considerably.

The purpose of the initialisation step is to identify the atoms of $\circlearrowleft$ and to construct $\mathcal{D}^{\emptyset}$. $\circlearrowleft$ can be easily obtained by marking literals with a stratification

level. A cycle-safe justification of $\mathcal{D}^{\emptyset}$ can now be derived by assigning to $J(d)$ an atom in the body with a smaller stratification level for each disjunctively defined atom d which is locally defined in an LFD subdefinition and to $J(\neg c)$ a negative literal in the body for $\neg c$ with a smaller stratification level for each $\neg c$ that is the negation of a conjunctively defined atom c and is locally defined in a GFD subdefinition.

Recall from Section 4 that the goal is to apply SAT on $comp(\mathcal{D}^{\emptyset})$, and to maintain during the search a witness for the current interpretation. We start with a high level description of the procedure and then elaborate on the details.

The main procedure iterates over the following steps until either a solution is found (a total interpretation and a witness for it) or the search space is exhausted.

1. Select a cycle-safe justification J_{cs}.
2. Use the SAT solver to update the current interpretation by performing unit propagation on $comp(\mathcal{D}^{\emptyset})$ (and on the propositional part of the theory).
3. Use the state of SAT solver to construct a supporting justification J_s.
4. If J_s is not a witness then:
   - **4a** Use J_{cs} to adjust J_s so that it becomes a witness. In case this fails, a set $Cycle$ is obtained of defined literals that cannot have a witness under the current interpretation. $Cycle$ consists of two kinds of defined literals, namely positive defined atoms which are locally defined in LFD subdefinitions and negative defined literals which are locally defined in GFD subdefinitions. In every supporting justification it holds that for each $p \in Cycle$ such that p is locally defined in an LFD subdefinition $\mathcal{D}'$, the subjustification of p has a positive $\mathcal{D}'$-cycle, and for each $\neg p \in Cycle$ such that $\neg p$ is locally defined in a GFD subdefinition $\mathcal{D}'$, the subjustification of $\neg p$ has a negative $\mathcal{D}'$-cycle.
   - **4b** If J_s is still not a witness, make sure that all positive atoms in $Cycle$ which are locally defined in LFD subdefinitions will be set to be false and all negative literals which are locally defined in GFD subdefinitions will be set to be false as well in the next iteration.

Step 1: Selection of a cycle-safe justification. In the first iteration, the cycle-safe justification J_{cs} is constructed with the procedure described in the beginning of this section. In later iterations, the most recent witness is used as cycle-safe justification, i.e., the supporting justification J_s of the previous iteration becomes the cycle-safe justification when it is a witness.

Step 2: SAT solving. This is a propagation step by the underlying SAT-solver. Note that this includes clause learning and backtracking when propagation leads to the detection of a conflict.

Step 3: Construct a Supporting Justification. We assume here that SAT solver we are extending implements unit propagation using 2WL scheme. This scheme keeps clauses satisfiable by maintaining for each clause an invariant on its two

watched literals. Let $W_1(c)$ and $W_2(c)$ be the watched literals of clause c and I the current interpretation, then the invariant is as follows: either $I(W_1(c)) = \mathbf{t}$ or $W_1(c)$ is not interpreted in I and $I(W_2(c)) \neq \mathbf{f}$.

We use the watching literals to construct a supporting justification J_s. This requires to set $J_s(d)$ for every disjunctively defined atom d and to set $J_s(\neg c)$ for every $\neg c$ that is the negation of a conjunctively defined atom c. Let $d \leftarrow d_1 \vee \ldots \vee d_n$ be the rule defining d. $comp(\mathcal{D})$ contains the clause $\neg d \vee d_1 \vee \ldots \vee d_n$; let W_1 and W_2 be the watched literals of this clause. If $W_1 = \neg d$ then $J_s(d) = W_2$, otherwise $J_s(d) = W_1$. Let $c \leftarrow c_1 \wedge \ldots \wedge c_n$ be the rule defining c. $comp(\mathcal{D})$ contains the clause $c \vee \neg c_1 \vee \ldots \vee \neg c_n$; let W_1 and W_2 be the watched literals of this clause. If $W_1 = c$, then $J_s(\neg c) = W_2$, otherwise $J_s(\neg c) = W_1$. Knowing that the current interpretation is the result of a propagation step, one can easily verify that J_s is a supporting justification.

Step 4a: Find a witness. The supporting justification can have positive $\mathcal{D}'$-cycles where $\mathcal{D}'$ is an LFD subdefinition and negative $\mathcal{D}'$-cycles where $\mathcal{D}'$ is a GFD subdefinition. Because J_{cs} is cycle-safe, it must be the case that each positive $\mathcal{D}'$-cycle where $\mathcal{D}'$ is an LFD subdefinition contains at least one disjunctively defined atom d with $J_s(d) \neq J_{cs}(d)$ and each negative $\mathcal{D}'$-cycle where $\mathcal{D}'$ is a GFD subdefinition contains at least one $\neg c$, which is the negation of a conjunctively defined atom c, with $J_s(\neg c) \neq J_{cs}(\neg c)$. Let us call such literals *cycle sources*. Cycle sources belong to the set of literals on which both justifications disagree: $DS = \{l \mid J_s(l) \neq J_{cs}(l)\}$. The overall strategy is to check for each element cs in DS whether it is a cycle source, and if so, to perform local adjustments on the supporting justification so that cs is no longer part of a positive $\mathcal{D}'$-cycle where $\mathcal{D}'$ is an LFD subdefinition or part of a negative $\mathcal{D}'$-cycle where $\mathcal{D}'$ is a GFD subdefinition ("justifying cs"). Obviously, the smaller DS, the less work this step requires.

The further processing then consists of justifying each element cs in DS until either DS is empty and hence J_s is a witness of the current interpretation or some cs could not be justified, in which case a set $Cycle$ as described in the high level algorithm is returned.

Analyse(cs) (We omit the case that cs is a disjunctively defined atom due to the space restrictions. Actually it is quite similar to the case that cs is the negation of a conjunctively defined atom, which will be presented as follows.)

The negative defined literal cs, which is the negation of a conjunctively defined atom in the definition, is not false in the current interpretation and possibly belongs to a negative $\mathcal{D}'$-cycle in the supporting justification J_s where $\mathcal{D}'$ is a GFD subdefinition.

In an initialisation step, the procedure marks all negative literals as *unsafe* that are defined in some GFD subdefinition $\mathcal{D}'$ and are on a $\mathcal{D}'$-path in J_s that leads to cs. This means that all negative literals that belong to an eventual negative $\mathcal{D}'$-cycle are marked as unsafe; however, also other negative literals can be marked as unsafe. If $J_s(cs)$ is not marked, then there is no negative $\mathcal{D}'$-cycle

passing through cs and we are done. Otherwise, cs is a cycle source and J_s has to be adjusted.

By "to justify the negation $\neg c$ of a conjunctively defined atom c" we mean setting $J_s(\neg c)$ such that $\neg c$ is no longer part of a negative $\mathcal{D}'$-cycle through cs; "to justify the negation $\neg d$ of a disjunctively defined atom d" means showing that negations of all body literals of d are justified. The purpose of the algorithm is to justify cs.

The negation $\neg c$ of a conjunctively defined atom c can be justified either by setting $J_s(\neg c)$ to a literal that is not marked unsafe, or by setting it to a negative literal that in turn can be similarly justified. To this end, a working queue $\mathcal{Q}$, initialised with cs, is maintained: $\mathcal{Q}$ contains literals that can still be tried to be justified.

The algorithm also maintains a set $Cycle$, initialised with cs, of literals that are waiting to be justified. If the algorithm fails, then the elements of $Cycle$ have not been justified.

Literals are popped from $\mathcal{Q}$ and processed, until either it is ensured that cs is no longer part of a negative $\mathcal{D}'$-cycle or $\mathcal{Q}$ is empty, in which case $Cycle$ is returned. Let $\neg c$ be the popped element. If it is no longer marked as unsafe, it has already been justified and the next element can be popped. Otherwise, two cases are distinguished. They rely on a procedure Justify($\neg q$) described afterwards.

$\neg c$ is the negation of the conjunctively defined atom c. Let B_c be the body of the defining rule for c. If $\neg B_c$ has a negative literal $\neg b$ that is neither marked nor false, then set $J_s(\neg c) = \neg b$ ($\neg c$ is not false under the current interpretation, hence, to preserve support, $\neg b$ has to be non false as well) and perform Justify($\neg c$). Note that $\neg c$ is now justified: $\neg b$ is not marked and hence has no $\mathcal{D}'$-path to cs. Furthermore, it has no $\mathcal{D}'$-path to $\neg c$ either, because all negative literals with a $\mathcal{D}'$-path to $\neg c$ in J_s are marked. Therefore adding the edge $\neg c \rightarrow \neg b$ cannot create a new negative $\mathcal{D}'$-cycle.
If all non false literals in $\neg B_c$ are marked (they are negative literals as positive atoms cannot be marked), the ones that are not yet in $Cycle$ are all pushed on $\mathcal{Q}$ and added to $Cycle$: justifying any of them suffices to justify $\neg c$ (Justify will take care of doing that).

$\neg c$ is the negation of the disjunctively defined atom c. Let B_c be the body of the defined rule for c. If $\neg B_c$ has no marked literal, $\neg c$ is justified (not part of a negative $\mathcal{D}'$-cycle through cs), so Justify($\neg c$) is performed. Otherwise, a marked negative defined literal $\neg q$ is selected from $\neg B_c$ (preferably one already in $Cycle$) and, if not yet in $Cycle$, added to it and pushed on $\mathcal{Q}$. The negative literal $\neg q$ is called the *guard* of $\neg c$. Adding only this guard to $\mathcal{Q}$ (or none if already in $Cycle$), instead of all marked negative body literals, has the advantage that no computation time will be lost on other negative body literals in case this guard cannot be justified. In case it can, Justify($\neg q$) will add $\neg c$ to $\mathcal{Q}$ again for reconsideration.

Justify($\neg q$) The "unsafe" mark is removed from the negative literal $\neg q$ and the negative literal is removed from $Cycle$. Moreover, if it is an element from DS

then it can be removed from DS as well, as it can no longer be a cycle source. Finally, if it is cs itself, we are done and can start with processing the next element in DS. If it is not cs, we have to continue:

- For every $\neg c \in Cycle$, which is the negation of a conjunctively defined atom c, with $\neg q \in \neg B_c$, set $J_s(\neg c) = \neg q$ and perform Justify($\neg c$). Indeed, if $\neg q$ is no longer part of the negative $\mathcal{D}'$-cycle through cs, then so is $\neg c$ in the changed J_s.
- For every $\neg c$ that is the negation of a disjunctively defined atom c and has $\neg q$ as guard, $\neg c$ is pushed on $\mathcal{Q}$.

If $\mathcal{Q}$ becomes empty before cs could be justified, some negative literals (at least cs itself) are still in $Cycle$, and all possible supporting subjustifications for them have been exhaustively searched; none has been found that does not have negative $\mathcal{D}'$-cycle through cs. These implies that these negative literals cannot have a witness, hence it is correct to add the learning clauses in step 4b.

Step 4b: Learning clauses from the Cycle-set . When the supporting justification cannot be adjusted into a cycle-safe supporting justification, the set $Cycle$ of defined literals that cannot have a witness subjustification is returned. The positive atoms in the set that are locally defined in an LFD subdefinition have to be set false in the current interpretation and the negative literals in the set that are locally defined in a GFD subdefinition have to be set false as well in the current interpretation. To properly integrate this with the SAT solver and its backtracking search, this is achieved by extending the theory with an appropriate learned clause for each of these literals.

Define $Ante = (\bigcup_{d \in Cycle, d \text{ positive}, d \text{ disjunctively defined}} B_d \cup \bigcup_{\neg c \in Cycle, \neg c \text{ negation of } c, c \text{ conjunctively defined}} \neg B_c) \setminus Cycle$. The falsity of those literals forces the falsity of the literals in the cycle set. A so-called *loop formula* $\bigvee Ante \vee \neg(\bigvee Cycle)$ captures this (adapted from [13]); its CNF contains one reason clause for each literal in $Cycle$.

Overview of the main procedure of the algorithm. To a standard contemporary DPLL-based SAT solver, we add a phase of "definitional propagations", to be executed after the unit propagations on the clauses of the theory as well as on $comp(\mathcal{D}^{\emptyset})$. For this phase we introduce a new data structure: a justification J that is certainly cycle-safe in $\mathcal{D}^{\emptyset}$.

The phase starts by finding a supporting justification J'; in case of a 2WL implementation J_W is used for this. The set of cycle sources CS is then initialized: the literals on which J and J' differ. The Analyse algorithm is then applied on literals $cs \in CS$, one by one, whereby J' changes, and CS may be decreased. If the algorithm terminates with a non-empty set $Cycle$, the definitional propagation phase adds the corresponding loop formulae and then terminates, after which standard unit propagations are resumed. Otherwise, CS will eventually be emptied, and we can then set $J := J'$: this justification is both cycle-safe and supporting. Also, W can be adapted to J' such that $J_W = J'$, so that on later occasions the set of cycle sources will likely be smaller.

6 Conclusions and Related Work

In this paper, we introduced the fixpoint definitions, which is an alternative expression of fixpoint constructs, and the logic FO(FD), which is an extension of first order logic with fixpoint definitions. We investigated the correspondence between fixpoint definitions and non-monotone inductive definitions in FO(ID), which is a knowledge representation language integrating ASP-style logic programming and classical logic. We presented PC(FD), which is the propositional fragment of FO(FD), and demonstrated how to extend existing SAT solvers to become SAT(FD) solvers.

Related work is provided by Gupta et al. in [11]. They introduced coinduction, which corresponds to the greatest fixpoint constructor, into logic programming to obtain the coinductive logic programming. They discussed applications of coinductive logic programming into programming verification, model checking and non-monotonic reasoning, in particular its manifestation as answer set programming. However, in the coinductive logic programming, naively mixing coninduction and induction leads to contradictions. This contradiction is resolved by stratifying inductive and coinductive predicates in a program. Indeed, arbitrary cyclical nesting of least and greatest fixpoint expressions is allowed in FO(FD).

We are currently implementing the algorithm of SAT(FD) as an extension to the SAT(ID) solver MiniSat(ID) [15], which is a SAT solver for PC(ID), the propositional fragment of FO(ID). MiniSat(ID) is built based on the popular SAT solver MiniSat [9]. The comparison between MiniSat(ID) and the best ASP solvers is discussed in detail in [15]. We intend to integrate the SAT solver for PC(FD) into MiniSat(ID) and the resulting solver MiniSat(ID,FD) will hopefully exhibit the expected benefits.

References

1. L. Afanasiev, T. Grust, M. Marx, J. Rittinger, and J. Teubner. An Inflationary Fixed Point Operator in XQuery. In *ICDE 2008*, pages 1504–1506.
2. J. Barwise and L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Publications, 1996.
3. N. Bidoit and M. Ykhlef. Fixpoint Calculus for Querying Semistructured Data. In *WebDB'98*, volume 1590 of *Lecture notes in Computer Science*, pages 78–97. Springer, 1999.
4. D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive Description Logics with fixpoints based on automata on infinite trees. In *IJCAI'99*, pages 84–89.
5. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
6. M. Denecker. Extending classical logic with inductive definitions. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *CL*, volume 1861 of *Lecture Notes in Computer Science*, pages 703–717. Springer, 2000. 44

7. M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions. *Transactions On Computational Logic (TOCL)*, 9(2), March 2008.
8. M. Denecker and J. Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 84–96. Springer, 2007.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
10. E. Allen Emerson. Real-time and the Mu-Calculus (preliminary report). In *REX Workshop Mook'91*, volume 600 of *Lecture notes in Computer Science*, pages 176–194. Springer, 1992.
11. G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive Logic Programming and Its Applications. In V. Dahl and I. Niemelä, editors, *ICLP*, volume 4670 of *Lecture notes in Computer Science*, pages 27–44. Springer, 2007.
12. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
13. F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
14. M. Mariën, D. Gilis, and M. Denecker. On the relation between ID-Logic and Answer Set Programming. In J. J. Alferes and J. A. Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2004.
15. M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2008*, Lecture Notes in Computer Science, pages 211–224. Springer, 2008.
16. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC'01*, pages 530–535. ACM, 2001.
17. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(t). *J. ACM*, 53(6):937–977, 2006.
18. D. Park. Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5:59–78, 1970.
19. D. Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.
20. N. Pelov and E. Ternovska. Reducing inductive definitions to propositional satisfiability. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 2005.
21. J. P. Marques Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
22. R. S. Streett and E. Allen Emerson. An automata theoretic decision procedure for the propositional μ-calculus. *Information and Computation*, 81:249–264, 1989.
23. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Embedding Functions into Disjunctive Logic Programs

Yisong Wang[1,2], Jia-Huai You[2], and Mingyi Zhang[3]

[1] Department of Computer Science, Guizhou University, Guiyang, China
[2] Department of Computing Science, University of Alberta, Canada
[3] Guizhou Academy of Sciences, Guiyang, China

Abstract. We extend the notions of completion and loop formulas of normal logic programs with functions to a class of nested expressions that properly include disjunctive logic programs. We show that answer sets for such a logic program can be characterized as the models of its completion and loop formulas. These results provide a basis for computing answer sets of disjunctive programs with functions, by solvers for the Constraint Satisfaction Problem. The potential benefit in answer set computations for this approach has been demonstrated previously in the implementation called FASP, for normal logic programs with functions. We also present a formulation of completion and loop formulas for disjunctive logic programs with variables. This paper focuses on the theoretical development of these extensions.

1 Introduction

Logic programming based on stable model/answer set semantics, called *answer set programming* (ASP), has been considered a promising paradigm for declarative problem solving. The general idea is to encode a problem by a logic program such that the answer sets of the program correspond to the solutions of the problem [1, 20, 21]. With the state-of-the-art ASP solvers such as CLASP, CMODELS, SMODELS, and DLV, ASP has been applied to a number of practical domains [1].

ASP has been extended in several directions, one of which is to logic programs with nested expressions (or just *nested expressions*) [16]. More recently, nested expressions have been extended to quantified equilibrium logic [22] and arbitrary first-order sentences [8].

Another direction is the idea of computing answer sets of logic programs by utilizing off-the-shelf solvers from other constraint solving formalisms, e.g., by SAT solvers [18, 9], by pseudo boolean solvers [19], and by CSP solvers [17]. The idea is that if a program has no positive loops, or is *tight on a set* which is a model of completion hence a supported model of the program [7], the model is then an answer set. Otherwise, loop formulas can be used to eliminate models of completion that are not answer sets, and more importantly, perform conflict-driven learning and generate non-chronological backtracking. The loop formulas approach has been extended to disjunctive logic programs [12].

In the third direction, ASP has recently been extended with functions. There are two different approaches to accommodating functions. One of them treats functions over Herbrand interpretations [2, 13, 23, 5], in which, just like Horn clause logic programming, functions are interpreted by fixed mappings, and are language symbols used

to define recursive data structures. This approach yields a language which is more expressive than the standard function-free ASP language.

In the other approach, to economically and naturally encode problems in ASP, Lin and Wang considered adding functions into normal logic programs in which functions are taken for mappings over finite, non-Herbrand domains together with the unique name assumption [17]. They extended the notions of completion and loop formulas and show that through program completion and loop formulas, a normal logic program with functions can be transformed to a Constraint Satisfaction Problem (CSP). Thus, off-the-shelf CSP solvers can be used as black boxes in computing answer sets. The system FASP[4] is such an implementation.

In the approach by Lin and Wang, a program with functions will be grounded to a finite ground program for the answer set computation. Thus it doesn't deal with infinite domains of any sort. Its aim is at providing a flexible knowledge representation language enabling both relations and functions. The approach is closely related to the functional logic language of [4]. A main difference is in [4], functions are partial, while in [17] they must be total.

In this paper, we consider further adding functions into nested logic programs with rules of the form

$$a_1; \ldots; a_n \leftarrow b_1, \ldots, b_m, G$$

where a_i are atoms, b_j are atoms or equality atoms, and G is a nested formula in which every occurrence of an atom occurs in the scope of the negation-as-failure operator "not". Following [12], these programs with nested expressions are called *disjunctive logic programs*.

Disjunctive logic programs of this kind can be seen to represent nested expressions as defined in [16] in the following sense. First, since nested expressions can be transformed to disjunctive logic programs with negation-as-failure in the head [3], the latter can be viewed as a normal form for nested expressions. Also in [16], it was shown that a negative literal in the head of such a disjunctive rule can be moved to the body by adding a not to it (i.e., $not\, a$ in the head becomes $not\, not\, a$ in the body).

In this paper, we define answer sets for nested expression with functions, and then formulate completion and loop formulas for disjunctive logic programs with functions. We show that loop formulas, along with program completion, capture answer sets of disjunctive logic programs with functions. This can be seen as a generalization of the main results of [17]. It turns out that, in order to incorporate functions, the notions of completion, dependency graphs, loops, and loop formulas all require a nontrivial generalization.

In general, a logic program may have an exponential number of loops and loop formulas [15]. To avoid computing similar loops, first-order loops and loop formulas were proposed for normal logic programs with variables [6], which are recently extended to arbitrary first-order sentences [11]. Since the language of disjunctive logic programs with functions is a many-sorted first-order language and an encoding in it is often written with variables, this motivated us to extend first-order loops and loop formulas to disjunctive logic programs with functions. This can be regarded a generalization of normal logic programs [6] and disjunctive logic programs [11] to include functions.

[4] http://www.cse.ust.hk/fasp/

We begin in the next section with a review of the definitions of answer sets for nested expressions. In Section 3 we add functions into nested expressions and extend these definitions to the new context. Then in Section 4, we formulate first-order loops and loop formulas for these programs. Related work is commented in Section 5. Section 6 contains final remarks and a discussion of future work.

2 Preliminary

The concept of atom is defined as in propositional logic. *Elementary formulas* are atoms and special symbols $\bot$ ("false") and $\top$ ("true").[5] *Formulas* are built from elementary formulas using the unary connective not (negation as failure) and the binary connectives "," (conjunction) and ";" (disjunction). A *rule with nested expressions* is an expression of the form

$$F \leftarrow G \tag{1}$$

where both G and F are formulas. A *logic program with nested expressions* (or called *nested logic program*) is a finite set of rules with nested expressions.

For any formula F, G and H, we may write

$$F \rightarrow G; H$$

to stand for the formula

$$(F, G); (not\ F, H)$$

which reads like an if-then-else statement.

Let F be a formula and Z a set of atoms. That Z *satisfies* F, written $Z \models F$, is defined as follows:

- for an atom a, $Z \models a$ if $a \in Z$
- $Z \models \top$
- $Z \models (F, G)$ if $Z \models F$ and $Z \models G$
- $Z \models (F; G)$ if $Z \models F$ or $Z \models G$
- $Z \models not\ F$ if $Z \not\models F$.

Note that, since $\bot$ is not an atom, it follows that $Z \not\models \bot$, for any set of atoms Z.

Z *satisfies* a logic program P if for every rule of the form (1) in P, $Z \models F$ whenever $Z \models G$. The *reduct* F^Z of a formula F w.r.t. a set of atoms Z, is defined recursively as

- for elementary F, $F^Z = F$
- $(F, G)^Z = (F^Z, G^Z)$
- $(F; G)^Z = (F^Z; G^Z)$
- $(not\ F)^Z = \begin{cases} \bot, & \text{if } Z \models F^Z \\ \top, & \text{otherwise} \end{cases}$
- $(F \leftarrow G)^Z = (F^Z \leftarrow G^Z)$.

[5] The syntax defined in [16] allows classical negation. Classical negation can be eliminated by introducing auxiliary atoms [14].

The *reduct* of a logic program P w.r.t. a set Z of atoms is the following set:

$$\{(F \leftarrow G)^Z : F \leftarrow G \in P\}.$$

A set of atoms Z is an *answer set* of a nested logic program P not containing *not* if Z is a minimal set satisfying P. For a nested logic program P, Z is an *answer set* of P if Z is an answer set of the reduct P^Z.

For example, consider the logic program P

$$a \leftarrow not\ not\ a. \tag{2}$$

The reduct of P w.r.t. Z is $a \leftarrow \top$ if $a \in Z$, and $a \leftarrow \bot$ otherwise. Thus P has two answer sets $\emptyset$ and $\{a\}$.

3 Nested Expressions with Functions

Now, we assume that the underlying language $\mathcal{L}$ is a many-sorted first-order language which may have pre-interpreted function symbols like the standard arithmetic functions such as "$+$", "$-$" and so on. *Elementary formulas* are atoms, equality atoms (written $s = t$)[6], and special symbols $\bot$ ("false") and $\top$ ("true"), and *formulas* are built from elementary formulas using "*not*", "," and ";" as before. By abuse of notation, we may write $s \neq t$ for $not\ (s = t)$. A *nested logic program with functions* (a *logic program* or just a *program*) is a finite set of rules of the form

$$H \leftarrow F \tag{3}$$

where H and F are formulas of $\mathcal{L}$, together with a set of type definitions, one for each type τ used in the logic program, of the form

$$\tau : D \tag{4}$$

where D is a finite nonempty set of elements, called a *domain*[7]. Here we require that if a constant c of type τ occurs in a rule of a program P then c must be contained in the domain of τ. Recall that $\mathcal{L}$ is a many-sorted first-order language. A logic program with variables is taken as the shorthand for the instantiated ground program; i.e., if a variable x is of type τ and the domain of τ is D according to the type definitions, then x is instantiated by elements in D. Thus we equate a logic program with variables with its grounded program unless otherwise stated.

Recall that once a variable in a rule is replaced by objects of a domain, the grounded rules may have symbols not in the original language $\mathcal{L}$. In the following, we let $\mathcal{L}_P$ be the language that extends $\mathcal{L}$ by introducing a new constant for each object in the domain

[6] Unless explicitly stated otherwise, an atom refers to a non-equality atom, of the form $p(\boldsymbol{t})$, where p is a predicate symbol. We distinguish atoms from equality atoms for convenience.

[7] We require domains to be finite for the practical implementation reason, since we will define loop formula for ground programs later, if using an infinite domain, a ground program may be infinite, in this case, loop formulas may be not well-defined.

of a type, but not a constant in $\mathcal{L}$. These new constants will have the same type as their corresponding objects. Now the fully instantiated rules will be in the language $\mathcal{L}_P$.

An interpretation of a program not only assigns truth values to ground instances of relations, but also assigns a mapping to each function symbol. Formally, an *interpretation* I of a logic program P, which is a first-order structure of $\mathcal{L}_P$, is a mapping such that

- The domains of I are those specified in the type definitions of P.
- A constant is mapped to itself.
- If R is a relation of arity $\tau_1 \times \cdots \times \tau_n$, and the type definitions $\tau_i : D_i$, $1 \leq i \leq n$, are in P, then I assigns a relation to R, denoted R^I, such that $R^I \subseteq D_1 \times \cdots \times D_n$.
- If f is a function of type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau_{n+1}$, $n \geq 1$, and the type definitions $\tau_i : D_i$, $1 \leq i \leq n+1$, are in P, then I assigns a mapping to f, denoted f^I, from $D_1 \times \cdots \times D_n$ to D_{n+1}.

Note that pre-interpreted functions follow their standard interpretations, which do not change from one interpretation to another.

Notice also that the notion of interpretation here is defined for a logic program instead of the underlying language $\mathcal{L}$. In the following, whenever we talk about interpretations of a formula, we assume that the formula under discussion occurs in the underlying logic program, where the type definitions are fixed.

Let I be an interpretation. The interpretation of a constant c under I, denoted c^I, is c. Inductively, if $s = f(t_1, ..., t_n)$ is a term, and each t_i, $1 \leq i \leq n$, is already interpreted, denoted t_i^I, then s^I denotes the constant mapped from the vector $\langle t_1^I, ..., t_n^I \rangle$ by f^I. This notation naturally extends to vectors of terms. In addition, for an n-ary predicate p and an n-vector $\boldsymbol{t}$, we will write $p^I(\boldsymbol{t})$ to denote the valuation of $p(\boldsymbol{t})$ under I.

We now define *satisfaction* for formulas with functions. Below, we only define it for elementary formulas. Along with the definition given in Section 2, the definition of satisfaction can be extended straightforwardly to all formulas. Let F be an elementary formula and I an interpretation. We say I *satisfies* F, written $I \models F$, if

- $F = \top$;
- $F = p(\boldsymbol{t})$, and $p^I(\boldsymbol{t})$ holds under I, where p is a predicate;
- F is an equality atom $t = t'$, and $t^I = t'^I$ (i.e., t and t' are mapped to the same constant).

To extend the notion of reduct to logic programs with functions, it is sufficient to define it for elementary formulas as well. Let F be an elementary formula and I an interpretation. The *reduct* of F w.r.t. I, written F^I, is defined as

- $p(\boldsymbol{t}^I)$ if $F = p(\boldsymbol{t})$;
- $\top$ if $F = \top$ or F is an equality atom such that $I \models F$;
- $\bot$ if $F = \bot$ or F is an equality atom such that $I \not\models F$.

With this, the notion of reduct as introduced in Section 2 naturally extends to logic programs with functions. Let P^I be the reduct of a program P w.r.t. I. Evidently, P^I mentions no functions, equalities and the negation as failure operator "*not*". Answer sets for such logic programs have been defined before: a set of atoms M is an answer

set of such a program P if M is a minimal set satisfying P. We now extend it to logic programs with functions.

In the following, given an interpretation I, we write I^a to denote the set of atoms that contain no functions and they are true under I.

Definition 1. *Let P be a nested logic program with functions and I an interpretation of P. I is an* answer set *of P if I^a is an answer set of P^I.*

Example 1. Consider the logic program P:

$$f : \tau \to \tau, \quad p : \tau, \quad \tau : \{0, 1\},$$
$$(p(1); f(0) \neq f(1)) \leftarrow (f(0) = 1 \to not\, p(1); p(0)). \tag{5}$$

Notice that the types of the predicate p and the function f are part of the given language, not the program P; we write them in P for clarity. The rule (5) stands for

$$(p(1); f(0) \neq f(1)) \leftarrow (f(0) = 1, not\, p(1)); (f(0) \neq 1, p(0)). \tag{6}$$

Consider the following interpretation for P:

- I_1 with $f^{I_1}(0) = f^{I_1}(1) = 0$, $p^{I_1}(0)$ and $p^{I_1}(1)$ are false. P^{I_1} consists of a single rule
$$p(1); \bot \leftarrow (\bot, \top); (\top, p(0))$$
which is equivalent to
$$p(1) \leftarrow p(0).$$
Since $I_1^a = \emptyset$ and $\emptyset$ is an answer set of P^{I_1}, thus I_1 is an answer set of P.

Traditionally, two logic programs are said to be equivalent if they have the same set of answer sets. For nested expressions, Lifschitz et al. [16] propose a stronger notion of equivalence: Two formulas F and G are *equivalent* if, for any two interpretations I_1 and I_2, $I_1 \models F^{I_2}$ iff $I_1 \models G^{I_2}$.

We adopt the same notion of equivalence for nested logic programs with functions.

Proposition 1. *For any formulas F and G, the rule $F; t = t' \leftarrow G$ is equivalent to $F \leftarrow G, t \neq t'$.*

Lifschitz et al. also showed a number of results (cf. Propositions 3–7 in [16]). We can easily extend these results to nested logic programs with functions. Particularly relevant to this paper, we can show that any rule of the form (3) is equivalent to

- a finite set of rules of the form
$$a_1; \ldots; a_n \leftarrow G \tag{7}$$
where G is a formula;
- a finite set of rules of the form[8]
$$a_1; \ldots; a_n \leftarrow b_1, \ldots, b_m, G \tag{8}$$

[8] It is known that for a polynomial time transformation new propositional symbols may need to be used, e.g., to convert a conjunctive normal form to a disjunctive normal form (for transformation of nested expressions see, e.g., [24]).

where $a_i (1 \leq i \leq n)$ and $b_j (1 \leq j \leq m)$ are atoms or equality atoms of $\mathcal{L}$, and G is a formula of $\mathcal{L}$ in which every occurrence of an atom is in the scope of the negation-as-failure operator "not". For convenience, we abbreviate a rule of the form (8) by

$$F \leftarrow B, G \tag{9}$$

where F stands for "$a_1; \ldots; a_n$" and B stands for "$b_1, \ldots, b_m$". In the following, a logic program consisting of rules of the form (9) is called a *disjunctive logic program with functions* (or a *disjunctive logic program*, or just a *program* if no confusion arises).

It is important to mention that though functions can enrich the language for knowledge representation, they are not absolutely necessary semantically speaking. As shown in [17], functions can be eliminated as follows.

Let P be a logic program that may have functions. For each function $f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ in P, we introduce two corresponding relations f_r and $\overline{f}_r$. They both have the arity $\tau_1 \times \cdots \times \tau_n \times \tau$, and informally $f_r(x_1, ..., x_n, y)$ stands for $f(x_1, ..., x_n) = y$ and $\overline{f}_r(x_1, ..., x_n, y)$ for $f(x_1, ..., x_n) \neq y$. Now let $\mathbb{F}(P)$ be the union of the rules obtained by grounding the following rules for each function f in P using the domains in the type definitions of P:

$$\begin{aligned}
&\leftarrow f_r(x_1, \ldots, x_n, y_1), f_r(x_1, \ldots, x_n, y_2), y_1 \neq y_2, \\
&f_r(x_1, \ldots, x_n, y) \leftarrow not\ \overline{f}_r(x_1, \ldots, x_n, y), \\
&\overline{f}_r(x_1, \ldots, x_n, y) \leftarrow f_r(x_1, \ldots, x_n, z), y \neq z.
\end{aligned}$$

Let $\mathbb{R}(P)$ be the set of rules obtained from the rules in P by the following transformation:

- evaluate all terms that mention only constants and pre-interpreted functions to constants;
- repeatedly replace each functional term $f(u_1, \ldots, u_n)$, where each u_i is a simple term in which it does not mention a function symbol, by a new variable x and add $f_r(u_1, \ldots, u_n, x)$ to the body of the rule as a conjunctive term where the term appears;
- instantiate all the rules obtained in the previous step, please note that, equality atom will be replaced with $\top$ if it is of the form $c = c$, and $\bot$ if it is of the form $c = c'$ where c and c' are two distinct constants.

$\mathbb{F}(P) \cup \mathbb{R}(P)$ is a logic program without functions and equality, and there is a one-to-one correspondence between it and P:

Theorem 1. *Let P be a nested logic program with functions. An interpretation I for P is an answer set of P iff $\mathbb{R}(I)$ is an answer set of $\mathbb{F}(P) \cup \mathbb{R}(P)$, where $\mathbb{R}(I)$ is the set of atoms that are true in I:*

$$\mathbb{R}(I) = \{p(\boldsymbol{c}) \mid p^I(\boldsymbol{c})\ holds\} \cup \{f_r(\boldsymbol{c}, a) \mid f^I(\boldsymbol{c}) = a\} \cup \{\overline{f}_r(\boldsymbol{c}, a) \mid f^I(\boldsymbol{c}) \neq a\}.$$

3.1 Completion

We now define completion for nested logic programs with functions, which generalizes that of [12] due to the incorporation of functions.

Let P be a logic program. An atom $p(c_1, \ldots, c_k)$ is said to *reside in* P if p is a predicate of type $\tau_1 \times \ldots \times \tau_n$ in P, $\tau_i : D_i$ is in the type definitions of P and $c_i \in D_i$ for each $i (1 \leq i \leq n)$. By $\mathcal{A}toms(P)$ we denote the set of atoms residing in P.

Given a formula F, we denote by $pa(F)$ the set of the *positive atoms* in F. An atom $p(\boldsymbol{t})$ is said to be *positive in* F if there is at least one occurrence of $p(\boldsymbol{t})$ in F that is not in the scope of negation as failure.

In the following, we identify a nested formula with a classical formula by replacing "," with "$\wedge$", ";" with "$\vee$" and "*not* " with "$\neg$". Also, when we talk about the completion of a logic program, we always assume the rules in the program are of the form (7).

Let P be a logic program whose rules are of the form (7). The *completion* of P, written $\mathit{COMP}(P)$, is defined as the set of classical formulas:

- for each rule of the form (7) in P

$$G \supset \bigvee_{1 \leq i \leq n} a_i \tag{10}$$

- and for each atom $p(\boldsymbol{c}) \in \mathcal{A}toms(P)$,

$$p(\boldsymbol{c}) \supset \bigvee_{1 \leq i \leq n} \left[\begin{array}{r} G_i \wedge \left(\bigwedge_{q(\boldsymbol{s}) \in pa(F_i)} \neg q(\boldsymbol{s}) \right) \\ \wedge \left(\bigvee_{p(\boldsymbol{t}) \in pa(F_i)} \boldsymbol{c} = \boldsymbol{t} \right) \\ \wedge \left(\bigwedge_{p(\boldsymbol{t}) \in pa(F_i)} (\boldsymbol{c} = \boldsymbol{t} \vee \neg p(\boldsymbol{t})) \right) \end{array} \right] \tag{11}$$

  where
  - $(F_1 \leftarrow G_1), \ldots, (F_n \leftarrow G_n)$ are the rules of P such that the atom $p(\boldsymbol{t})$ occurs in $F_i (1 \leq i \leq n)$ for some $\boldsymbol{t}$,
  - $q(\boldsymbol{s})$ is an atom such that q is distinct from p, and
  - $\boldsymbol{c} = \boldsymbol{t}$ if the two vectors have the same length and their corresponding components are all equal.

Intuitively, the equation (11) says that any atom, that holds under some interpretation, must have some supports. Note that the definition generalizes that of [12] since the formulas in the second and third lines of (11) are equal to $\top$ if the underlying language of P is propositional.

Example 2. Consider the following logic program P

$$\begin{array}{c} p, q : \tau, \quad f : \tau \to \tau, \quad \tau : \{0, 1\}, \\ p(f(0)); p(f(1)); f(0) = f(1) \leftarrow p(0), \mathit{not}\, q(f(0)). \end{array} \tag{12}$$

By Proposition 1, any head equality atom can be moved to the body and negated. We thus consider, equivalently, the following rule instead:

$$p(f(0)); p(f(1)) \leftarrow p(0), \mathit{not}\, q(f(0)), f(0) \neq f(1) \tag{13}$$

Note that $\mathcal{A}toms(P) = \{p(0), p(1), q(0), q(1)\}$. $\mathcal{COMP}(P)$ consists of

$$\begin{aligned}
p(0) \wedge \neg q(f(0)) \supset p(f(0)) \vee p(f(1)) \vee f(0) = f(1),\\
q(0) \supset \bot,\\
q(1) \supset \bot,\\
p(0) \supset (p(0) \wedge \neg q(f(0)) \wedge f(0) \neq f(1)) \wedge (0 = f(0) \vee 0 = f(1)) \wedge\\
(0 = f(0) \vee \neg p(f(0))) \wedge (0 = f(1) \vee \neg p(f(1))),\\
p(1) \supset (p(0) \wedge \neg q(f(0)) \wedge f(0) \neq f(1)) \wedge (1 = f(0) \vee 1 = f(1)) \wedge\\
(1 = f(0) \vee \neg p(f(0))) \wedge (1 = f(1) \vee \neg p(f(1))).
\end{aligned}$$

Suppose I is an interpretation of P such that $f^I(0) = 0$, $f^I(1) = 1$, and $p^I(0)$ holds. The reduct P^I consists of a unique rule

$$p(0); p(1); \bot \leftarrow p(0), \top$$

whose answer set is $\emptyset$ while $I^a = \{p(0)\}$. Thus I is not an answer set of P. However, we can verify that I is a model of $\mathcal{COMP}(P)$.

3.2 Loops and Loop Formulas

Recall in [17], we say that an atom $p(t_1, \ldots, t_n)$ can *cover* an atom $p(c_1, \ldots, c_n)$ in $\mathcal{A}toms(P)$ if for each $1 \leq i \leq n$,

- if t_i mentions only constants and pre-interpreted functions, then t_i can be evaluated to c_i,
- if t_i is $f(\boldsymbol{s})$ and cannot be evaluated independently of interpretations, then c_i has the same type as the range of f.

Intuitively, this means that $p(t_1, \ldots, t_n)$ may become $p(c_1, \ldots, c_n)$ under some assignment to functions.

Given a logic program P, the *positive dependency graph* of P, written G_P, is the directed graph (V, E), where $V = \mathcal{A}toms(P)$, and for any $p(\boldsymbol{c}), q(\boldsymbol{d}) \in V$, $(p(\boldsymbol{c}), q(\boldsymbol{d})) \in E$ if there is a rule $F \leftarrow G$ in P such that

- there is an atom $p(\boldsymbol{t}) \in pa(F)$ that can cover $p(\boldsymbol{c})$,
- there is an atom $q(\boldsymbol{s}) \in pa(G)$ that can cover $q(\boldsymbol{d})$,
- if the ith element in the above $\boldsymbol{t}$ and the kth element in the above $\boldsymbol{s}$ are syntactically identical, then the ith element in $\boldsymbol{c}$ and the kth element in $\boldsymbol{d}$ are also syntactically identical.

A nonempty subset L of $\mathcal{A}toms(P)$ is a *loop* of P if G_P has a non-zero length cycle that goes through only and all the nodes in L. In the following, to define loop formulas, we assume that every rule has the form (9). Let L be a loop of a logic program P, and an atom $p(\boldsymbol{c}) \in L$. The *external support formula* of $p(\boldsymbol{c})$ relative to L, written

$ES(p(\boldsymbol{c}), L, P)$, is the following formula:

$$\bigvee_{1\leq i\leq n}\left[\begin{array}{r} B_i \wedge G_i \wedge \left(\bigvee_{p(\boldsymbol{t})\in pa(F_i)} \boldsymbol{c}=\boldsymbol{t}\right) \\ \wedge\left(\bigwedge_{\substack{q(\boldsymbol{d})\in L \\ q(\boldsymbol{t})\in pa(B_i)}} \boldsymbol{t}\neq\boldsymbol{d}\right) \\ \wedge\left(\bigwedge_{q(\boldsymbol{t})\in pa(F_i)}\left(\left(\bigwedge_{q(\boldsymbol{d})\in L}\boldsymbol{t}\neq\boldsymbol{d}\right)\supset\neg q(\boldsymbol{t})\right)\right)\end{array}\right] \tag{14}$$

where $(F_1 \leftarrow B_1, G_1), \ldots, (F_n \leftarrow B_n, G_n)$ are all of the rules of P such that, for each $i (1 \leq i \leq n)$, there exists an atom $p(\boldsymbol{t'}) \in pa(F_i)$ that can cover $p(\boldsymbol{c})$, B_i is in the form of conjunction of atoms or equality atoms, and G_i is a formula in which every occurrence of each atom is in the scope of "not". The intended meaning of (14) is implied by its name, i.e., an atom that holds under an interpretation must have at least one support which does not depend on any atom in the loop L.

The *loop formula* of L in P, written $LF(L, P)$, is then the following formula:

$$\bigvee_{A\in L} A \supset \bigvee_{A\in L} ES(A, L, P). \tag{15}$$

This definition clearly generalizes the one for disjunctive loop formulas [12].

Example 3. (Continue with Example 2) $L = \{p(0)\}$ is a unique loop of P. $LF(L, P)$ is the following formula

$$\begin{array}{r} p(0) \supset (p(0) \wedge \neg q(f(0)) \wedge f(0) \neq f(1)) \wedge (0 = f(0) \vee 0 = f(1)) \\ \wedge(0 = f(0) \vee \neg p(f(0))) \wedge (0 = f(1) \vee \neg p(f(1))) \\ \wedge(0 \neq 0)\end{array}$$

which is equivalent to $p(0) \supset \bot$ and then $\neg p(0)$. Clearly, the interpretation I of P in Example 2 does not satisfy $LF(L, P)$. Consequently I is not an answer set of P.

Theorem 2. *Let P be a nested logic program with functions. An interpretation I for P is an answer set of P iff I is a model of $COMP(P) \cup LF(P)$ where $LF(P)$ is the set of loop formulas of P.*

To compute answer sets of normal logic programs with functions, through completion and loop formulas, FASP requires no function occurring in predicates and functions as an argument. For this purpose, a transformation $\mathbb{T}$ was introduced [17]. Similarly, it is not difficult to transform a nested logic program with functions into one that contains no function in any predicate and function.

4 First-Order Loop Formulas

In the section, we assume that a logic program contains a finite set of rules possibly with variables. We further assume that every rule of a logic program is of the form (9), and through variable renaming, no two rules share common variables.

Recall that, the underlying language $\mathcal{L}$ is a many-sorted first-order language. Thus every predicate has an arity that specifies the number of arguments the predicate has and the type (sort) of each argument, and similarly for constants and functions. Variables also have types associated with them, and when they are used in formulas, their types are normally clear from the context [17].

Let $\mathcal{D}$ be a collection of domains in which there is a unique domain corresponding to each type τ, denoted by $\mathcal{D}_\tau$. Given a typed first-order sentence φ, the *instantiation* of φ on $\mathcal{D}$, written $\varphi|\mathcal{D}$, is a formula defined inductively as follows:

- if φ does not have quantifications, then $\varphi|\mathcal{D}$ is the result of replacing $d = d$ by $\top$ and $d_1 = d_2$ by $\bot$ in φ, where d is any constant, and d_1 and d_2 are any two distinct constants;
- $\exists x_\tau.\varphi|\mathcal{D}$ is $\left(\bigvee_{d\in\mathcal{D}_\tau} \varphi(x_\tau/d)\right)|\mathcal{D}$;
- $(\varphi_1 \vee \varphi_2)|\mathcal{D}$ is $\varphi_1|\mathcal{D} \vee \varphi_2|\mathcal{D}$;
- $(\neg\varphi)|\mathcal{D}$ is $\neg(\varphi|\mathcal{D})$.

Let r be a rule of the form (9). We say r is not in *normal form* if there is an atom in $pa(F)$ containing at least one constant. Otherwise, we say r is in *normal form*. A logic program is in *normal form* if every rule of the logic program is in normal form. Obviously, we can turn every logic program to normal form using equality. In the following, we assume every logic program is in normal form unless stated otherwise.

A *binding* is an expression of the form α/t_τ where α is a variable of type τ or a function term $f(\boldsymbol{t})$ whose range is of the type τ, t_τ a variable or constant of type τ. A *substitution* is a set of bindings containing at most one binding for each variable and functional term.

Given a logic program P (with variables), the *completion* of P, denoted by *comp*(P), is the set of formulas defined as

- for each rule of the form (9) in P,

$$\forall \boldsymbol{x}(\exists \boldsymbol{y}.(B \wedge G) \supset F) \tag{16}$$

  where $\boldsymbol{x}$ is the tuple of variables occurring in F and $\boldsymbol{y}$ is the tuple of variables occurring in B or G but not in F;
- for each predicate p,

$$\forall \boldsymbol{x}.p(\boldsymbol{x}) \supset \bigvee_{1\le i\le n} \exists \boldsymbol{y}_i. \left[\begin{array}{r} (B_i \wedge G_i) \wedge \left(\bigwedge_{q(\boldsymbol{s})\in pa(F_i)} \neg q(\boldsymbol{s})\right) \\ \wedge \left(\bigvee_{p(\boldsymbol{t})\in pa(F_i)} \boldsymbol{x} = \boldsymbol{t}\right) \\ \wedge \left(\bigwedge_{p(\boldsymbol{t})\in pa(F_i)} (\boldsymbol{x} = \boldsymbol{t} \vee \neg p(\boldsymbol{t}))\right) \end{array}\right] \tag{17}$$

  where
  - q is a predicate different from p;
  - $\boldsymbol{x}$ is a tuple of distinct variables that are not in P, and match p's arity;
  - $(F_1 \leftarrow B_1, G_1), \ldots, (F_n \leftarrow B_n, G_n)$ are the rules in P whose head mentions the predicate p;
  - for each $1 \le i \le n$, $\boldsymbol{y}_i$ is the tuple of variables occurring in B_i, G_i or F_i but not in $\boldsymbol{x}$.

In particular, if a predicate p does not occur in the head of any rule in P, then we have $\forall \boldsymbol{x}.\neg p(\boldsymbol{x})$ in the completion of P.

Given a logic program P, its *(first-order) dependency graph* G_P can be defined similar to that of [6, 11]. Formally, let $\sigma(P)$ be the signature consisting of object and predicate constants occurring in the rules of P. Now G_P is the directed graph (V, E), where

- V is a set of non-equality atoms formed from $\sigma(P)$, along with an infinite supply of typed variables; please note that there is no atom in V mentioning function symbols.
- $(p(\boldsymbol{t})\theta, q(\boldsymbol{t}')\theta)$ is in E if there is a rule of the form (9) in P such that $p(\boldsymbol{t}) \in pa(F)$, $q(\boldsymbol{t}') \in B$ and θ is a substitution.

A finite non-empty subset L of V is a *(first-order) loop* of P if there is a non-zero length cycle in G_P that goes through only and all the nodes in L. Please note that, since $\sigma(P)$ has nothing to do with the type definitions of P, loops of P are independent of the domains of P.

Let P be a logic program, L a loop of P and $p(\boldsymbol{t}) \in L$. The *(first-order) external support formula* of $p(\boldsymbol{t})$ with respect to L, written $es(p(\boldsymbol{t}), L, P)$, is the disjunction of

$$\bigvee_{\theta : p(\boldsymbol{t}) \in pa(F\theta)} \exists \boldsymbol{y}. \left[\begin{array}{r} (B\theta \wedge G\theta) \\ \wedge \left(\bigwedge_{\substack{q(\boldsymbol{s}) \in L \\ q(\boldsymbol{s}') \in pa(B\theta)}} \boldsymbol{s}' \neq \boldsymbol{s} \right) \\ \wedge \left(\bigwedge_{q(\boldsymbol{s}) \in pa(F\theta)} \left(\left(\bigwedge_{q(\boldsymbol{s}') \in L} \boldsymbol{s}' \neq \boldsymbol{s} \right) \supset \neg q(\boldsymbol{s}) \right) \right) \end{array} \right] \tag{18}$$

for all rules of the form (9) in P, where θ is a substitution that maps the variables occurring in $\boldsymbol{t}$ to terms appearing in F, $\boldsymbol{y}$ is a tuple of variables occurring in $B\theta$, $G\theta$, or $F\theta$ but not in $\boldsymbol{t}$. The *(first-order) loop formula* of L for P, written *lf*(L, P), is then the following formula:

$$\forall \boldsymbol{x}. \left(\bigvee_{A \in L} A \supset \bigvee_{A \in L} es(A, L, P) \right) \tag{19}$$

where $\boldsymbol{x}$ is the tuple of variables occurring in L.

Example 4. Consider the following logic program P:

$$\begin{aligned} f : \tau \to \tau, \quad p, q : \tau, \\ p(f(a)) \leftarrow p(a), f(a) \neq a, \\ q(x); p(f(a)) \leftarrow q(f(a)), f(x) = a. \end{aligned}$$

We firstly turn its rules to normal form

$$\begin{aligned} p(y_1) \leftarrow p(a), f(a) \neq a, y_1 = f(a), \\ q(z_1); p(z_2) \leftarrow q(f(a)), f(z_1) = a, z_1 = x, z_2 = f(a). \end{aligned}$$

From the first rule, we can see that a is a constant of the type τ. The completion of P consists of the formulas

$$\begin{aligned} \forall y_1.(p(a) \wedge f(a) \neq a \wedge y_1 = f(a) \supset p(y_1)), \\ \forall z_1, z_2.(\exists x.(q(f(a)) \wedge f(z_1) = a \wedge z_1 = x \wedge z_2 = f(a)) \supset q(z_1) \vee p(z_2), \end{aligned}$$

$$\forall y_2.p(y_2) \supset \left[\begin{array}{r} \exists y_1.((p(a) \wedge f(a) \neq a \wedge y_1 = f(a) \\ \wedge y_1 = y_2 \wedge (y_1 = y_2 \vee \neg p(y_1)))) \\ \vee(\exists x, z_1, z_2.(q(f(a)) \wedge f(z_1) = a \wedge z_1 = x \wedge z_2 = f(a) \wedge \\ \neg q(z_1) \wedge (z_2 = y_2) \wedge (z_2 = y_2 \vee \neg p(z_2)))) \end{array}\right],$$

$$\forall y_2.q(y_2) \supset \left[\begin{array}{r} (\exists x, z_1, z_2.(q(f(a)) \wedge f(z_1) = a \wedge z_1 = x \wedge z_2 = f(a) \wedge \\ \neg p(z_2) \wedge (z_1 = y_2) \wedge (z_1 = y_2 \vee \neg q(z_1)))) \end{array}\right].$$

It is easy to see that $\{p(a)\}, \{q(a)\}$, and $\{q(x_1), \ldots, q(x_n)\}$ are loops of P. The loop formula $lf(\{p(a)\}, P)$ is

$$p(a) \supset \left[\begin{array}{r} (p(a) \wedge f(a) \neq a \wedge a = f(a) \wedge a \neq a \wedge (a \neq a \vee \neg p(a))) \vee \\ (\exists x, z_1.(q(f(a)) \wedge f(z_1) = a \wedge z_1 = x \wedge a = f(a) \wedge (a \neq a \vee \neg p(a)))) \end{array}\right].$$

Let's consider the domain $D = \{a, b\}$ in $\mathcal{D}$ for type τ. $lf(\{p(a)\}, P)|\mathcal{D}$ equals to

$$p(a) \supset (q(f(a)) \wedge f(a) = a \wedge \neg p(a) \wedge (f(a) = a \vee f(a) = b)).$$

Theorem 3. *Let P be a logic program in normal form and $\mathcal{D}$ is a collection of type definitions such that, for each type τ used in P, there is a finite and non-empty domain $D \in \mathcal{D}$ and D contains all τ-type constants occurring in P. An interpretation I of $P \cup \mathcal{D}$ is an answer set of $P \cup \mathcal{D}$ if and only if I is a model of $(comp(P) \cup lf(P))|\mathcal{D}$ where $lf(P)$ is the set of loop formulas of P.*

This theorem can be seen a generalization of Theorem 1 of [6], Theorem 2 of [17] and Proposition 1 of [11] for disjunctive logic programs.

5 Related Works

Functions are widely used in logic formalism stemming from first-order logic. In the ASP community, functions have already been considered in the general theory of stable models [8] and in Quantified Equilibrium Logic (QEL) [22]. The two theories generalize logic programs with nested expressions. In QEL, an equilibrium model is a Kripkle structure, for which no algorithm for computing answer sets is given. In the general theory of stable models, the answer set semantics is defined by translating first-order sentences into second-order ones. Lee and Meng proposed the notions of first-order loop formulas for disjunctive logic programs and arbitrary sentences. However they focused on Herbrand interpretations only. They also considered loop formulas in second-order logic for arbitrary first-order sentences. However, the notion of loops depends on a given interpretation in advance [11].

Calimeri *et al*. considered integrating functions into disjunctive logic programs and implemented it into DLV [5]. Again, they considered Herbrand models instead of non-Herbrand ones. To our knowledge, a closely related work is due to Cabalar [4]. A main difference is that functions in [4] are partial while in our case they are total. This difference has impact on how knowledge is represented. More importantly, the totality of functions enables a translation of programs to instances of CSP. Lee also defined the notions of loop formulas for nested logic programs directly. However, only propositional case is considered, i.e., function symbols are excluded [10].

6 Conclusion and Future Work

We have considered adding functions to disjunctive logic programs and formulated completion, loops and loop formulas, thus generalizing the main results of [17] to nested logic programs with functions. This enable us to extend FASP for such disjunctive logic programs in the future.

We have also shown how to extend the first-order loops and loop formulas to these programs. In general, an arbitrary program with variables and negation may be sensitive to the change of domains. Some restriction, like safety, has been proposed to guarantee domain independence [11]. It is worthwhile to investigate the same problem under non-Herbrand interpretations.

Acknowledgement

Yisong and Mingyi were partially supported by the Natural Science Foundation of China under grant 90718009, the Fund of Guizhou Science and Technology: 2008[2119], the Fund of Education Department of Guizhou Province: 2008[011] and Scientific Research Fund for talents recruiting of Guizhou University 2007[042].

References

1. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On finitely recursive programs. In *Proceedings of the Twenty Third International Conference on Logic Programming*, pages 89–103, Porto, Portugal, 2007. Springer.
3. Annamaria Bria, Wolfgang Faber, and Nicola Leone. Normal form nested programs. In *Logics in Artificial Intelligence, 11th European Conference*, volume 5293 of *Lecture Notes in Computer Science*, pages 76–88, Dresden, Germany, 2008. Springer.
4. Pedro Cabalar. A functional action language front-end. In *(presented at) The Third International Workshop on Answer Set Programming: Advances in Theory and Implementation*, Bath, UK, July 2005. http://www.dc.fi.udc.es/∼cabalar/asp05_C.pdf.
5. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in asp: Theory and implementation. In *Logic Programming, 24th International Conference*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424, Udine, Italy, 2008. Springer.
6. Yin Chen, Fangzhen Lin, Yisong Wang, and Mingyi Zhang. First-order loop formulas for normal logic programs. In *Proceedings of Tenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 298–307, Lake District of the United Kingdom, 2006. AAAI Press.
7. Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.
8. Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 372–379, 2007.
9. Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

10. Joohyung Lee. A model-theoretic counterpart of loop formulas. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 503–508, Edinburgh, Scotland, UK, 2005. Professional Book Center.
11. Joohyung Lee and Yunsong Meng. On loop formulas with variables. In *Proceedings of Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, pages 444–453, Sydney, Australia, 2008.
12. Jooyhung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proceedings of the Nineteenth International Conference on Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 451–465, Mumbai, India, 2003. Springer.
13. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
14. Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
15. Vladimir Lifschitz and Alexander A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.
16. Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.
17. Fangzhen Lin and Yisong Wang. Answer set programming with functions. In *Proceedings of Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, pages 454–464, Sydney, Australia, 2008.
18. Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.
19. Lengning Liu and Miroslaw Truszczynski. Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelligence Research*, 27:299–334, 2006.
20. V. Wiktor Marek and Miroslaw Truszczynski. Stable logic programming - an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*. K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, eds, Springer-Verlag, 1999.
21. Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
22. David Pearce and Agustín Valverde. Towards a first order equilibrium logic for nonmonotonic reasoning. In *Logics in Artificial Intelligence, 9th European Conference*, volume 3229 of *Lecture Notes in Computer Science*, pages 147–160, Lisbon, Portugal, 2004. Springer.
23. Mantas Simkus and Thomas Eiter. $\mathbb{FDNC}$: Decidable non-monotonic disjunctive logic programs with function symbols. In *Proceedings of Fourteenth International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 514–530, Yerevan, Armenia, 2007. Springer.
24. Jia-Huai You, Li-Yan Yuan, and Zhang Mingyi. On the equivalence between answer sets and models of completion for nested logic programs. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 859–866, Acapulco, Mexico, 2003. Morgan Kaufmann.

A General Method To Solve Complex Problems By Combining Multiple Answer Set Programs

Marcello Balduccini

Intelligent Systems, OCTO
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract This paper introduces the notion of Answer Set Programming (ASP) Machine, which is loosely inspired to Turing's Oracle Machines. The aim of this research is to use ASP "oracles" and ASP-represented transition systems to allow programmers to use exclusively ASP to solve problems that are beyond the expressive power of the basic language, rather than having to resort to auxiliary, often procedural programs. The advantage of our approach is a more uniform level of abstraction throughout the programs used to solve the problem, greater elaboration tolerance, as well as simplified proofs of the properties of programs, such as soundness and completeness.

1 Introduction

Answer Set Programming (ASP) [1,2] is a powerful programming paradigm that features sophisticated knowledge representation and reasoning capabilities.

In recent years, ASP has been used for a number of successful applications (e.g. [3,4,5,6]). One hurdle that programmers often face when using ASP, is that normal programs[1] can only solve NP-complete problems [7]. Whenever a problem of higher computational complexity is to be solved, either a suitable extension of ASP has to be selected (e.g. disjunctive programs, CR-Prolog [8], weak constraints [9]), or ad-hoc solutions have to be devised, such as writing an auxiliary, often procedural, program. In this second approach, the problem is reduced to computing the answer sets of a suitable sequence of ASP programs. For example, to find shortest plans, one can use the `#minimize` statement of LPARSE/SMODELS [10], or (in particular to have more control on the search strategy) write a normal program that solves the decision problem, and an auxiliary program that solves the optimization problem by selecting at each step a different plan length limit, and by checking if the normal program together with a specification of the given limit is consistent.

Unfortunately, it is often not easy to find suitable extensions of ASP that can be used in a natural way to solve the problem at hand. In that case, programmers are forced to write auxiliary programs. As we mentioned above, in other cases, one resorts to the use of auxiliary programs to have more control on the search strategy and increase efficiency.

[1] That is, logic programs with negation as failure but no disjunction.

Such auxiliary programs are typically relatively conceptually simple, but they still substantially complicate the task of proving properties of the overall program, such as soundness and completeness. Moreover, from a practical point of view, these auxiliary programs, being often procedural, require a substantial shift of perspective, and their writing involves the rather error-prone and cumbersome task of writing specifications at a substantially different level of abstraction compared to the ASP programs at the core of the application.

An additional motivation to the exclusive use of ASP stems from the recent observation [11,12] that describing an algorithm using a transition system instead of pseudocode makes it easier to prove its properties, compare it with other algorithms, and design new algorithms.

In this paper we explore an extension of the ASP paradigm that exploits these observations. Our approach to solving problems of high complexity is indeed based on reducing the task to computing the answer sets of a suitable sequence of ASP programs, but we use ASP to form such sequence. Although we do not suggest that our approach be used indiscriminately for general-purpose programming, we believe that it can be useful in simplifying the task of writing many ASP-based applications, as well as for testing search strategies before implementing with more efficient paradigms.

The paper is structured as follows. In Section 3 we describe our framework and show its application to a well-known NP-hard problem. In Section 4 we show how to the computations involved in our framework can be automated. In Section 5 we discuss related work. Finally, in Section 6 we draw conclusions and discuss future work.

2 Background

The syntax and semantics of answer set programming [1,2] are defined as follows. Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual. A literal is either an atom a or its strong (also called classical or epistemic) negation $\neg a$. The sets of atoms and literals formed from Σ are denoted by $atoms(\Sigma)$ and $literals(\Sigma)$ respectively.

A *rule* is a statement of the form:[2]

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

where r is an optional name of the rule (a label useful when talking about the rule), h and l_i's are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe h. We call h the *head* of the rule, and $\{l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n\}$ the *body* of the rule. Given a rule r, we denote its head and body by $head(r)$ and $body(r)$ respectively.

[2] For simplicity we focus on non-disjunctive programs. Our results extend to disjunctive (and other) programs in a natural way.

Often, rules of the form $h \leftarrow \text{not } h, l_1, \ldots, \text{not } l_n$ are abbreviated into $\leftarrow l_1, \ldots, \text{not } l_n$, and called *constraints*. The intuitive meaning of a constraint is that its body must not be satisfied.

An ASP *program* (or program for short) is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules over Σ. Often we denote programs by just the second element of the pair, and let the signature be defined implicitly. In that case, the signature of Π is denoted by $\Sigma(\Pi)$.

A set A of literals is *consistent* if no two complementary literals, a and $\neg a$, belong to A. A literal l is *satisfied* by a consistent set of literals A if $l \in A$. In this case, we write $A \models l$. If l is not satisfied by A, we write $A \not\models l$. A set $\{l_1, \ldots, l_k\}$ of literals is satisfied by a set A of literals ($A \models \{l_1, \ldots, l_k\}$) if each l_i is satisfied by A.

Programs not containing default negation are called *definite*. A consistent set of literals A is *closed* under a definite program Π if, for every rule of the form (1) such that the body of the rule is satisfied by A, the head belongs to A.

Definition 1. *A consistent set of literals A is an* answer set *of definite program Π if A is closed under all the rules of Π and A is set-theoretically minimal among the sets closed under all the rules of Π.*

The *reduct* of a program Π with respect to a set of literals A, denoted by Π^A, is the program obtained from Π by deleting:

- Every rule, r, such that $l \in A$ for some expression of the form not l from the body for r;
- All expressions of the form not l from the bodies of the remaining rules.

We are now ready to define the notion of answer set of a program.

Definition 2. *A consistent set of literals A is an* answer set *of program Π if it is an answer set of the reduct Π^A.*

To simplify the programming task, variables are often allowed to occur in ASP programs. A rule containing variables (called a *non-ground* rule) is then viewed as a shorthand for the set of its *ground instances*, obtained by replacing the variables in it by all the possible ground terms. Similarly, a non-ground program is viewed as a shorthand for the program consisting of the ground instances of its rules.

Later, we will also need the following notation. Given a program Π and a literal l, we say that Π *entails* l, and write $\Pi \models l$, if, for every answer set A of Π, $A \models l$. We say that Π *does not entail* l, and write $\Pi \not\models l$, if there exists one answer set A of Π such that $A \not\models l$.

3 ASP Machines and Execution Traces

Our approach is loosely inspired to the notion of Oracle Turing Machines [13], and consists in reducing the computation needed to solve a problem to a sequence of calls to

an "oracle" Ω, such that both the oracle and the program that generates the sequence of calls can be written in ASP. At this stage of the investigation, we focus on NP-complete oracles, as they can be implemented directly using ASP normal programs. Hence, from now on we identify an oracle with the normal program that implements it.

We denote the signature of an oracle Ω by Σ_Ω, and the set of literals, formed from Σ_Ω according to the usual conventions, by lit_Ω. An *input* to Ω is a consistent subset of lit_Ω.[3]

Given an input I, invoking oracle Ω is reduced to computing the answer sets of $I \cup \Omega$. To simplify dealing with situations in which $I \cup \Omega$ is inconsistent, we work under the convention that, for every input I, every answer set of $I \cup \Omega$ is non-empty.[4] Thus, an *output* of Ω (for input I) is an answer set of $I \cup \Omega$, or $\emptyset$ if $I \cup \Omega$ is inconsistent. Notice that, generally speaking, $I \cup \Omega$ may have multiple answer sets. We denote the *set of outputs* of Ω for I by $\Omega(I)$ (if $I \cup \Omega$ is inconsistent, it follows that $\Omega(I) = \{\emptyset\}$).

The computation that leads to calling the oracle is modeled as a sequence of transitions. Because of the similarity with the domain of reasoning about actions and change (see e.g. [14]), we call *fluents* the properties of interest of the states of the computation, whose truth value typically varies with state transitions. A *fluent literal* is either a fluent f or its negation $\neg f$. A state of the computation is a consistent set of fluent literals.[5]

A *configuration* is a pair $\langle \sigma, \rho \rangle$, where σ is a state of the computation and ρ is a consistent subset of lit_Ω. Intuitively, σ is the current state of the computation and ρ is one output from the latest call to the oracle. The initial configuration is $\langle \sigma^i, \emptyset \rangle$, where σ^i is a pre-determined initial state. Our goal is to use ASP to model a transition function that takes as input a configuration $\langle \sigma, \rho \rangle$ and returns a pair $\langle \sigma', \pi' \rangle$, where σ' is the next state of the computation and π' is the input to the next call to the oracle. The next configuration will then be $\langle \sigma', \rho' \rangle$, for some $\rho' \in \Omega(\pi')$ (there may be multiple next configurations). The computation terminates when it reaches a *terminal configuration*, which we will define using ASP. A terminal configuration may be successful, or failed, meaning that the configuration does not lead to a solution.

The formalization is as follows. Notice that, because here we focus on using a normal program to model the transition function, we restrict our attention to NP-complete transition functions. Given oracle Ω, let Σ_τ be a signature such that:

- all constant and function symbols of Σ_Ω occur also in Σ_τ;
- all the fluent literals of interest can be formed from function and constant symbols of Σ_τ;[6]
- all relation symbols of Σ_Ω are function symbols in Σ_τ;[7]

[3] Although here we use a different approach, one might also view Ω as an lp-function [14].

[4] This can be trivially achieved by ensuring that the oracle contains at least one fact.

[5] Although it is possible to require states of the computation to be also *complete* sets of fluent literals, that does not appear to play a major role in the formulation of our approach.

[6] As often done in the literature, we assume that either terms can contain strong negation, or a suitable function symbol and axioms are introduced to allow writing negative fluent literals.

[7] That is, we reify the relations of Σ_Ω.

– unary relation symbols $state$, $next\_state$, $param$, $result$, and zero-ary relation symbols $terminal$ and $failed$ belong to Σ_τ, and are called *fixed relations*.

Literals of Σ_τ are denoted by lit_τ and are formed according to the usual conventions. Notice that the conditions above, together with the normal assumption that function and relation symbols in a signature are disjoint, imply that the relation symbols of Σ_Ω and of Σ_τ are disjoint. In the rest of the paper, we use the term Ω-literals to refer to both the literals formed from Σ_Ω and to the corresponding terms formed from Σ_τ. The fixed relations of Σ_τ are used to encode the transition function and to provide an interface with the oracle. Intuitively:

– $state(L)$ says that fluent literal L holds in the current state of the computation;
– $next\_state(L)$ says that fluent literal L will hold in the next state of the computation;
– $terminal$ says that the current configuration is terminal;
– $failed$ says that the current configuration does not lead to a solution;
– $param(I)$ says that Ω-literal I is part of the input for the next call to oracle Ω;
– $result(O)$ says that Ω-literal O is part of an output of the latest call to Ω.

Given a relation r and a set of literals A, $A|_r$ (called restriction of A to r) denotes the set of literals of A formed by relation r. If r is a unary relation, by $A \downarrow_r$ we denote the set of arguments of the atoms from $A|_r$. For example, $\{r(a), r(c)\} \downarrow_r = \{a, c\}$. We also denote by $A \uparrow_r$ the set of atoms formed by unary relation r, with the elements of A as arguments. For example, $\{a, c\} \uparrow_r = \{r(a), r(c)\}$.

A configuration $\langle \sigma, \rho \rangle$ is encoded in ASP by means of relations $state$ and $result$. For example, the configuration $\langle \{f_1, f_2\}, \{r_1, r_2\} \rangle$ is encoded by the set of atoms $\{state(f_1), state(f_2), result(r_1), result(r_2)\}$. We denote the ASP encoding of a configuration γ by $\alpha(\gamma)$. More precisely, $\alpha(\langle \sigma, \rho \rangle) = (\sigma \uparrow_{state}) \cup (\rho \uparrow_{result})$.

The transition function is encoded by a program τ, over signature Σ_τ

Definition 3. *Program τ over Σ_τ is a* transition program *if, for every configuration γ:*

– *if some answer set of $\alpha(\gamma) \cup \tau$ entails $terminal$, then $\alpha(\gamma) \cup \tau$ has a unique answer set;*
– *for every answer set A of $\alpha(\gamma) \cup \tau$, $A \models failed$ implies $A \models terminal$.*

For simplicity, in this paper we focus on deterministic transitions, and consequently, for every configuration γ, $\alpha(\gamma) \cup \tau$ has a unique answer set (but recall that the oracle may have multiple answer sets).

Definition 4. *An* ASP machine *is a tuple $\langle \tau, \sigma^i, \Omega \rangle$, where τ is a transition program, Ω is an oracle, and σ^i is the initial state of the computation.*

To simplify the notation, we adopt the convention that the initial state of the computation is the set $\{initial\}$, where $initial$ is a suitable fluent literal from Σ_τ. In that case, an ASP machine is denoted simply by the pair $\langle \tau, \Omega \rangle$.

Definition 5. *An* execution trace *for ASP machine* $\langle \tau, \sigma^i, \Omega \rangle$ *from configuration* $\langle \sigma_0, \rho_0 \rangle$ *is a (possibly infinite) sequence:*

$$\langle \sigma_0, \rho_0, \pi_1, \sigma_1, \rho_1, \pi_2, \sigma_2, \rho_2, \pi_3, \sigma_3, \rho_3, \ldots \rangle,$$

where σ_i*'s are states of the computation, and* π_i*'s,* ρ_i*'s are subsets of* lit_Ω*, such that:*

- $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau \models terminal$ *iff* σ_i, ρ_i *are the last two elements of the sequence;*
- *for every* i *such that* $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau \not\models terminal$*, there exists an answer set* A *of* $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau$ *such that: (i)* $\sigma_{i+1} = A \downarrow_{next\_state}$*, (ii)* $\pi_{i+1} = A \downarrow_{param}$*, and (iii)* $\rho_{i+1} \in \Omega(\pi_{i+1})$.

It follows from the definition that a configuration $\gamma_0 = \langle \sigma_0, \rho_0 \rangle$ is an execution trace from γ_0 for ASP machine $\langle \tau, \sigma^i, \Omega \rangle$ if $\alpha(\langle \sigma_0, \rho_0 \rangle) \cup \tau \models terminal$. By execution trace for an ASP machine $\langle \tau, \sigma^i, \Omega \rangle$, without any reference to a configuration, we mean an execution trace of the ASP machine from its *initial configuration* $\langle \sigma^i, \emptyset \rangle$.

We distinguish between *finite execution traces* and *infinite execution traces*. In a finite execution trace $\langle \sigma_0, \rho_0, \ldots, \pi_n, \sigma_n, \rho_n \rangle$, the pair formed by its last two elements, $\langle \sigma_n, \rho_n \rangle$, is called the *terminal configuration*. We also distinguish between *successful (finite) execution traces* and *failed (finite) execution traces*. A finite execution trace s is *failed* if its terminal configuration $\langle \sigma_n, \tau_n \rangle$ is such that $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau \models failed$. Otherwise, s is *successful*.

It is not difficult to check that every finite execution trace has $2 + 3k$ elements for some $k \geq 0$. We call k the *number of transitions* in the execution trace.

Definition 6. *A set* $\sigma \cup \rho$ *is a* hyper answer set *of an ASP machine* $\langle \tau, \sigma^i, \Omega \rangle$ *if* $\langle \sigma, \rho \rangle$ *is a terminal configuration for some successful execution trace for* $\langle \tau, \sigma^i, \Omega \rangle$.

To better illustrate the framework developed so far, let us demonstrate how it can be applied to the task of solving the Traveling Salesman Problem (TSP). Given a directed graph G with edges labeled by weights (representing the cost of traveling from one vertex to the other) and an initial vertex v_0, the goal, in the TSP, is to find a Hamiltonian cycle C from v_0 such that the sum of the weights of the edges traversed by C (also called *cost*) is minimal. The TSP is known to be NP-hard, while the decision problem version of TSP is NP-complete. Although the TSP can be solved quite naturally with some extensions of ASP, such as the `#minimize` statement of LPARSE/SMODELS [10] or weak constraints from DLV [9], we chose this problem because it is well-known and has a simple structure. Furthermore, to demonstrate the programmer's control on the search strategy, we implement the transition program so that it performs binary search.

We use an oracle Ω_H to solve the decision problem version of TSP. That is, Ω_H finds a Hamiltonian cycle C from v_0 *such that the cost of* C *is less than or equal to some given limit* l.

A possible ASP program for Ω_H is:[8]

% Path P visits every vertex V at most once.
$\leftarrow vertex(V2), vertex(V1), vertex(V),$
$in(V1, V), in(V2, V), V1 \neq V2.$
$\leftarrow vertex(V2), vertex(V1), vertex(V),$
$in(V, V1), in(V, V2), V1 \neq V2.$

% Path P must visit every vertex of the graph.
$reached(V2) \leftarrow vertex(V1), vertex(V2), init(V1), in(V1, V2).$
$reached(V2) \leftarrow vertex(V1), vertex(V2), reached(V1), in(V1, V2).$

$\leftarrow vertex(V), \text{not } reached(V).$

$\{in(V1, V2) : edge(V1, V2)\}.$

% Edge cost – cost of a selected edge.
$[r_1]\ ecost(V1, V2, L) \leftarrow$
$vertex(V1), vertex(V2), in(V1, V2), length(V1, V2, L).$

$[r_2]\ cost(T) \leftarrow$
$T[ecost(V1, V2, C) : vertex(V1) : vertex(V2) : ldom(C) = C]T,$
$ldom(T).$

% Path P must have a cost no greater than the current limit.
$\leftarrow ldom(T), ldom(L), cost(T), limit(L), T > L.$

The input to the oracle is provided as follows: the list of vertexes is specified by relation $vertex$; the initial vertex is specified by relation $init$; an edge from v_1 to v_2 is specified by an atom $edge(v_1, v_2)$; the weight of the edge from v_1 to v_2 is specified by an atom $length(v_1, v_2, l)$;[9] the limit to the cost of the Hamiltonian cycle is specified as $limit(l)$; relation $ldom$ specifies the domain for the weights and path cost.[10]

The first 6 rules use well-know techniques to find Hamiltonian cycles. Rule r_1 determines the cost of each selected edge. Rule r_2 uses a special LPARSE/SMODELS [10] construct to compactly specify that $cost(T)$ must hold if T is the cost of the selected Hamiltonian cycle. Finally, the last constraint discards any cycles whose cost is greater than the given limit.

[8] The program shown here is an extension of the one described at `http://www.cs.ttu.edu/~mgelfond/FALL02/asp.pdf`. To increase readability, the program is written using constructs available in LPARSE/SMODELS [10]. Converting to the language described in Section 2 is not difficult.

[9] More compact representations are also possible, which avoid using two separate relations $edge$ and $length$, but we prefer this encoding as it allows us to build incrementally on top of the existing ASP solutions to the problem of finding Hamiltonian cycles in non-weighted graphs.

[10] The use of $ldom$ is required only for proper grounding by LPARSE. We include it here to make the program directly executable with LPARSE+SMODELS [10].

Consider the problem instance P_{TSP} encoded by:

$$ldom(0..50).$$

$$vertex(s0; s1; s2; s3).$$
$$init(s0).$$

$$length(s0, s1, 6). \quad length(s1, s2, 5). \quad length(s2, s3, 4).$$
$$length(s3, s0, 3). \quad length(s0, s2, 2). \quad length(s1, s3, 1).$$

% Edges oriented in the opposite direction, same weights.
$$length(s1, s0, 6). \quad length(s2, s1, 5). \quad length(s3, s2, 4).$$
$$length(s0, s3, 3). \quad length(s2, s0, 2). \quad length(s3, s1, 1).$$

$$edge(V1, V2) \leftarrow length(V1, V2, L).$$

It is not difficult to check that $\Omega_H \cup P_{TSP} \cup \{limit(25)\}$ has an answer set containing the atoms $\{in(s0, s1), in(s1, s3), in(s3, s2), in(s2, s0), cost(13)\}$, encoding the solution to the decision problem corresponding to path $\langle s0, s1, s3, s2, s0\rangle$, of cost 13.

A transition function that performs binary search is encoded by program, τ_H, consisting of the rules described next. The general idea is to maintain, as part of the state of the computation, an encoding of the current search interval and of its midpoint, used as the limit value for the next call to the oracle. Fluents of interest are, thus, $latest\_min(V)$, $latest\_max(V)$ and $latest\_limit(V)$. The first set of rules from τ_H detects terminal configurations, checks if a Hamiltonian cycle was found by the latest call to the oracle, and detects failed execution traces.

$$terminal \leftarrow$$
$$\quad ldom(T),$$
$$\quad state(latest\_min(T)), state(latest\_max(T)).$$

$$hamcycle\_found \leftarrow$$
$$\quad ldom(T), result(cost(T)).$$

$$failed \leftarrow$$
$$\quad terminal, \text{not } hamcycle\_found.$$

The next set of rules uses information about the current state to determine the next search interval and its midpoint, and encode them using auxiliary relations, as follows:[11]

$$new\_interval(dom\_min, dom\_max) \leftarrow state(first\_run).$$

$$new\_interval(X, Y) \leftarrow$$
$$\quad ldom(X), ldom(Y),$$
$$\quad hamcycle\_found, state(latest\_limit(Y)), state(latest\_min(X)).$$

[11] The search could be made more efficient by taking into account, in the determination of the next search interval, the cost of the Hamiltonian cycle just found by the oracle. For illustrative purposes, however, we use the simpler technique described here.

$new\_interval(X+1, Y) \leftarrow$
$\quad ldom(X), ldom(Y),$
$\quad$ not $hamcycle\_found, state(latest\_limit(X)), state(latest\_max(Y)).$

$selected\_limit(T) \leftarrow$
$\quad ldom(X), ldom(Y), ldom(T), new\_interval(X, Y), T = ((X+Y)/2).$

In the rules above, dom_min and dom_max are predefined constants that define the range of interest for the search. Also notice how $hamcycle\_found$ is used, above, to select either the left or right subinterval of the current search interval. The next set of rules determines the next state of the computation from the auxiliary relations just defined:

$$next\_state(latest\_min(X)) \leftarrow ldom(X), ldom(Y), new\_interval(X, Y).$$

$$next\_state(latest\_max(Y)) \leftarrow ldom(X), ldom(Y), new\_interval(X, Y).$$

$$next\_state(latest\_limit(T)) \leftarrow ldom(T), selected\_limit(T).$$

The final set of rules defines the parameters to be passed to the oracle, and in particular the value of the limit for the decision problem:

$$param(limit(V)) \leftarrow ldom(V), selected\_limit(V).$$
$$param(ldom(X)) \leftarrow ldom(X).$$
$$param(vertex(Vert)) \leftarrow vertex(Vert).$$
$$param(init(Vert)) \leftarrow init(Vert).$$
$$param(edge(V1, V2)) \leftarrow edge(V1, V2).$$
$$param(length(V1, V2, L)) \leftarrow length(V1, V2, L).$$

Let us now focus on constructing an execution trace $\langle \sigma_0, \rho_0, \pi_1, \sigma_1, \rho_1, \ldots \rangle$ for the ASP machine $\langle \tau_H \cup P_{TSP}, \Omega_H \rangle$ from the initial configuration $\gamma_0 = \langle \{initial\}, \emptyset \rangle$. To save space, we will construct the relevant portions of the answer sets of the various programs by using the informal meaning of the ASP rules, rather than mathematical proofs. Let dom_min, dom_max be respectively 0 and 50. It is not difficult to show that $\alpha(\gamma_0) \cup \tau_H$ has a unique answer set A_0 containing $new\_interval(0, 50)$, $selected\_limit(25)$, $param(limit(25))$, and the corresponding definition of relation $next\_state$. According to our definition of execution trace, $\pi_1 = A_0 \downarrow_{param} = \{limit(25), vertex(s0), \ldots\} = P_{TSP} \cup \{limit(25)\}$ and $\sigma_1 = \{latest\_min(0), latest\_min(50), latest\_limit(25)\}$. Next, let us consider ρ_1. By definition, $\rho_1 \in \Omega_H(\pi_1)$. Notice that $\Omega_H \cup \pi_1$ has possibly multiple answer sets. One such answer set encodes the solution to the decision problem with $cost(13)$ shown earlier. Let us select that solution for the execution trace considered in this example. Hence, $\rho_1 = \{in(s0, s1), in(s1, s3), in(s3, s2), in(s2, s0), \ldots\}$. Let γ_1 denote $\langle \sigma_1, \rho_1 \rangle$. The next step consists in determining σ_2 and π_2 from the answer set, A_1, of $\alpha(\gamma_1) \cup \tau_H$. It can be shown that A_1 contains atoms $new\_interval(0, 25)$ and $selected\_limit(12)$. Hence, $\sigma_2 = \{latest\_min(0), latest\_max(25), latest\_limit(12)\}$ and $\pi_2 = P_{TSP} \cup \{limit(12)\}$. Set $\Omega_H(\pi_2)$ contains two answer sets, both encoding solutions with cost 11. Let us pick arbitrarily one such answer set as ρ_2. At this point, σ_3 and π_3 can

be found as above: σ_3 is $\{latest\_min(0), latest\_max(12), latest\_limit(6)\}$, while π_3 is $P_{TSP} \cup \{limit(6)\}$. Because no Hamiltonian cycle exists of cost 6 or less for the given instance, $\Omega_H(\pi_3) = \{\emptyset\}$. Hence, $\rho_3 = \emptyset$. Let us now consider the answer set, A_3, of $\alpha(\langle\sigma_3, \rho_3\rangle)$. Obviously, $hamcycle\_found \notin A_3$. That causes the right search subinterval to be selected, that is $A_3 \supseteq \{new\_interval(7, 12), selected\_limit(9)\}$. Thus, $\pi_4 = P_{TSP} \cup \{limit(9)\}$. As the shortest Hamiltonian cycle for the problem instance considered here has cost 11, once again $\Omega_H(\pi_4) = \{\emptyset\}$. The search proceeds along these lines until subinterval $[11, 11]$ is selected. In other words, it is not difficult to show that there exists some index k such that $\sigma_k = \{latest\_min(11), latest\_max(11), latest\_limit(11)\}$ and $\pi_k = P_{TSP} \cup \{limit(11)\}$. Set $\Omega_H(\pi_k)$ contains two answer sets, both encoding solutions with cost 11. Let us select:

$$\rho_k = \{in(s0, s2), in(s2, s1), in(s1, s3), in(s3, s0), cost(11), \ldots\}.$$

Let us now consider $A_k = \alpha(\langle\sigma_k, \rho_k\rangle)$. Clearly $terminal \in A_k$. Moreover, $hamcycle\_found \in A_k$, which implies that $failed \notin A_k$. Therefore, $\langle\sigma_0, \rho_0, \pi_1, \ldots, \sigma_k, \rho_k\rangle$ is a successful execution trace, and $\sigma_k \cup \rho_k$ is a hyper answer set of the ASP machine. The hyper answer set encodes the solution, of cost 11, corresponding to the path $\langle s0, s2, s1, s3, s0\rangle$.

4 Computing Execution Traces

An algorithm that computes a successful finite execution trace for an ASP machine $\langle\tau, \Omega\rangle$ from configuration $\langle\sigma_0, \rho_0\rangle$ is shown below (Algorithm 1).

Algorithm 1: FindSuccessfulTrace

Input: ASP machine $\langle\tau, \Omega\rangle$
A configuration $\langle\sigma_0, \rho_0\rangle$
Output: A successful finite execution trace $\langle\sigma_0, \rho_0, \pi_1, \sigma_1, \rho_1, \ldots, \sigma_n, \rho_n\rangle$ or $\perp$ if none was found.

1 $A :=$ the answer set of $\tau \cup \alpha(\langle\sigma_0, \rho_0\rangle)$
2 **if** $\{terminal, failed\} \subseteq A$ **then return** $\perp$
3 **if** $terminal \in A$ **then return** $\langle\sigma_0, \rho_0\rangle$
4 $\pi := (A \downarrow_{param})$
5 $\sigma := (A \downarrow_{next\_state})$
6 $O := \Omega(\pi)$
7 **while** $O \neq \emptyset$ **do**
8 $\quad\rho :=$ an arbitrary element of O
9 $\quad O := O \setminus \{\rho\}$
10 $\quad s := FindSuccessfulTrace(\langle\tau, \Omega\rangle, \langle\sigma, \rho\rangle)$
11 $\quad$ **if** $s \neq \perp$ **then return** $\langle\sigma_0, \rho_0, \pi\rangle \circ s$
12 **end**
13 **return** $\perp$

Given a configuration $\gamma_0 = \langle \sigma_0, \rho_0 \rangle$, algorithm FindSuccessfulTrace starts by checking whether γ_0 is failed. That is accomplished, as per the definition of execution trace, by checking whether the answer set A of $\tau \cup \alpha(\langle \sigma_0, \rho_0 \rangle)$ contains $\{terminal, failed\}$. If γ_0 is failed, the algorithm returns $\perp$. Next, FindSuccessfulTrace checks if γ_0 is terminal, and if so returns it. Otherwise, the algorithm extracts from A the next state of the computation, σ, and the parameters for the call to the oracle, π. The oracle is then invoked, and its outputs are stored in O. Notice that, by definition, O is guaranteed to be non-empty. Next, the algorithm attempts to construct a successful trace using each output of the oracle. To do that, an arbitrary element ρ of O is selected and removed from O. The algorithm is called recursively, to attempt to find a successful execution trace s from the new configuration $\langle \sigma, \rho \rangle$. If the attempt succeeds, the algorithm prepends to s the initial configuration as well as the parameters used in the call to the oracle and returns the resulting execution trace. Otherwise, the algorithm selects another output from the call to the oracle, and iterates. If all the attempts fail at constructing a successful execution trace from the configuration obtained from σ and an output of the oracle, the algorithm returns $\perp$. Let us now discuss soundness and completeness of the algorithm.

Lemma 1. *For every ASP machine $\langle \tau, \Omega \rangle$ and every configuration $\langle \sigma_0, \rho_0 \rangle$, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma_0, \rho_0 \rangle)$ returns a sequence $s \neq \perp$, then the first two elements of s are σ_0 and ρ_0; that is,*

$$s = \langle \sigma_0, \rho_0, \ldots, \rangle.$$

It is not difficult to show that every sequence (that is, every result except for $\perp$) returned by $FindSuccessfulTrace$ has $2 + 3k$ elements, for some $k \geq 0$. Therefore, let us extend the notion of number of transitions to the sequences returned by algorithm $FindSuccessfulTrace$. We denote the number of transitions in such a sequence s by $|s|$.

Theorem 1. *For every ASP machine $\langle \tau, \Omega \rangle$ and every configuration $\langle \sigma_0, \rho_0 \rangle$, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma_0, \rho_0 \rangle)$ returns a sequence $s \neq \perp$, then s is a successful execution sequence for $\langle \tau, \Omega \rangle$ from $\langle \sigma_0, \rho_0 \rangle$.*

Proof. *We proceed by induction on the number of transitions in the sequence returned by algorithm $FindSuccessfulTrace$.*

Base case: $|s| = 0$.
If $|s| = 0$, then by definition s contains two elements, that is, by Lemma 1, $s = \langle \sigma_0, \rho_0 \rangle$. Hence, s was returned at step 3 of the algorithm, which implies that $\tau \cup \alpha(\langle \sigma_0, \rho_0 \rangle)$ entails $terminal$, but it does not entail failed. By Definition 5, s is a successful execution trace.

Inductive step.
Let us assume that, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma, \rho \rangle)$ returns a sequence s with $n - 1$ transitions, then s is a successful execution trace, and let us prove that, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma, \rho \rangle)$ returns a sequence s' with n transitions, then s' is a successful execution trace.

Because $|s'| \geq 1$, s' contains at least $2 + 3 \cdot 1 = 5$ elements. Hence, the final result of algorithm $FindSuccessfulTrace$ must have been returned by step 11. Let σ_1, ρ_1,

π_1 denote the values of variables σ, ρ, and π at the moment of the execution of step 11, and notice that those variables had the same values at step 10. Consequently, there exists s such $s = FindSuccessfulTrace(\langle\tau, \Omega\rangle, \langle\sigma_1, \rho_1\rangle)$ and $s' = \langle\sigma_0, \rho_0, \pi_1\rangle \circ s$. By Lemma 1, the first two elements of s are σ_1 and ρ_1; that is, $s = \langle\sigma_1, \rho_1, \ldots\rangle$.

Because $|s'| = n$, $|s| = n-1$. By inductive hypothesis, s is a successful execution trace from $\langle\sigma_1, \rho_1\rangle$.

To prove the thesis, we need to show that the conditions of Definition 5 are satisfied. Because s has already been shown to be a successful execution trace, the conditions simplify to:

1. *$\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$ entails neither $terminal$ nor $failed$;*
2. *the answer set, A, of $\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$ is such that:*
   (a) *$\sigma_1 = A \downarrow_{next\_state}$;*
   (b) *$\pi_1 = A \downarrow_{param}$;*
   (c) *$\rho_1 \in \Omega(\pi_1)$.*

The fact that $\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$ entails neither $terminal$ nor $failed$ follows from the observation that, if that were not the case, steps 2 and 3 of the algorithm would have returned either $\perp$ or a sequence with 0 transitions.[12]

The fact that the other conditions hold is demonstrated as follows. Notice that the value of variable π does not change between step 4 and step 11. We have already established that the value of π at step 11 is π_1. Hence, step 4 guarantees that $\pi_1 = A \downarrow_{param}$. With a similar reasoning, we conclude that $\sigma_1 = A \downarrow_{next\_state}$ and $\rho_1 \in \Omega(\pi_1)$.

Therefore, s' is an execution trace by Definition 5. The fact that it is finite follows from the fact that s is finite. Finally, s' is successful because, if step 2 detects a failed execution sequence, the algorithm returns $\perp$ instead of a sequence.

Obviously, the algorithm is not complete, as it finds only one successful execution trace. What is more important, however, is that not even termination is guaranteed: in fact, if an infinite execution trace exists, nothing prevents the algorithm from attempting to construct it. To make guarantees about termination, we need to consider a more restricted class of ASP machines.

Definition 7. *An ASP machine $\mu = \langle\tau, \Omega\rangle$ is* finite *if every execution trace of μ is finite.*

Finite ASP machines allow one to guarantee not only termination, but also a weak notion of completeness.

Theorem 2. *For every finite ASP machine $\mu = \langle\tau, \Omega\rangle$:*

1. *$FindSuccessfulTrace(\mu, \langle\sigma^i, \emptyset\rangle)$ terminates;*
2. *If a successful execution trace exists for μ, $FindSuccessfulTrace(\mu, \langle\sigma^i, \emptyset\rangle)$ returns a successful execution trace.*

Proof. *Both statements can be proven by induction similarly to Theorem 1.*

[12] Regarding step 2, the fact that τ is a transition program guarantees that, if $failed$ is entailed by $\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$, then $terminal$ is also entailed.

It is possible to extend $FindSuccessfulTrace$ into an algorithm that returns a set of successful finite execution traces, as shown below (Algorithm 2). Not only the extended algorithm is sound, but, for finite ASP machines, it can also be shown to terminate and be complete.

Algorithm 2: FindAllSuccessfulTraces

Input: ASP machine $\langle \tau, \Omega \rangle$
A configuration $\langle \sigma_0, \rho_0 \rangle$
Output: A (possibly empty) set of successful finite execution traces.

```
1  A := the answer set of τ ∪ α(⟨σ0, ρ0⟩)
2  if {terminal, failed} ⊆ A then return ∅
3  if terminal ∈ A then return {⟨σ0, ρ0⟩}
4  π := (A ↓param)
5  σ := (A ↓next_state)
6  H := ∅
7  O := Ω(π)
8  while O ≠ ∅ do
9      ρ := an arbitrary element of O
10     O := O \ {ρ}
11     S := FindAllSuccessfulTraces(⟨τ, Ω⟩, ⟨σ, ρ⟩)
12     if S ≠ ∅ then H := H ∪ {⟨σ0, ρ0, π⟩ ∘ s | s ∈ S}
13 end
14 return H
```

5 Related Work

In [11] and [12], it is argued that describing an algorithm using a transition system instead of pseudocode makes it easier to prove its properties, compare it with other algorithms, and design new algorithms.

The fact that ASP provides a convenient framework to represent state transitions was highlighted by the research on using ASP for reasoning about actions and change, see e.g. [15,14,4].

Several extensions of the language of ASP allow one to deal with problems of complexity higher than NP-complete, e.g. [1,9,8,16].

The GnT system [17] finds answer sets of disjunctive logic programs by computing the answer sets of two normal programs automatically derived from the original program. Our approach is more general than [17] in that the oracle and the transition program are independently programmed, rather than obtained from automatic translation, and thus our technique can be applied to solve a variety of problems besides just the computation of the answer sets of disjunctive programs.

In [7], an ASP program is used to simulate a non-deterministic Turing machine. Hence, the encoding presented there is at a considerably lower level of abstraction than the one we discussed. Furthermore, [7] is focused upon a single ASP program, while in the present work a considerable effort was devoted to developing a suitable framework allowing the interaction between two ASP programs.

6 Conclusions and Future Work

In this paper we have explored the use of answer set programming to solve problems that cannot be solved with a normal program. Although extensions of ASP exist that allow solving some of these problems, for practical applications often programmers are forced to resort to writing auxiliary (often procedural) programs, which reduce the task of solving the problem to that of computing the answer sets of a suitable sequence of ASP programs. The rather different level of abstraction used in the auxiliary programs, compared to the ASP programs at the core of the application, causes various difficulties, and makes it hard to prove properties of the overall program. Our approach is based on the consideration that algorithms can be represented by transition systems, and that ASP has been proven to be a useful tool for representing state transitions. By using ASP to encode the auxiliary algorithms needed to solve problems of complexity beyond NP-complete, we remove the problems introduced by the use of different levels of abstraction, and simplify proving the properties of the overall program.

This paper has been focused upon solving problems by calling an NP-complete oracle. However, our definitions extend to more powerful oracles, ultimately allowing to use an ASP machine as an oracle to another ASP machine. Another interesting extension consists in allowing the use of multiple oracles, with the transition program determining which oracles to execute at each transition.

Finally, although here we have mostly focused on finite execution sequences, we believe that infinite execution sequences deserve attention. In fact, infinite execution sequences appear to be useful in the specification of closed-loop algorithms, such as agent control loops. In this area, ASP machines might provide an interesting middle-ground between the fully-logical specifications (e.g. [18]) and the mixed procedural/logical specifications (e.g. [19]) of control loops.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing (1991) 365–385
2. Marek, V.W., Truszczynski, M. The Logic Programming Paradigm: a 25-Year Perspective. In: Stable models and an alternative logic programming paradigm. Springer Verlag, Berlin (1999) 375–398
3. Soininen, T., Niemela, I.: Developing a declarative rule language for applications in product configuration. In: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages. (May 1999)
4. Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence (2006)

5. Baral, C., Chancellor, K., Tran, N., Joy, A., Berens, M.: A Knowledge Based Approach for Representing and Reasoning About Cell Signalling Networks. In: Proceedings of the European Conference on Computational Biology, Supplement on Bioinformatics. (2004) 15–22
6. Son, T.C., Sakama, C.: Negotiation Using Logic Programming with Consistency Restoring Rules. In: IJCAI'09. (2009)
7. Marek, V.W., Remmel, J.B.: On the expressibility of stable logic programming. Journal of Theory and Practice of Logic Programming (TPLP) **3**(4–5) (2003) 551–567
8. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In Doherty, P., McCarthy, J., Williams, M.A., eds.: International Symposium on Logical Formalization of Commonsense Reasoning. AAAI 2003 Spring Symposium Series (Mar 2003) 9–18
9. Buccafurri, F., Leone, N., Rullo, P.: Strong and Weak Constraints in Disjunctive Datalog. In: Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97). Volume 1265 of Lecture Notes in Artificial Intelligence (LNCS). (1997) 2–17
10. Niemela, I., Simons, P., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1–2) (Jun 2002) 181–234
11. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Module Theories: From an Abstract Davis-Putnam-Longemann-Loveland Procedure to DPLL(T). Journal of Artificial Intelligence Research **53**(6) (2006) 937–977
12. Lierler, Y.: Abstract Answer Set Solvers. In: Proceedings of the 24th International Conference on Logic Programming (ICLP08). (Dec 2008) 377–391
13. Turing, A.M.: Systems of logic based on ordinals. Proceedings of the London Mathematical Society. Second Series **45** (1939) 161–228
14. Gelfond, M.: Representing Knowledge in A-Prolog. In Kakas, A.C., Sadri, F., eds.: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. Volume 2408., Springer Verlag, Berlin (2002) 413–451
15. Lifschitz, V., Turner, H.: Representing transition systems by logic programs. In: Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99). Number 1730 in Lecture Notes in Artificial Intelligence (LNCS), Springer Verlag, Berlin (1999) 92–106
16. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 03), Morgan Kaufmann (Aug 2003)
17. Janhunen, T., Niemela, I., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. In: Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference (KR00), Morgan Kaufmann (2000) 411–419
18. Levesque, H.J., Reiter, R., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31** (1997)
19. Balduccini, M., Gelfond, M.: The AAA Architecture: An Overview. In: AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents (AITA08). (Mar 2008)