Organon F 20 (Supplementary Issue 2) 2013: 187-205

# Logic Programming and Interactive Applications[1]

ALEXANDER ŠIMKO

Department of Applied Informatics. Faculty of Mathematics, Physics and Informatics
Comenius University. Mlynská dolina. 842 48 Bratislava. Slovak Republic
simko@ii.fmph.uniba.sk

JOZEF ŠIŠKA

Department of Applied Informatics. Faculty of Mathematics, Physics and Informatics
Comenius University. Mlynská dolina. 842 48 Bratislava. Slovak Republic
siska@ii.fmph.uniba.sk

RECEIVED: 11-12-2012 ▪ ACCEPTED: 02-03-2013

**Abstract**: Answer Set Programming (ASP) is a logic programming based, truly declarative formalism for general purpose problem solving. Its declarative nature allows users to solve problems by defining *what the solutions are* instead of *how to find them*. Complete lack of an imperative component in ASP makes creation of end user applications or integration with other systems demanding. External tools that can process and interpret the output of ASP solvers are needed. To address this issue in the case of simple applications with an input – output interaction loop we introduce a framework for iterative logic applications. Such applications consist of a core logic program that is used to evaluate user actions w.r.t. their current state and to derive a new state of the application. We take care to define the framework in a way that allows it to be used also with other formalism, especially SAT solvers. We also present a web based implementation of such framework for ASP.

**Keywords:** Answer Set Programming – Declarative Problem Solving – Interactive Applications – Logic Programming.

[1] This work was supported by the Slovak national project VEGA no. 1/1333/12.

## 1. Introduction

The aim of this work is to present Logic Programming (LP) and specifically Answer Set Programming (ASP) as a formalism for practical applications that can react to user input and provide appropriate output.

Logic programming presents a robust problem solving and knowledge modelling tool. Rooted in classical logic, it provides an expressive and concise language. Answer set programming, as a truly declarative extension of LP, provides a strong formalism for general purpose problem solving.

The declarative nature of ASP allows users to solve problems by defining *what the solutions are* instead of *how to find them*. Efficient solver implementations make it possible to use such approach for an expanding range of applications.

Complete lack of an imperative component in ASP means that creation of non-trivial end user applications or integration with other systems requires external tools that can process and interpret the output of ASP solvers. Creating and maintaining such support systems can easily be more demanding than creating the logic programs themselves.

To address this issue in the case of simple applications with no external interaction except for a contained input – output interaction loop we introduce a framework for iterative logic applications. Such applications consist of a core logic program that is used to evaluate user actions w.r.t. their current state and to derive a new state of the application. We take care to define the framework in a way that allows it to be used also with other formalism, especially SAT solvers.

We also present an implementation of such framework for ASP. A web based system is used to present the application state to the user and allow him to select actions to be executed. A range of ASP solvers can then be utilized to compute answer sets of the corresponding logic program, which are then used to create a new application state. XSLT stylesheets are used as a declarative way to define how the application state is presented to the user.

This article is organized as follows: in Section 2 we give a brief overview of logic programming and define logic programs and answer sets; in Section 3 we present Answer Set Programming as a fully declarative approach to problem solving and also describe selected ASP solver implementations; in Section 4 we introduce the framework for interactive logic applications and describe its implementation in Section 5.

## 2. Logic Programming

In 1965 Robinson introduced resolution (Robinson 1965) as a method of automated theorem proving that started an era of logic related software. Later, Kowalski and Colmeraurer realized, that *logic can be used as a programming language* (Lloyd 1987). Implication $a,b \Rightarrow c$ can be understood in two ways: (i) declaratively – if $a$ and $b$ have been computed, also $c$ is computed, and (ii) imperatively – in order to compute $c$ we need to first compute $a$ and $b$. Collection of implications without negation (more precisely definite clauses), called *rules,* is understood as a logic program, and resolution is used to answer queries to the program. Programming language *PROLOG* was born.

The original goal of PROLOG was to be a declarative programming language. However, in effort to provide an effective implementation it diverged from that goal (Sterling – Shapiro 1986). PROLOG is sensitive to an order: (i) of the rules in a program, (ii) of the literals in a rule. Later, PROLOG was extended in order to allow the use of negative information. Special type of negation, called *negation as failure,* was allowed in the conditional part of the rule. Semantics of the negation was given procedurally: answer to *not a* is true, if the resolution procedure's answer to *a* is false.

Over time several attempts to give logic programs a declarative semantics appeared. A model-theoretic characterization of semantics of a logic program can be found in (Lloyd 1987). Different approaches to declarative semantics of negation of failure appeared: stratified programs (Apt – Blair – Walker 1988), compilation of logic programs with negation into classical logic (Clark 1977), well-founded models (Gelder – Ross – Schlipf 1988). Finally, stable model semantics (Gelfond – Lifschitz 1988), using reduction to logic programs without negation, was defined. In 1991 it was extended to allow for disjunction and classical negation. *Answer set semantics* (Gelfond – Lifschitz 1991) was born.

In what follows we provide modified version of the definitions of answer set semantics from (Baral 2003).

### *2.1. Syntax*

In this subsection we present the syntax of logic programming.

**Definition 1** (Alphabet). *An* alphabet *of a logic program consists of symbols divided into six classes:*

- *variables,*
- *object constants,*
- *function symbols,*
- *predicate symbols,*
- *connectives "$\neg$", "$\leftarrow$", "not", "or" and comma symbol, and*
- *punctuation symbols "(", ")".*

*Each function and predicate symbol is associated with a natural number, called* arity.

We will follow the convention that object constants, function symbols and predicate symbols start with lowercase, and variables start with uppercase letters.

**Definition 2** (Term). *A* term *is inductively defined as follows:*

- *A variable is a term.*
- *An object constant is a term.*
- *If $t_1, \dots, t_n$ are terms and $f$ is an $n$-ary function symbol, then $f(t_1, \dots, t_n)$ is a term.*

**Definition 3** (Atom). *An* atom *is an expression of the form $p(t_1, \dots, t_n)$ where $t_1, \dots, t_n$ are terms and $p$ is an $n$-ary predicate symbol.*

**Definition 4** (Literal). *A* literal *is an expression of the form $a$ or $\neg a$, where $a$ is an atom.*

**Definition 5** (Rule). *A* rule *is an expression of the form*

$$l_0 \text{ or } \dots \text{ or } l_k \leftarrow l_{k+1}, \dots, l_m, \text{ not } l_{m+1}, \dots, \text{not } l_n \tag{1}$$

*where* $0 \leq \mathrm{k} \leq m \leq n \in \mathbb{N}$ *and each $l$ is a literal.*

*A rule with $n = 0$ is called* fact.

**Notation 1.** *Let $r$ be a rule of the form 1. Then*

- *$l_0$ or $\dots$ or $l_k$ is called the* head *of the rule $r$, and we use $head(r)$ to denote the set $\{l_0, \dots, l_k\}$,*
- *$l_{k+1}, \dots, l_m$, not $l_{m+1}, \dots,$ not $l_n$ is called the* body *of $r$ and we denote it by $body(r)$,*
- *$l_{k+1}, \dots, l_m$ is called the* positive body *of $r$, and we use $body^+(r)$ to denote the set $\{l_{k+1}, \dots, l_m\}$,*

- *not* $l_{m+1}, \ldots,$ *not* $l_n$ *is called the* negative body *of r, and we use* $body^{-}(r)$ *to denote the set* $\{l_{m+1}, \ldots, l_n\}$.

Logic programming uses two kinds of negation:

- explicit negation $\neg a$ meaning that an agent believes $a$ is false,
- default negation *not a* meaning that an agent does not believe $a$ is true, but it does not necessary mean the agent believes $a$ is false.

Moreover, logic programming uses the connective "or" which differs from the connective "$\vee$" from the classical logic. Informally, *a or b* means an agent believes $a$ is true or believes $b$ is true, as opposed to $a$ is true or $b$ is true. In the classical logic, $a \vee \neg a$ is entailed by each theory. On the other hand, *a or* $\neg a$ is not entailed by each logic program (Gelfond – Kahl 2012).

**Definition 6** (Language). *A* language *given by an alphabet consists of all the rules constructed from the symbols of the alphabet.*

**Definition 7** (Logic program). *A* logic program *over a language is a finite set of rules.*

**Definition 8** (Ground term, atom, literal, rule). *A term (atom, literal, rule) is called* ground *iff it does not contain variables.*

In logic programming, a rule with variables is viewed as a shorthand for all its ground instances, i.e. all rules obtained by replacing the variables by ground terms.

**Definition 9** (Grounding). *Let* $\mathcal{L}$ *be a language, and r be a rule.*

*The* grounding *of r in* $\mathcal{L}$*, denoted as* $ground(r,\mathcal{L})$ *is the set of all the rules obtained from r by all possible substitutions of ground terms from* $\mathcal{L}$ *for the variables in r.*

*Let P be a logic program. Then* grounding *of P in* $\mathcal{L}$ *is* $ground(P,\mathcal{L}) = \bigcup_{r \in P} ground(r,\mathcal{L})$.

When a logic program $P$ is given without any explicit language, we assume the implicit language containing exactly the symbols from the program $P$. We denote this language by $\mathcal{L}_P$. Then $ground(P)$ denotes $ground(P,\mathcal{L}_P)$.

Semantics of a logic program $P$ is given by a semantics of $ground(P)$.

### *2.2. Semantics*

Answer set semantics assigns to each logic program a set of answer sets – alternative belief sets a rational agent might accept. Rationality is given by the principles (Gelfond – Kahl 2012):

- if an agent believes in the body of a rule, it must believe in the head of the rule,
- an agent does not believe in contradictions,
- an agent believes nothing it is not forced to believe

In definition of answer set semantics we will work with grounded programs. Definition is divided into two parts. First, a semantics is defined for positive programs – programs without default negation. Then the definition is extended to the general case.

**Definition 10** (Consistency). *A set $S$ of ground literals is* consistent *iff it does not contains literals $a$ and $\neg a$, where $a$ is an atom.*

**Definition 11** (Satisfaction). *Let $S$ be a set of ground literals. Let $r$ be a rule. $S$ satisfies*

- *$body(r)$ iff $body^{+}(r) \subseteq S$ and $body^{-}(r) \cap S = \emptyset$,*
- *$head(r)$ iff $head(r) \cap S \neq \emptyset$,*
- *$r$ iff $S$ satisfies $head(r)$ if it satisfies $body(r)$.*

*$S$ satisfies a logic program $P$ iff $S$ satisfies each $r \in P$.*

**Definition 12** (Positive program). *A logic program $P$ is called* positive *iff $body^{-}(r) = \emptyset$ for each $r \in P$.*

**Definition 13** (Answer sets of a positive program). *A consistent set of ground literals $S$ is an answer set of a positive logic program $P$ iff $S$ is a subset-minimal set of literals that satisfies $P$, i.e. there is no subset $S' \subset S$ that satisfies $P$.*

The definition of answer sets for the general case is non-constructive. First, a candidate for an answer set is guessed. Then it is tested, whether it is stable. First, the rules with unsatisfied negative bodies are removed. Since the rules left have satisfied negative bodies, the negative bodies are also removed. The resulting program is positive, and it's semantics is already defined. The answer set guess is stable iff it is an answer set of the reduced program.

**Definition 14** (Gelfond-Lifschitz reduction). *Let $P$ be a logic program and $S$ be a set of literals.*

*Then,* reduction *of $P$ w.r.t. $S$, denoted $P^S$, is the set $\{head(r) \leftarrow body^+(r) : r \in P$ and $body^-(r) \cap S = \emptyset\}$.*

**Definition 15** (Answer sets). *Let $P$ be a logic program, and $S$ be a set of literals.*

*$S$ is an answer set of $P$ iff $S$ is an answer set of $P^S$.*

Answer set semantics enjoys the following nice property.

**Proposition 1** (Exclusive support). *Let $P$ be a logic program, $S$ an answer set of $P$, and $l$ be a literal.*

*If $l \in S$, then there is a rule $r \in P$ such that:*

- *$S$ satisfies $body(r)$, and*
- *$head(r) \setminus \{1\} \not\subseteq S$.*

**Definition 16** (Entailment). *Let $P$ be a logic program, and $l$ be a literal.*

*$l$ is entailed by a program $P$, denoted $P \models l$ iff $l \in S$ for each answer set $S$ of $P$.*

*$Cn(P) = \{l : P \models l\}$ will denote the set of all the consequences of $P$.*

**Proposition 2**. *Cn operator is non-monotonic, i.e. it does not hold that for each programs $P_1$, $P_2$ such that $P_1 \subseteq P_2$ we have that $Cn(P_1) \subseteq Cn(P_2)$.*

**Example 1**. *Consider the programs $P_1$:*

$$a \leftarrow not\ b$$

*and $P_2$:*

$$\begin{array}{lll} a & \leftarrow & not\ b \\ b & \leftarrow & \end{array}$$

*We have that $Cn(P_1) = \{a\}$ and $Cn(P_2) = \{b\}$. Hence $P_1 \subseteq P_2$, but $Cn(P_1) \not\subseteq Cn(P_2)$.*

A logic program is not guaranteed to have an answer set.

**Example 2**. *Consider the program $P$*

$r_1 : a \quad \leftarrow$

$r_2 : inc \leftarrow not\ inc,\ a$

| $S$ | $P^S$ | *answer sets of* $P^S$ |
|---|---|---|
| $\emptyset$ | $a \leftarrow$ , $inc \leftarrow a$ | $\{a, inc\}$ |
| $\{a\}$ | $a \leftarrow$ , $inc \leftarrow a$ | $\{a, inc\}$ |
| $\{inc\}$ | $a \leftarrow$ | $\{a\}$ |
| $\{a, inc\}$ | $a \leftarrow$ | $\{a\}$ |

*For each answer set candidate $S$ we have that $S$ is not an answer set of $P^S$. Therefore $P$ has no answer set.*

*The rule $r_2$ works here as an integrity constraint. It causes each answer set candidate $S$ such that $a \in S$ to be eliminated.*

**Definition 17** (Integrity constraint). *An expression of the form*

$$\leftarrow l_1, \ldots, l_m, \ not \ l_{m+1}, \ldots, not \ l_n \tag{2}$$

*is called an* integrity constraint.

**Notation 2.** *Let $c$ be an integrity constraint of the form 2. By $body^+(c)$ we denote the set $\{l_1, \ldots, l_m\}$, and by $body^-(c)$ we denote the set $\{l_{m+1}, \ldots, l_n\}$.*

**Definition 18** (Violation). *We say that a set of literals $S$ violates a ground integrity constraint $c$ iff $body^+(c) \subseteq S$, and $body^-(c) \cap S = \emptyset$.*

An integrity constraint is used to eliminate an answer set candidate that violates the integrity constraint.

Given a logic program $P$, an integrity constraint $c$ is understood as a shorthand for the rule $inc \leftarrow not \ inc, body(c)$, where $inc$ is a new literal present in neither $P$ nor $c$.

## 3. Declarative problem solving

*Answer set programming* (Marek 1999, Niemelä 1999) has emerged as a declarative problem solving paradigm using logic programming under answer set semantics. A logic program is written in a way, that its answer sets correspond to the solutions of a given problem (Figure 1). A logic program is usually written in a *generate-and-test* way – it is divided into three parts:

- *instance,* encoding the problem's instance,
- *generator,* whose answer sets correspond to solution candidates, and
- *tester,* that eliminates candidates that are not solutions.

**Example 3.** *Consider the classical problem of putting N chess queens on the $N \times N$ board in a way that no two queens attack each other.*

*A generator could be written as follows:*

$$g_1 : on(X, Y) \leftarrow d(X), d(Y), not\ \neg on(X, Y)$$
$$g_2 : \neg on(X, Y) \leftarrow d(X), d(Y), not\ on(X, Y)$$

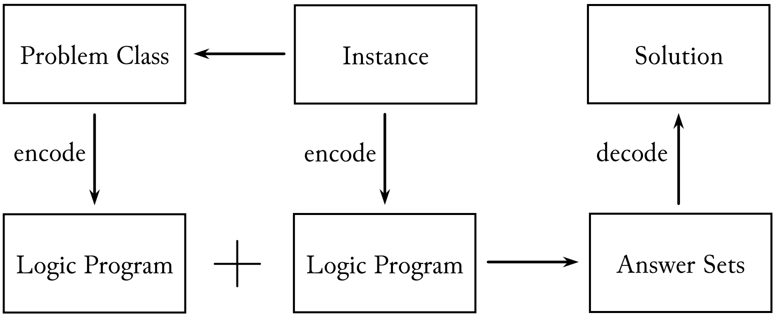

Figure 1: Answer Set Programming

*$d(X)$ means that $X$ is a valid row/column number. $on(X, Y)$ means that a queen is on the position $(X, Y)$.*

*Together with the encoding of problem instance $N = 2$*

$$i_1 : d(1) \leftarrow$$
$$i_2 : d(2) \leftarrow$$

*it has 16 answer sets. The following sets*

$$S_1 = \{d(1), d(2), \neg on(1,1), \neg on(1,2), \neg on(2,1), \neg on(2,2)\}$$
$$S_2 = \{d(1), d(2), on(1,1), \neg on(1,2), on(2,1), \neg on(2,2)\}$$
$$S_3 = \{d(1), d(2), on(1,1), on(1,2), on(2,1), on(2,2)\}$$

*are three of them.*

*A tester could be written as follows:*

$$t_1 : \leftarrow on(X, Y1), on(X, Y2), Y \neq Y2$$
$$t_2 : \leftarrow on(X1, Y), on(X2, Y), X \neq X2$$
$$t_3 : \leftarrow on(X1, Y1), on(X2, Y2), X \neq X2, Y \neq Y2, |X - X2| = |Y - Y2|$$

$$
\begin{array}{lcl}
t_4 : hasq(X) & \leftarrow & on(X,Y) \\
t_5 : & \leftarrow & d(X), not\ hasq(X)
\end{array}
$$

*Expressions of the form $X = Y$ and $X \neq Y$ are not part of the syntax of rules. They are understood as conditions for grounding.*

*The integrity constraint $t_1$ eliminates the answer set candidates, in which two queens are at the same column. The integrity constraint $t_2$ eliminates the answer set candidates, in which two queens are at the same row. The integrity constraint $t_3$ eliminates the answer set candidates, in which two queens are at the same diagonal. The integrity constraints $t_4$, $t_5$ check whether $N$ queens are used.*

*The tester eliminates all the answer set candidates for the presented instance as the 2 – queen problem has no solution.*

Answer set programming is well suited for constraint satisfaction problems. A constraint satisfaction problem is given by a set of variables, each associated with the domain, and a set of constraints. The task is to find a variable assignment such that all constraints are satisfied – constraints put conditions on variable assignment. A constraint satisfaction problem can be directly solved using generate-and-test approach of answer set programming. Generator generates answer set candidates, where each candidate represents one variable assignment. Constraints of the constraint satisfaction problem are then transformed into the rules and integrity constraints of a tester, which tests whether an assignment is a valid one.

Many real world problems can be understood as a constraint satisfaction problem, e.g. design of a computer configuration given a set of requirements (which can be seen as the constraints), construction of a plan of a given length etc.

Moreover answer set programming is not only restricted to constraint satisfaction problem. It can be easily used to solve problems from NP complexity class, given a problem is defined in a guess-and-test manner: (i) first, a candidate is guessed, and (ii) then it is tested whether it is a solution to the problem. Again, generator is used to generate answer set candidates, each representing a problem solution candidate, and tester performs the test.

Answer set programming in not the only option for such tasks. Alternative approach is for example to encode a problem using propositional logic in a way that the models of the theory represent the problem solutions. Afterwards a SAT solver is employed to compute the solutions. However, answer set programming has a big advantage over SAT. In a logic program, we can divide a program into two parts: (i) the first one representing an instance of the problem class, and (ii) the second, describing solutions of the problem class in general, without knowing an instance. On the other hand, when using propositional logic, we cannot separate a theory in this way. Hence, in order to use both SAT solvers and modular representation of a problem, we must use some other language and use a transformation to propositional logic.

## *3.1. ASP solvers*

An ASP solver is a computer program that accepts a logic program on its input, and provides its answer sets on its output.

Inspired and built upon the success of SAT solvers, many ASP solvers were implemented, and many optimization techniques were developed. We mention the most prominent ones, namely *smodels* (http://www.tcs.hut.fi/Software/smodels/), *clasp* (Gebser et al. 2007), *claspD* (Drescher et al. 2008), and *DLV* (http://www.dlvsystem.com/dlvsystem/). *Clasp* and *smodels* do not allow disjunction in the heads of rules, while *claspD* and *DLV* do. The reason is a higher computational complexity of disjunctive logic programs (Baral 2003).

The *smodels* solver uses a two step computation process. First, an input program is processed by a parser. It parses the input program and produces a machine readable format usable by the *smodels* solver. It also performs grounding of the input program, whereby the variables in rules are replaced by ground literals. A simplification of the ground program is also performed and the result is passed to the solver.

Historically, *lparse* was the first parser that was used together with the *smodels* solver. Later, the *gringo* parser and *clasp* solver were developed. To provide compatibility, *gringo* and *clasp* support the input language of *lparse* and *smodels*.

On the other hand, the *DLV* solver works directly on programs with variables, and does not require an external parser. However, it does not support function symbols.

In addition to the syntax presented in Section 2.1, these solvers support many extensions, such as:

- *aggregates* – allow for example to determine the number of literals satisfying certain condition,

- *optimization statements* – allow to compute only minimal answer sets w.r.t. a given criterion,
- *choice rules* – enable to express generators in a compact and readable form.

## 4. Interactive logic applications

In this section we present a framework for interactive applications based on logic (model based) formalisms. We start with a simple generalization of such formalisms and then use it to define an application, its states and (iterative) execution.

Although we are concerned mainly with logic programs, any formalism with a model-theoretic semantics and an appropriate implementation can be used to create the kind of applications we present in this work. Our only requirement is that the formalism defines a semantics that assigns models (answer sets, solutions) to programs (theories).

**Definition 19** (LP Formalism). *A* Logic Programming Formalism *is a triple* $(\mathcal{P}, \mathcal{M}, \mathrm{Sem})$, *where* $\mathcal{P}$ *is a (non-empty) set of possible programs,* $\mathcal{M}$ *is a (non-empty) set of possible models and* $\mathrm{Sem} : \mathcal{P} \rightarrow 2^{\mathcal{M}}$ *is a function that assigns sets of models to programs.*

Throughout this work we consider Answer Set Programming as our intended formalism, $\mathcal{P}$ thus being the set of all logic programs and $\mathcal{M}$ the set of all possible answer sets. ASP solvers provide direct means of implementation and the expressivity of logic programs allows the creation of applications with very little need of external processing.

Propositional logic also provides applicable formalisms with implementations in the form of SAT solvers. However, the restrictions on propositional formulae and the restrictive input formats used in SAT solvers require additional layers of pre- and post-processing for any meaningful applications. To show that it is possible to cover even such cases with our framework we also include a few remarks about propositional logic where relevant.

The goal of an interactive application is the repetitive execution and evaluation of user actions against the current state of the application. The application framework is responsible for the visualisation of the application's state along with the allowed actions and for the calculation of a new state. Because the user can choose from multiple actions, which can have non-deterministic consequences, there can be various possible outcomes in each states. The application can thus be either linearly *executed* or the relationships of application states and actions can be studied in the form of an *execution graph.*

We start by defining the application itself. In its simplest form, the application consists of a main program (theory), input, output and presentation function. The presentation function interprets a current state of the application, presents it to the user and returns his chosen action(s). The input function combines the main program with all this information into a program in the used LP Formalism. Semantics of the formalism is then used to find the models of this program and (if possible) one of them is selected. The output function is used to convert such resultant model into a new application state.

**Definition 20** (Interactive application). *Let* $(\mathsf{P}, \mathsf{M}, \mathrm{Sem})$ *be an LP Formalism, let* $\mathsf{P}^A$ *be a set of application programs. An* interactive application *over* $(\mathsf{P}, \mathsf{M}, \mathrm{Sem})$ *is a tuple*

$$A = (P, S, \mathsf{A}, In, Out, User, s_0)$$

*where*

1. *$P \in \mathsf{P}^A$ represents the main application program,*
2. *$S$ is a non-empty set of possible application states,*
3. *$\mathsf{A}$ is a set of possible user actions,*
4. *$In : \mathsf{P}^A \times S \times 2^{\mathsf{A}} \to \mathsf{P}$ is an input transformation,*
5. *$Out : \mathsf{M} \to S$ is an output transformation,*
6. *$User : S \times \mathbb{N} \to 2^{\mathsf{A}}$ is a function that represents user input at a specific time.*

**Definition 21** (Application Instance). *An* application instance *is a tuple* $(A, s)$ *where* $A = (P, S, \mathsf{A}, In, Out, User, s_0)$ *is an application and* $s \in S$ *is a state. We say that* $(A, s)$ *is an instance of* $A$ *with an actual state* $s$.

To abstract from the structure of the programs as much as possible, we delegate all responsibility for state and action transformations and their incorporation into the main program to the input transformation function. In the case of logic programming, as well as many other formalisms, this can be easily reduced to the transformation of the application state into a logic program ($In^S$) where the input transformation itself can then be given as

$$In = \mathsf{P} \cup In^S(s) \cup User(s, i).$$

However even in the case of logic programming the more general definition of *In* allows some interesting possibilities that we talk about in section 6.

Also, because we do not require *P* to actually be a program from $\mathsf{P}$, this allows us to easily do various kinds of preprocessing on *P*. To implement propositional logic based applications through the use of SAT solvers, *P* can be a set of formulae schemes that are appropriately translated into a set of propositional formulae by *In* based on the actual state of the application.

To obtain the next state when executing an application we simply apply the semantics to the results of the input transformation *In*. Because we allow the semantics to assign multiple models to a program (representing possible outcomes of user's actions), we arbitrarily select a single one of them. This model is then transformed into the application's new state using the *Out* transformation.

**Definition 22** (Application Iteration). *Let* $(A, s_i)$ *be an application instance at time* $i \in \mathbb{N}$. *The next state of the instance is*

$$s_{i+1} = Out\left(Sel\left(\mathrm{Sem}\left(In\left(P, s_i, User\left(s_i, i\right)\right)\right)\right)\right)$$

*where* $Sel : 2^{\mathsf{M}} \to \mathsf{M}$ *is a selection function that selects an arbitrary model from a set of models or returns a special (i.e. empty) model if the set is empty.*

**Example 4**. *Let us consider a simple two player game: each player has a switch. At every step each player can either flip his switch or leave it as it is. The first player wins when exactly one of the switches is turned off.*

*We can formulate this game as an interactive application (over logic programs) in which the user plays against the computer:*

$$\begin{array}{rcl} S & = & 2^{\{on(u),\ on(c)\}} \\ \mathsf{A} & = & \{flip(u), pass(u)\} \\ User(s, i) & = & \begin{cases} \{flip(u)\} \\ \{pass(u)\} \end{cases} \\ In(P, s, a) & = & P \cup s \cup \{a\} \\ Out(M) & = & \{on(X) \mid next(X) \in M\} \\ s_0 & = & \emptyset \end{array}$$

*Possible application states are subsets of* $\{on(u), on(c)\}$*, where* $on(u)$ $(on(c))$ *represents that the switch belonging to the user (computer player) is turned on. There are two possible actions for the users: flip it* ($flip(u)$) *or leave it be* ($pass(u)$)*. The User function presents the current application state to the users and returns one of these actions based on his choice.*

*The input transformation adds the current state and user's action to the program as facts. The output transformation creates a new state based on the presence of the predicate next in the selected answer set.*

*The main (logic) program P of the application is formulated as follows:*

$$\begin{aligned}
flip(c) &\leftarrow \text{not } pass(c)\\
pass(c) &\leftarrow \text{not } flip(c)\\
next(X) &\leftarrow on(X), \text{not } flip(X)\\
next(X) &\leftarrow \text{not } on(X), flip(X)
\end{aligned}$$

*The first two rules generate two answer sets that correspond to the respective moves of the computer player. The last two rules evaluate the results of the actions.*

## 4.1. Application analysis

In addition to simple execution of applications our framework allows another option: a formal way to define and study the interaction of application states and user actions. Transitional properties of a specific application can be visualized through a graph representing the relations between application states and actions. Various properties of such graph or its subpart can then be studied.

**Definition 23** (Execution graph). *Let* $\mathcal{A} = (P, S, \mathsf{A}, In, Out, User, s_0)$ *be an application. A (complete)* execution graph *for* $\mathcal{A}$ *is a graph* $G_{\mathcal{A}} = (V, E)$ *with* $V = S \cup \mathsf{P}$ *and a set of labeled edges E such that*

$$\begin{aligned}
(s, Q, a) \in E &\quad \textit{iff} \quad In(P, s, a) = Q\\
(Q, s, M) \in E &\quad \textit{iff} \quad M \in Sem(Q) \wedge Out(M) = s
\end{aligned}$$

**Example 5.** *Let us consider the application (game) from example 4. Figure 2 depicts the complete execution graph for this application. Circled vertices represent states while the rest represent actual programs (without the main program P) with u and c instead of* $on(u)$ *and* $on(c)$ *respectively.* $flip(u)$ *and* $pass(u)$ *are also abbreviated as* $f(u)$ *and* $p(u)$*. Each vertex representing a state has two outgo-*

*ing edges: each for one of the user's actions. Each vertex representing a program has also two outgoing edges representing the two possible answer sets generated by the first two rules of P that encode the computer moves.*

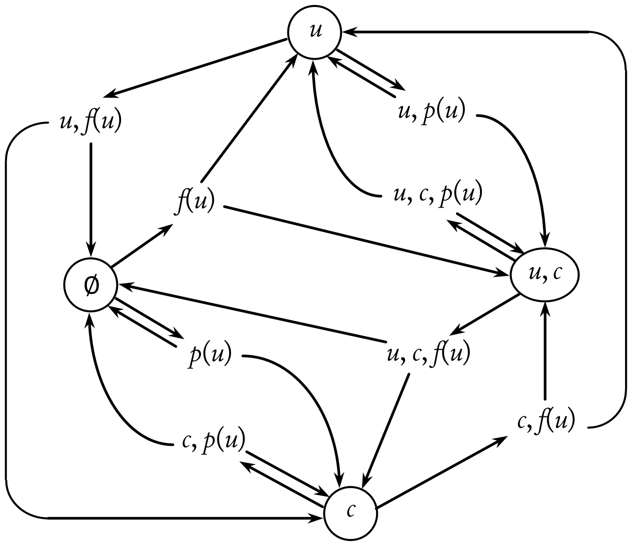

Figure 2: The execution graph for application from example 5

A complete execution graph is a tool that allows theoretical study of an application. When working with actual applications it often makes more sense to consider only a finite subset of the execution graph accessible from the initial state. This can represent the so far explored options of the application execution.

**Definition 24** (Partial execution graph). *A* partial execution graph $(V', E')$ *for an application A is a connected sub-graph of the (complete) execution graph for A such that*

- *$(V', E')$ contains $s_0$,*
- *for each vertex $v \in V'$ there is a (oriented) path from $s_0$ to $v$,*
- *for each vertex $v \in \mathsf{P}$ it contains all its outgoing edges from the execution graph.*

*A* trace *for an instance $(A, S)$ is a path $h_0, h_1, \ldots, h_n$ in a partial execution graph for A such that $h_0 = s_0$ and $h_n = s$.*

## 5. Implementation

The interactive application framework described in the previous section was implemented as a Python web application (ILPA. A system for interactive logic applications. http://dai.fmph. uniba.sk/~siska/ilpa/). It allows users to create applications composed from logic programs, to define input and output transformations and to create user interfaces (presentation functions) using XSLT stylesheets.

The implementation is based on a configurable backend that allows the usage of various LP implementations such as SMODELS (http://www.tcs.hut.fi/Software/smodels/), DLV (http://www.dlvsystem.com/dlvsystem/) or clasp (Gebser et al. 2007). The system also implements basic preprocessing options as well as a simple modularization framework (Šiška 2011).

Application states are represented as sets of atoms and usually serialized as a XML document. A configurable framework of atom filters and translations can be used to define input and output transformations, however the input transformation only translates the state into a set of facts and then simply joins them with the main program and user actions (passed as is from the presentation function). A XSLT transformation can be used as general-purpose filter that allows greater flexibility when translating states.

The presentation function is implemented as a XSTL template that transforms the state (represented as a XML document) into HTML code that is presented to the user. Special hyperlinks can be generated, that are then interpreted as user's actions.

Users can edit logic programs and XSLT stylesheets stored on the server, as well as define applications and create and execute their instances. In addition to simple linear execution of an application, the user can also explore the (expanding) partial execution graph of an instance.

## 6. Conclusion

We presented Answer Set Programming as a very potent formalism for general purpose problem solving. Its declarative nature allows users to search for solutions be defining *what they are* instead of *how to find them*. Various efficient solver implementations provide the means to use ASP in an expanding range of applications.

To reduce the amount of additional software and tools required when integrating ASP solutions into other applications we introduced a framework for iterative, logic based applications that can use LP as their main (and only) language. We defined an *interactive application,* its instance and execution step (iteration). We presented our implementation of this framework that allows users to create their own application and to run them using a web based interface.

In order not to restrict this framework only to logic programs and the answer set semantics, we assumed a generalized formalism with a model-theoretic semantics as the fundamental building block of the framework. Together with flexible definitions of input and output transformations this allows the use of various other formalism such as SAT solvers for proposition logic.

The flexibility of the input transformation allows also the creation of applications that actually modify the executed program on the fly. The input transformation can introduce new rules into the final program based on current application state or even remove rules from the main program. This can lead to the creation of evolving applications that change their behaviour based on the history of actions. It would be interesting to further study such approach and to compare it with similar work such as Evolving Logic Programming (Alferes et al. 2002).

## References

ALFERES, J. J. – BROGI, A. – LEITE, J. A. – MONIZ PEREIRA, L. (2002): Evolving logic programs. In: *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*. London> Springer-Verlag, 50-61.

APT K. R. – BLAIR, H. A. – WALKER, A. (1988): *Foundations of Deductive Databases and Logic Programming*. Chapter Towards a theory of declarative knowledge. San Francisco: Morgan Kaufmann Publishers Inc., 89-148.

BARAL, C. (2003): *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

CLARK, K. L. (1977): Negation as failure. In: *Logic and Data Bases*. 293-322.

DRESCHER, C. – GEBSER, M. – GROTE, T. – KAUFMANN, B. – KÖNIG, A. – OSTROWSKI, M. – SCHAUB, T. (2008): Conflict-driven disjunctive answer set solving. In: Brewka, G. – Lang, J. (eds.): *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*. AAAI Press, 422-432.

GEBSER M. – KAUFMANN B. – NEUMANN, A. – SCHAUB, T. (2007): Conflict-driven answer set solving. In: Veloso, M. (ed.): *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJ- CAI'07)*. AAAI Press/The MIT Press, 386-392. Available at http://www.ijcai.org/papers07/contents.php.

VAN GELDER A. – ROSS, K. A. – SCHLIPF, J. S. (1988): Unfounded sets and well-founded semantics for general logic programs. In: Edmondson-Yurkanan, C. – Yannakakis, M. (eds.): *PODS*. ACM, 221-230.

GELFOND, M. – KAHL, Y. (2012): *Knowledge Representation, Reasoning, and the Design of Intelligent Agents*.

GELFOND, M. – LIFSCHITZ, V. (1988): *The Stable Model Semantics for Logic Programming*. MIT Press, 1070-1080.

GELFOND, M. – LIFSCHITZ, V. (1991): Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 365-385.

LLOYD, J. W. (1987): *Foundations of Logic Programming*. New York: Springer-Verlag.

MAREK, V. W. (1999): Stable models and an alternative logic programming paradigm. In: *In The Logic Programming Paradigm: a 25-Year Perspective*. Springer-Verlag, 375-398.

NIEMELÄ, I. (1999): Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.* 25, Nos. 3-4, 241-273.

ROBINSON, J. A. (1965): A machine-oriented logic based on the resolution principle. *J. ACM* 12, No. 1, 23-41.

ŠIŠKA, J. (2011): Declarative description of module dependencies in logic programming. In: *Znalosti 2011*.

STERLING, L. – SHAPIRO, E. (1986): *The Art of Prolog*. Cambridge (MA): MIT Press.