Integrating Answer Set Programming with Object-oriented Languages[⋆]

Jakob Rath and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
jakob.rath@student.tuwien.ac.at, redl@kr.tuwien.ac.at

Abstract. Answer Set Programming (ASP) is a declarative programming paradigm which allows for easy modeling and solving of hard problems that are often cumbersome to implement in object-oriented programming languages. It was successfully applied to a range of applications from artificial intelligence, such as combinatorial or scheduling problems. On the other hand, real-world applications for end-users usually consist also of components which cannot be (easily) solved in ASP, such as user interaction via graphical user interfaces, presentation of results, and interfaces to data sources. Instead, realizing such components is typically in the domain of traditional (object-oriented) programming languages. To address this issue, we introduce a *language* which allows for a *formal specification of the input and output of an ASP program*, which can be exploited to easily interface the program from object-oriented languages using a dedicated library. While the language is independent from the concrete object-oriented language, we also provide and present a reference implementation as a Python library. We then discuss some applications which can be realized on top of our approach.

Keywords: Answer Set Programming, Nonmonotonic Reasoning, Interface to Object-oriented Languages

1 Introduction

Answer Set Programming (ASP) is a declarative programming paradigm based on nonmonotonic reasoning and a multi-model semantics [7]. The problem at hand is encoded as an ASP program in such a way that its models, called *answer sets*, correspond one-to-one to the solutions of the problem. Thanks to disjunction and default negation, the formalism has a high expressiveness, and thanks to various language extensions such as aggregates, many problems from artificial intelligence such as combinatorial and scheduling problems can be encoded in ASP in an intuitive way. In contrast, solving such problems in traditional object-oriented programming languages is often cumbersome as an algorithm needs to be specified. ASP has also been successfully applied to various real-world applications from industry, e.g. *workforce management* [12] and *automatic suggestion of holiday plans for tourists* [9]; for further examples we refer to [8].

[⋆] This research has been supported by the Austrian Science Fund (FWF) project P27730.

However, typical applications for end-users also contain components which cannot be (easily) solved in ASP, but their realization is rather in the domain of traditional object-oriented languages. These components include, for instance, graphical user interfaces, presentation of results, and interfaces to data sources. As a concrete example, consider a packing problem which needs to be solved by employees of a logistics company, such as to distribute a set of goods to a minimum number of trucks under given side constraints. While the core problem is a typical use case for ASP, if it occurs as part of a real application, data needs to be imported from databases, parameters need to be entered by the user, and the results must be further processed by other system components, such as for accounting purposes. In ASP, the input is specified via facts and the output is presented as answer sets. However, since typical users of such an application are no computer scientists and are not used to read and write formal notations, more appropriate interfaces must be developed. Moreover, even if users are used to ASP programs, a manual transfer of data between the ASP program and other components is cumbersome. Instead, this should be transparent from the user. Hence, an interface between the ASP program and other components is needed.

An ad hoc solution when developing an application is to implement such an interface from scratch. To this end, facts are generated and piped to the ASP solver, which computes its answer sets that are then parsed and transformed into objects. However, while the details of generating facts and transforming the answer sets to objects depend on the application, it seems that these steps are similar in most cases. This calls for a generic interface which can be instantiated depending on the application at hand.

To address this issue, we present a language for ASP which allows the programmer to *annotate* ASP programs with *specifications of their input and output*. Based on these annotations, the ASP program can then be used from the object-oriented code similarly to modules by sending input to it and retrieving its answer sets in form of objects. We specify the language independently of the concrete object-oriented language and the ASP solver at hand. Instead, the formalism can be instantiated for arbitrary languages resp. solvers which provide a certain minimum set of features. However, we also provide an implementation of this language for Python, using the DLVHEX solver [11] as solver backend.

Unlike existing approaches such as JASP [4] and EmbASP [5], our system uses annotations of the ASP program rather than embeddings of the ASP program into the object-oriented code, and the input and output is specified in a language-independent manner. This has the advantage that the program is more independent of the remaining components of the application, which allows for easier adoption or integration into multiple applications (which might even be implemented in different programming languages), similarly to modules in software engineering. As a further difference to some existing approaches such as [10], which modifies the ASP language by providing access to objects defined in the object-oriented code, our language does *not modify* but rather *extend* ASP in a conservative way using annotations, i.e., all annotated programs in our system are still ordinary ASP programs and can also be used independently.

The structure of the remaining part of the paper is as follows:

- In Section 2 we recapitulate the syntax and semantics of ASP.
- In Section 3 we introduce the language for specifying the input and output of ASP programs and illustrate it with examples.

– In Section 4 we present our prototypical implementation PY-ASPIO (ASP Interface to Object-oriented programs) in Python.
– In Section 5 we discuss possible real-world applications in more detail.
– In Section 6 we discuss related work and point out differences to ours.
– In Section 7 we conclude and give an outlook on possible future work.

2 Preliminaries

We briefly recapitulate Answer Set Programming (ASP) [7], and refer to [1] for a more in-depth overview of the field of ASP. Our alphabet consists of possibly infinite sets of constant symbols $\mathscr{C}$ (including all integers), variables $\mathscr{V}$, function symbols $\mathscr{F}$, and predicate symbols $\mathscr{P}$. We assume that $\mathscr{V}$ is disjoint from all other sets, while symbols may be shared between the other sets. We let the set of terms $\mathscr{T}$ be the least set such that $\mathscr{C} \subseteq \mathscr{T}$, $\mathscr{V} \subseteq \mathscr{T}$, and $f \in \mathscr{F}$, $T_1, \dots, T_\ell \in \mathscr{T}$ implies $f(T_1, \dots, T_\ell) \in \mathscr{T}$. An (ordinary) atom is of form $p(t_1, \dots, t_\ell)$ with predicate symbol $p \in \mathscr{P}$ and terms $t_1, \dots, t_\ell \in \mathscr{T}$, abbreviated as $p(\mathbf{t})$; we write $t \in \mathbf{t}$ if $t = t_i$ for some $1 \leq i \leq \ell$. A term resp. atom is called *ground* if it does not contain variables. A (default) literal is either an atom a or a default-negated atom $\text{not}\, a$.

Definition 1. *An* answer set program P *consists of rules*

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \dots, b_m, \text{not } b_{m+1}, \dots, \text{not } b_n \ , \qquad (1)$$

where each a_i and each b_j is an atom. A non-disjunctive rule with empty body (i.e., $k = 1$ and $n = 0$) is called a fact.

For such a rule r we let $H(r) = \{a_1, \dots, a_k\}$ be its *head*, $B^+(r) = \{b_1, \dots, b_m\}$ be its *positive body* and $B^-(r) = \{b_{m+1}, \dots, b_n\}$ be its *negative body*. A rule resp. program is ground if it contains only ground atoms.

An interpretation I is a subset of the set of atoms $A(P)$ occurring in the ground program P at hand, where $a \in I$, also denoted $I \models a$, expresses that a is true and $a \notin I$, also denoted $I \not\models a$, that a is false. Conversely, a negated literal $\text{not}\, a$ is satisfied under I, denoted $I \models \text{not}\, a$, if $I \not\models a$, and it is unsatisfied, denoted $I \not\models \text{not}\, a$, otherwise. A ground rule r of form (1) is satisfied under I, denoted $I \models r$, if $a_i \in I$ for some $1 \leq i \leq k$, or $b_i \notin I$ for some $1 \leq i \leq m$, or $b_i \in I$ for some $m + 1 \leq i \leq n$. A ground program P is satisfied under I, denoted $I \models P$, if each $r \in P$ is satisfied under I. A set of literals L is satisfied under I, denoted $I \models L$, if $I \models l$ for all $l \in S$.

The answer sets of a ground program P are defined using the *(GL-)reduct* [7] $P^I = \{H(r) \leftarrow B^+(r) \mid r \in P, I \not\models b \text{ for all } b \in B^-(r)\}$ of P wrt. an interpretation I.

Definition 2. *An interpretation I is an answer set of a ground program P, if I is a $\subseteq$-minimal model of P^I.*

Example 1. Consider the program $P = \{a \leftarrow \text{not}\, b;\ b \leftarrow \text{not}\, a\}$. Its answer sets are $I_1 = \{a\}$ and $I_2 = \{b\}$. □

We let $AS(P)$ be the set of answer sets of P. The answer sets of a program P with variables are given by the answer sets of its *grounding* $grnd(P)$, which results from P if all variables are replaced by all terms in all possible ways. Throughout the rest of the paper we assume that suitable safety conditions on P guarantee that $grnd(P)$ is finite.

3 Specifying the External Interface of ASP Programs

In this section we present a language which allows for specifying the interfaces of ASP programs in order to use them from object-oriented code. It comprises of the *input specification*, which declares what input arguments are expected and how they are mapped to ASP facts, and the *output specification*, which defines how the answer sets are mapped back to objects. The language is realized by conservative *annotations* added to the program, while the rules remain in ordinary ASP syntax and thus can also be used stand-alone. Each program has exactly one input and exactly one output specification.

In view of the implementation (cf. Section 4), such an ASP program can then be interfaced from the object-oriented program using a library, which receives the ASP program and input arguments as parameters. It then evaluates the ASP program under the given input and returns its results as objects generated from its answer sets. This is described by

$$O = eval(P, v_1, \dots, v_n),$$

where P is an ASP program, $v_1, \dots, v_n$ are input arguments, and O is a set of objects corresponding one-to-one to the answer sets (which in turn correspond to the solutions to the problem at hand). The object-oriented code can then process these objects in a loop.

Internally, the evaluation of an ASP program from object-oriented code with given input arguments consists of three steps:

1. Facts are generated from the input arguments according to the input specification.
2. These facts along with the original ASP program are passed to the ASP solver.
3. The answer sets are transformed into objects according to the output specification.

The exact transformation performed by *eval* will be described in the rest of this section. However, we first make some assumptions about the object-oriented language at hand. This is in order to allow for instantiating the approach also for arbitrary programming languages which provide the following minimum set of features (while we provide a reference implementation for Python, cf. Section 4).

3.1 The Object-oriented Language

Most importantly, our system assumes the language to be object-oriented. Data is organized in *classes*, which, for the purposes of this paper, are definitions of structures with *named attributes* and *methods*. An *object* is an instance of a class which assigns certain values to its attributes and is accessible via a variable in the object-oriented code. For an object x we let $x.attr$ be the value of the attribute $attr$.

The language must provide at least the classes *str* and *int* with the usual functionality for representing character strings and integers, respectively. Moreover, classes that are to be used during input mapping are required to provide a *toString* method which returns a string representation of the object at hand, that can be used as an ASP constant.

Furthermore, the following collection types are required, i.e., types that allow for storing (ordered or unordered) groups of objects. For these collection types we allow *type parameters* T specified in angle brackets; that is, the type of objects which can be stored in the respective collection is constrained by this parameter.

– *Set*$\langle T\rangle$: a collection of unique objects of type T.
– *Dictionary*$\langle K,V\rangle$: a mapping from objects of type K (the *keys*) to objects of type V (the *values*).
– *Tuple*$\langle T_1,\ldots,T_n\rangle$: an ordered list of fixed length n, where the component at position i is of type T_i for $1\le i\le n$.
– *Sequence*$\langle T\rangle$: a finite ordered sequence containing objects of type T, where elements are addressable by an integer index.

For a collection object x of type *Tuple* or a *Sequence*, let $x[i]$ for $i\in\mathbb{N}$ be its i-th element.

3.2 Input Specification

We now describe our language for specifying the input of an ASP program. The input specification defines the expected arguments and how they are mapped to ASP facts. Before we introduce the language for the general case, we show an intuitive example.

Example 2. Assume we have a graph represented by a set of instances of the class `Node`. The attribute `label` of this class is a unique string identifying the node, and the attribute `neighbors` is a list containing the neighbor nodes. The following input specification takes such a set of nodes as input and maps it to the two predicates `vertex` and `edge`:

```
1 INPUT (Set<Node> nodes) {
2    vertex(n.label) for n in nodes;
3    edge(n.label, m.label) for n in nodes for m in n.neighbors; }
```

More precisely, the input of the ASP program is a set of nodes, given as an instance of class `Set<Node>`, where `Node` is a custom class defined in the object-oriented code. Given this input, line 2 defines the predicate `vertex` by generating a fact `vertex(n.label)` for every object `n` in the set `nodes`.

Similarly, line 3 defines the predicate `edge` from the adjacency lists of the nodes. To this end, the loop-like construct iterates over the `nodes` and for each node over its (attribute) `neighbors`. Multiple iterations are evaluated from left to right, i.e., variables bound in an iteration are available in all iterations to the right, and in the predicate arguments. The outer loop iterates over all nodes, and, for each node, the inner loop over its neighbors. □

Definition of the Language. In general, an *input specification* ι is of the form

$$\texttt{INPUT}\ (t_1\ v_1,\ldots,t_n\ v_n)\ \{s_1;s_2;\ldots s_k;\}$$

where $v_1,\ldots,v_n$ are *input parameters* to the ASP program of types $t_1,\ldots,t_n$, and $s_1,\ldots,s_k$ are predicate specifications defined as follows. Each *predicate specification* s_i for $1\le i\le k$ is of form

$$p(x_1,\ldots,x_m)\ \texttt{for}\ w_1\ \texttt{in}\ y_1\ldots\texttt{for}\ w_\ell\ \texttt{in}\ y_\ell \tag{2}$$

where $p\in\mathscr{P}$ is a predicate symbol, $x_1,\ldots,x_m$ are objects of any type, $w_1,\ldots,w_\ell$ are (iteration) variables, and $y_1,\ldots,y_\ell$ are collections.

The first step in the evaluation of a program P with an input specification ι of the above form under parameters $v_1, \ldots, v_n$, i.e., the evaluation of $eval(P, v_1, \ldots, v_n)$, is the construction of facts $genFacts(\iota, v_1, \ldots, v_n) = \bigcup_{1 \le i \le k} genFacts(s_i, v_1, \ldots, v_n)$ from the given parameters. To this end, each predicate specification s_i of form (2) is handled independently as follows and yields a set of input facts $genFacts(s_i, v_1, \ldots, v_n)$.

The constructs **for** w_i **in** y_i in a predicate specification s for $1 \le i \le \ell$ are *iteration clauses* which are used to let w_i iterate over the contents of collection y_i, similarly to loops in procedural languages. If y_i is a `Set`, then w_i iterates over its elements, if it is a `Dictionary` resp. `Sequence`/`Tuple`, then w_i iterates over its *pairs* of the current key resp. index and value. Multiple iteration clauses are nested from left to right, i.e., the leftmost iteration clause defines the outermost iteration. For a predicate specification of form (2), an iteration variable w_i can be accessed in all y_j with $j > i$ and in $x_1, \ldots, x_n$.

Such a predicate specification s generates all facts $genFacts(s, v_1, \ldots, v_n)$ of the form $p(u_1, \ldots, u_m)$, where each term $u_j \in \mathscr{T}$ for $1 \le j \le m$ is the string representation x_j`.toString()` of the corresponding object x_j. In x_j, all iteration variables $w_1, \ldots, w_\ell$ defined by s and all input parameters $v_1, \ldots, v_n$ can be accessed.

Language Shortcuts. When iterating over a *Sequence* y using **for** w **in** y, the current index and element are accessed by $w[0]$ and $w[1]$, respectively. Iteration over a *Dictionary* y works analogously, where $w[0]$ and $w[1]$ yield the current key and value, respectively. Towards a more readable notation, further allow to use a list of iteration variables $(w_1, \ldots, w_m)$ in place of a single iteration variable w. Then, w_i is automatically assigned the value of $w[i]$ for all $1 \le i \le m$. For instance, an iteration **for** w **in** y over the key-value pairs $(w[0], w[1])$ in the *Dictionary* y may be written as **for** (k, v) **in** y. In case of iteration over nested collection types, this shortcut can be repeated recursively, i.e., an element in a list of iteration variables can itself be a list. Additionally, it is possible to use the anonymous variable _. Each occurrence of _ is viewed as a new variable that is never referenced.

Example 3. The following example illustrates the iteration over a sequence. The input is a series of measurements of the current *temperature* and *humidity*, respectively, which corresponds to the type `Sequence<Tuple<int, int>>`; the time point serves as index in this sequence.

```
INPUT (Sequence<Tuple<int, int>> readings) {
  temperature(x[0], x[1][0]) for x in readings;
  humidity(t, hum) for (t,(_,hum)) in readings; }
```

The specification of the predicate `temperature` uses a single iteration variable x. Since `readings` is of type *Sequence*, this iteration variable x is assigned pairs of the current index and value, where the value itself is a pair of temperature and humidity. Hence, $x[0]$ refers to the current index and $x[1]$ refers to a pair of measurements, where $x[1][0]$ is the temperature and $x[1][1]$ is the humidity. In contrast, the definition of `humidity` uses a (nested) pair of iteration variables $(t, (\_, hum))$ which are directly assigned the time point t and the humidity hum. Since the temperature value is not used in this definition, it is ignored by using an anonymous variable. □

3.3 Output Specification

The evaluation of an ASP program yields a collection of answer sets. The output specification enables the object-oriented program to extract information from them by assigning values to the attributes of a certain class depending on the atoms in the current answer set. Then, each answer set yields one instance of this class.

Before we introduce the language in the general case we present an intuitive example.

Example 4. Assume we have evaluated an ASP program that computes a graph represented by the predicates *vertex* and *edge* (cf. Example 2), and received the answer set I. Assume that every vertex v has exactly one associated color c represented by $color(v,c)$. Consider the answer set

$$I = \{vertex(a), vertex(b), vertex(c), edge(a,b), edge(a,c), \\ color(a, blue), color(b, red), color(c, red)\}$$

and the following output specification:

```
OUTPUT {
   labels = set { query: vertex(X); content: X; };
   red_nodes = set { query: color(X, red); content: X; }; }
```

It defines the values of the attributes `labels` and `red_nodes` of the output class, depending on the current answer set I. The value of the attribute `labels` is a new instance of class *Set*, whose elements x are extracted from atoms $vertex(x) \in I$. To this end, the **`query`** specifies a set of literals which are matched against the atoms in the answer set I. For every match, the argument terms of the matched atoms are assigned to the corresponding variables in the query, and an element to be added to the *Set* instance is constructed according to the **`content`** property. In this example, for the given answer set I, the value of `labels` will thus correspond to the set $\{a,b,c\}$. Similarly, the set `red_nodes` contains the labels of all red-colored nodes, i.e., the values $\{b,c\}$. □

Definition of the Language. We now explain output specifications in the general case. The basic building blocks are *(attribute) expressions* which transform atoms, sets of atoms, and/or the results of subexpressions to attribute values (see below). The value $mapOutput(e,I)$ of an expression e is itself an object, that is constructed relative to a fixed answer set I. Based on expressions, an *output specification* ω is then of the form

$$\textbf{OUTPUT}\ \{w_1 = e_1; \dots w_k = e_k;\}$$

where $w_1, \dots, w_k$ are pairwise distinct attributes and $e_1, \dots, e_k$ are expressions. For a program P with such an output specification ω, each answer set $I \in AS(P)$ is then mapped to an object $mapOutput(\omega, I)$, which contains the attributes w_i, whose values are given by $mapOutput(e_i, I)$, for all $1 \leq i \leq k$.

- *Basic Expressions* are integer and string constants e which evaluate to themselves, i.e., $mapOutput(e,I) = e$ for all I. We show their usage together with collection expressions.
- *Collection Expressions* are of one of the following forms:

- **`set`** { **`query`**: q; **`content`**: e; }
- **`sequence`** { **`query`**: q; **`index`**: i; **`content`**: e; }
- **`dictionary`** { **`query`**: q; **`key`**: k; **`content`**: e; }

In all cases, the **`query`** $q = l_1, \dots, l_n$ specifies a set of (possibly nonground) literals, where each variable must occur in a positive literal akin to safe rules, which are to be checked against the answer set I at hand in order to find substitutions $S(q) = \{\sigma\colon V(q) \to \mathscr{T} \mid I \models \sigma(q)\}$ for the variables $V(q)$ occurring in q, which satisfy the query under I, akin to query answering. Then, for each such substitution σ, **`content`** e specifies a (sub)expression which defines how to construct an object from the current variable substitution. In the simplest case, this is a variable occurring in q which will, after application of the substitution σ, be a basic expression (i.e., a constant). However, the **`content`** can also be nested collection or composite (see below) expressions, in which case it is recursively evaluated. Moreover, for **`sequence`**, i is a variable or integer constant, and for **`dictionary`**, k is a (sub)expression.

The value $\mathit{mapOutput}(e,I)$ of the expression $e =$ **`set`** { **`query`**: q; **`content`**: e'; } wrt. an answer set I is a *Set* with the elements $\{\mathit{mapOutput}(\sigma(e'),I) \mid \sigma \in S(q)\}$, where $\sigma(e')$ results from e' if all variables X occurring in e' are replaced by $\sigma(X)$. Given the expression $e =$ **`sequence`** { **`query`**: q; **`index`**: Y; **`content`**: e'; }, the value $\mathit{mapOutput}(e,I)$ wrt. an answer set I is an instance of *Sequence* containing all elements $\mathit{mapOutput}(\sigma(e'),I)$ for $\sigma \in S(q)$, ordered by the index $\sigma(Y)$ (which is assumed to be an integer and yields an error otherwise).

Example 5. To illustrate, consider the following output specification:

```
OUTPUT {
   indices = set { query: p(I, X); content: int(I); };
   xs = sequence { query: p(I, X); index: I; content: X; }; }
```

The definition of `indices` gathers the first argument of all atoms of p into a *Set*. Note that all constant symbols are mapped to *str* instances by default. The constructor `int` can be used to convert strings to *int* values. On the other hand, the definition of `xs` constructs an instance of *Sequence*. The positions of the elements in `xs` are determined by the variable given as **`index`**, which must occur in **`query`**. For instance, for the answer set $I = \{p(0,a), p(1,b), p(2,a)\}$, the value of `indices` is the set $\{0,1,2\}$ and the value of `xs` is the sequence (a,b,a). $\square$

Similarly, given $e =$ **`dictionary`** { **`query`**: q; **`key`**: k; **`content`**: v; }, the value $\mathit{mapOutput}(e,I)$ wrt. an answer set I is an instance of *Dictionary*, which maps for all $\sigma \in S(q)$ the key $\mathit{mapOutput}(\sigma(k),I)$ to the value $\mathit{mapOutput}(\sigma(v),I)$. Note that the result of applying a substitution σ to the **`key`** k is a general expression itself, which needs to be recursively evaluated. This is opposed to the **`index`** in the previous paragraph, which is, after application of σ, always a basic expression.

Example 6. The following output specification demonstrates how collection expressions can be nested. We assume that every node x is assigned exactly one color c, which is represented by an atom $\mathit{color}(x,c)$. Suppose we want to extract a dictionary which maps each color to the set of nodes with that color. This is done as follows:

```
OUTPUT {
   labels_by_color = dictionary {
      query: color(X, C);
      key: C;
      content: set { query: color(L, C); content: L; }; }; }
```

Evaluating this expression under the answer set I from Example 4, the dictionary `labels_by_color` yields the mappings $blue \mapsto \{a\}$ and $red \mapsto \{b, c\}$. Note that the variable C is introduced in the **dictionary** expression and for every match σ of its **query** `color(X, C)`, the **set** expression is evaluated with C fixed to the matched value $\sigma(C)$, thus generating a set of labels colored by color C. □

– *Composite Expressions* are instances of custom classes of the object-oriented language. They are created by passing appropriate parameters to their constructors, i.e., the expression $cls(e_1, \ldots, e_k)$ with (sub)expressions $e_1, \ldots, e_k$ creates an instance of the class *cls* by calling its constructor with the arguments constructed by the (sub)expressions $e_1, \ldots, e_k$. A special case thereof is the instantiation of *Tuple*, which is written as $(e_1, \ldots, e_k)$.

Example 7. We continue with the answer set I from Example 4.

```
OUTPUT {
   graph = Graph(
      set { query: vertex(X); content: X; },
      set { query: edge(X, Y); content: (X, Y); });
   colored_nodes = set {
      query: color(X, C);
      content: ColoredNode(X, C); }; }
```

The variable `graph` holds an instance of the custom class `Graph`, which is to be defined in the object-oriented language. To create the `Graph` instance, its constructor is called with the set of labels (cf. Example 4) and the set of edges as parameters. The set of edges is defined by a *Tuple* (x, y) for each atom $edge(x, y) \in I$. This example also demonstrates nesting of expressions. The constructor call contains two **set** expressions, and the **content** of the second **set** contains a tuple expression. The value of `colored_nodes` is a *Set* of instances of the class `ColoredNode`, which is defined in the object-oriented language. □

3.4 Overall Evaluation

Given an ASP program P with input specification ι and output specification ω, we can now describe the complete evaluation process under input arguments $v_1, \ldots, v_n$ by

$$eval(P, v_1, \ldots, v_n) = \{mapOutput(\omega, I) \mid I \in AS(P \cup genFacts(\iota, v_1, \ldots, v_n))\}.$$

That is, in the process of evaluating P with input arguments $v_1, \ldots, v_n$ we first generate the set of facts F from the input arguments according to the input specification of P. Then, the answer sets of $P \cup F$ are computed. Finally, each answer set is mapped back to an object as per the output specification of P, yielding a set of objects that can be processed in the object-oriented code.

4 Implementation in Python

We have implemented the language from the previous section in the PY-ASPIO library[1] (ASP Interface to Object-oriented programs) in the *Python* programming language[2]. The library utilizes *dlvhex*[3] as the underlying answer set solver, but adaptation to other reasoners is simple.

Object Model. In our implementation, an object *x* is accepted to have an attribute *attr* if `getattr(x, "attr")` does not raise an `AttributeError`[4]. Subscripts $x[i]$ can be used on any object *x* that supports subscription, not just *Sequence* and *Tuple* instances. For the output mapping, by default, we substitute the builtin Python classes `int`, `str`, `tuple`, `frozenset`, `list`, and `dict` for the respective abstract types *int*, *str*, *Tuple*, *Set*, *Sequence*, and *Dictionary*. In Python, the contents of sets and the keys of dictionaries are required to be *hashable* objects. Since `list` and `dict` are mutable collections and thus not hashable, they cannot immediately be used as contents of sets. However, it easy to replace these types by immutable, hashable variants either by using a constructor in the output specification, or by setting configuration parameters of the PY-ASPIO library.

Interface of PY-ASPIO. When interfacing an ASP program using the PY-ASPIO library, it is expected to contain the input and output specifications as defined above in special comments starting with `%!` inside the ASP code (while normal comments in ASP begin with just `%`). This ensures that annotations are conservative in the sense that the program uses still valid ASP syntax and can also be used independently of the PY-ASPIO library.

The central interface to the program is then provided by the class `Program`. It represents an ASP program and provides methods to evaluate it under given input and access its answer sets as objects. The actual ASP code can be provided either as file or as string passed to the constructor of the class. The input and output specifications contained in the program are parsed and interpreted at the time a program is accessed for the first time. Then, once a `Program` instance was created, the program can be evaluated multiple times with varying input arguments.

In the following we assume that `p` is an instance of `Program`. The ASP program is evaluated by calling the method `p.solve(...)` with arguments as defined by the input specification. This method returns a Python iterable that contains a `Result` instance for every answer set that has been computed. These `Result` objects possess attributes corresponding to the variables defined in the output specification of `p`.

If custom class constructors are used in the output specification, PY-ASPIO needs to be able to resolve class names. To this end, we distinguish two types of names:

– Qualified names (e.g., `package.module.Class`) are automatically resolved.
– Unqualified names must be registered manually before evaluating the ASP program. The programmer can either register each name separately with method calls of form `p.register(MyClass)`, or import all global names in the current scope with `p.register_dict(globals())`.

[1] Available at https://github.com/hexhex/py-aspio
[2] https://www.python.org
[3] http://www.kr.tuwien.ac.at/research/systems/dlvhex/
[4] Information about Python-specific terms is available at https://docs.python.org/3/

Most settings in PY-ASPIO have global and local counterparts. For example, it is possible to register names locally for the ASP program `p` by calling its instance methods, e.g., `p.register(...)`. On the other hand, simpler applications that need to set up these bindings once for all ASP programs may call the global counterparts on the PY-ASPIO module: `aspio.register(...)`.

Example 8. We show now a complete example of how the PY-ASPIO library is used. The Python script in Listing 1 loads the ASP program shown in Listing 2 and demonstrates three ways of evaluating the program and accessing the output data.

Listing 1: Python program in the file `coloring.py`

```python
1  from collections import namedtuple
2  import aspio
3  # Define classes and create sample data
4  Node = namedtuple('Node', ['label'])
5  ColoredNode = namedtuple('ColoredNode', ['label', 'color'])
6  Arc = namedtuple('Arc', ['start', 'end'])
7  a, b, c = Node('a'), Node('b'), Node('c')
8  nodes = {a, b, c}
9  arcs = {Arc(a, b), Arc(a, c), Arc(b, c)}
10 # Register class names with aspio
11 aspio.register_dict(globals())
12 # Load ASP program and input/output specifications from file
13 prog = aspio.Program(filename='coloring.dl')
14 # Iterate over all answer sets
15 for result in prog.solve(nodes, arcs):
16     print(result.colored_nodes)
17 # Shortcut if only one variable is needed (note prefix "each_")
18 for colored_nodes in prog.solve(nodes, arcs).each_colored_nodes:
19     print(colored_nodes)
20 # Compute a single answer set
21 result = prog.solve_one(nodes, arcs)
22 if result is not None: print(result.colored_nodes)
23 else: print('no answer set')
```

Listing 2: Mapping specification and ASP code in the file `coloring.dl`

```
1 %! INPUT (Set<Node> nodes, Set<Arc> arcs) {
2 %!    node(n.label) for n in nodes;
3 %!    edge(arc.start.label, arc.end.label) for arc in arcs; }
4 %! OUTPUT {
5 %!    colored_nodes = set {
6 %!       query: color(X, C);
7 %!       content: ColoredNode(X, C); }; }
8 color(X, red) v color(X, green) v color(X, blue) :- node(X).
9 :- edge(X, Y), color(X, C), color(Y, C).
```

The input specification defines two input parameters, `nodes` and `arcs` (cf. Listing 2, line 1). The `solve` method must thus be called with two arguments. The output specification declares a single output variable `colored_nodes` (cf. Listing 2, line 5), which is accessed with the same name in the Python code (cf. Listing 1, line 16). □

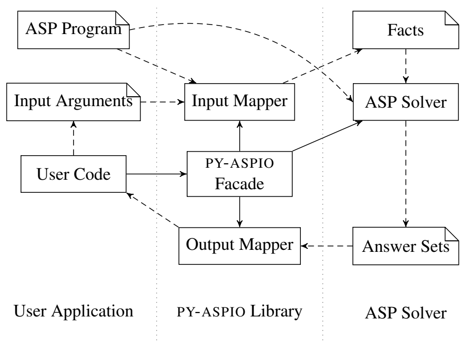

Fig. 1: PY-ASPIO Architecture (data flow $\dashrightarrow$, control flow $\longrightarrow$)

Implementation Architecture. Our system runs the ASP solver as a subprocess to compute answer sets, communicating via pipes and temporary files. Its architecture is shown in Figure 1. Upon invoking the solver, PY-ASPIO immediately returns a wrapper object representing the set of answer sets. Iterating over this wrapper object yields each answer set as soon as it is available, i.e., the client code is not forced to wait until all answer sets have been computed. To reduce communication overhead, PY-ASPIO uses the solver's `--filter` option in order to capture only predicates that are needed to construct output objects, which may allow for optimizations in the solver. The output objects themselves are constructed at the time when they are first accessed by the client code, which is realized by exploiting Python's attribute access mechanism.

The exact mapping of objects to ASP terms depends on their type. Integers are passed to the solver as-is to allow use of arithmetic, and strings are passed as quoted string constants (where enclosed quotation marks have been replaced by an appropriate escape sequence). Any other objects are first converted to strings by calling Python's `str` function and then mapped to quoted string constants.

For more detailed information about the PY-ASPIO library, we refer to the library's documentation and example programs available at https://github.com/hexhex/py-aspio.

5 Applications

In this section we discuss possible applications of the PY-ASPIO library. As noted in Section 1, Answer Set Programming has proven well suited for solving computationally hard problems. Typical examples are planning and scheduling problems under domain-specific constraints. They are cumbersome to implement in classical languages but can be modeled easily in a declarative language.

However, a solution of the underlying computational problem in form of an ASP program is usually not enough to allow practical application. End-users are typically not trained in logic programming and thus cannot be expected to directly edit ASP files, to enter input data in form of facts or to interpret the answer sets. Moreover, even for trained personnel, manual data entry in this form would be inefficient and calls for automated interfaces to other system components which realize, e.g., user interfaces, interfaces to data sources, or other parts which cannot (easily) be solved in ASP. Such an interface is provided by the PY-ASPIO library. Since Python is suitable for developing state-of-the-art graphical user interfaces (GUI, e.g., using pyqt[5]) and Web applications (e.g., using django[6]), one can create a Python program to support data entry and presentation of results. With help of the PY-ASPIO library, it is then easy to integrate an ASP program to solve the actual underlying problem.

We now discuss some concrete possible applications.

Creating Timetables for Schools. The pupils of a typical school are grouped as classes and each class, depending on its grade and possibly specialization, needs to receive instruction in certain subjects for a fixed number of hours per week. Every teacher can teach certain subjects and each class should be assigned one fixed teacher per subject. The challenge is then to find a timetable where each class fulfills the teaching requirements while observing a large number of constraints, e.g., a class cannot be taught two subjects at the same time, certain subjects require special facilities which may be limited; we refer to Faber et al. [3] for a more thorough discussion.

Since the application is to be used by the administrative personnel which is in charge of the creation of timetables it should come with an easy-to-use GUI. The application further needs to connect to a database for retrieval of information about classes, such as their teaching requirements, available teachers and their subjects. The GUI may present this data and allow for specifying constraints and desired properties (e.g., that the music room is not available on Fridays). These components are typically implemented in an object-oriented language.

However, the actual creation of the timetable is more easily realized in ASP. To this end, PY-ASPIO can be used to interface the ASP program. An input specification can be used to pass the input from the GUI to the program, and an output specification is used to extract the candidate timetables into objects which can be displayed in the GUI.

Afterwards, the results can be further processed by the object-oriented components. For instance, timetables may be printed for distribution to classes and teachers. To this end, the application first needs to layout the data in a printable format and then send this layout to the printer by relying on the printing functions of the operating system, which cannot be done in ASP.

Workforce Allocation Problem. As part of their regular operation, companies need to assign workers to tasks under a number of (possibly weak) constraints such as necessary skills and legal restrictions. Creating these allocation plans manually is a time-intensive and error-prone task. Ricca et al. [12] have developed an encoding in ASP, and a GUI in Java on top of the ASP program. Here, again, the object-oriented part of such an application needs to interface with other systems and the user to retrieve input data, and the output of the ASP program (allocation plans, employee statistics and/or a list of constraint violations) is used for additional processing. Besides displaying the plans, the system might generate reports about the workload statistics and forward them to the responsible managers, and automatically notify the workers of their new assignments once approved by the user.

[5] https://www.riverbankcomputing.com/software/pyqt/intro

[6] https://www.djangoproject.com/

6 Discussion and Related Work

The specification language has been designed to be independent from the concrete object-oriented language. While the types on the input parameters are not strictly required by our Python implementation, they allow our approach to be implemented for statically-typed languages such as C++ and Java. An implementation in C++, for example, can be realized with a separate compilation step, turning the annotated ASP program into a function that accepts the input parameters from the input specification and returns a result with a structure as defined in the output specification. The types of the expressions in the output specification can be inferred automatically to allow static declarations of the output variables, e.g. both definitions in Example 4 result in instances of $Set\langle str \rangle$.

Our system uses the DLVHEX solver as backend, which can solve HEX programs in addition to plain ASP programs. HEX is an extension of ASP that allows the incorporation of external computation sources (cf. [2] for details). Currently, external computation sources can be implemented via DLVHEX plug-ins written either in C++ or Python. In the latter case, PY-ASPIO allows for a seamless integration of application components, which cannot be easily realized in ASP, a declarative component, and the possibility to make callbacks to the procedural code.

We now discuss differences to existing works on bridging the gap between ASP and procedural or object-oriented languages. Oetsch et al. [10] present a system that integrates ASP with Java programs. As in our system, the ASP code resides in a separate file and is annotated with input parameters (here, the annotations show similarity to a Java function definition). However, in this system the input arguments are accessed from the rules of the ASP program using dedicated atoms, and special atoms govern the creation of output objects. Another approach is *JASP* [4], which extends Java by constructs to allow ASP code to be directly embedded in Java code, forming a hybrid language. This hybrid language is compiled to pure Java code that uses a lower-level API to interact with the ASP solver. An interesting aspect of this system is that standardized annotations of the Java Persistence API (JPA) for object-relational mapping are used to interface with ASP. In contrast to both JASP and the system presented by Oetsch et al., we do not modify the syntax of ASP itself. Because of this, all annotated programs are still valid ASP programs that can continue to be used in existing systems. Furthermore, most ASP variations can immediately be used without adaptation of the mapping library, e.g. external atoms via existing DLVHEX plugins can already be used with PY-ASPIO.

EmbASP [5] is a recent work that describes the abstract architecture of an ASP framework and implements this framework as a Java library with an emphasis on support for mobile platforms. However, while EmbASP intends to provide a language-agnostic ASP framework, the definitions of the input/output mapping still depend on Java-specific features (alternatives for other languages are discussed briefly). The mapping is guided by custom annotations on Java classes, which are associated one-to-one with a predicate, where annotations on their fields define the argument positions. *asp4j*[7] is a Java library that utilizes custom class annotations to guide the mapping process, similar to EmbASP. Unlike JASP, EmbASP and asp4j, the mapping annotations in our system are in the ASP code and separate from the object-oriented code. This means the ASP program forms a

[7] https://github.com/hbeck/asp4j

self-contained component, with the input and output specifications defining an external, object-oriented interface to the declarative ASP part, while still being independent from the concrete object-oriented language. This approach enables the ASP component to be maintained separately, and allows a single ASP file to be used simultaneously by multiple applications (possibly in different programming languages, provided the custom classes have the same names). While the ASP programmer still needs certain knowledge about the object-oriented world, the object-oriented programmer can call the ASP component as-is, without requiring much accomodation on the OOP side, e.g., usually no separate data-holding objects need to be constructed by the client code when using our approach.

Tweety [13] is a collection of Java libraries with the goal of providing a general framework for many different approaches to knowledge representation and reasoning. Its ASP component includes a parser for ASP programs, classes to construct ASP programs in Java, and connections to several solvers. *PyASP*[8] provides a Python wrapper of the answer set solving tools gringo and clasp from the Potassco suite [6]. While both Tweety and PyASP provide an easy way to invoke an ASP solver from object-oriented code, the burden of mapping between objects and facts/answer sets is still mostly on the user because input data is passed by manually instantiating fact objects, and answer sets are returned as lists of literals which, again, must be inspected manually.

7 Conclusion and Outlook

We have introduced a language to provide an object-oriented interface for ASP programs. It allows for the specification of input and output data of ASP programs in terms of objects of a conventional object-oriented language. The language is flexible and can be implemented for arbitrary object-oriented languages which provide a minimum set of features. The approach does not depend on a specific ASP dialect or solver, which allows many ASP extensions to be used together with input/output specifications. However, we also provide a reference implementation PY-ASPIO in Python that enables programmers to easily evaluate ASP programs from Python code with the help of this language. This implementation currently supports DLVHEX as underlying ASP solver, thus offering access to the full power of HEX programs in addition to regular ASP programs.

For future work, one possible starting point concerns the language itself. More features, such as the possibility to handle errors (e.g. duplicate indices when creating sequences) may be added to increase flexibility. Also, because of the independence of the specification language from a concrete object-oriented language, implementations for other languages may be provided. In particular, for statically typed, compiled languages such as C++ and Java. While this paper focuses on interfacing with object-oriented languages, the same approach can conceivably be extended to other languages that provide appropriate data structures (e.g., in Haskell, record types might be used instead of classes). Moreover, the ASP solver is currently executed in a separate process, which incurs overhead from inter-process communication. It is worthwhile to investigate the impact of this overhead and, in case this is significant, integrate the ASP solver as a shared library in the host process to enable more efficient communication.

[8] https://pypi.python.org/pypi/pyasp

References

1. Eiter, T., Ianni, G., Krennwallner, T.: Answer Set Programming: A Primer. In: 5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009. LNCS, vol. 5689, pp. 40–110. Springer (2009), http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf
2. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In: IJCAI. pp. 90–96. Professional Book Center (2005)
3. Faber, W., Leone, N., Pfeifer, G.: Representing school timetabling in a disjunctive logic programming language. In: Proceedings of the 13th Workshop on Logic Programming (WLP98). vol. 194 (1998)
4. Febbraro, O., Leone, N., Grasso, G., Ricca, F.: JASP: A Framework for Integrating Answer Set Programming with Java. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012. AAAI Press (2012), http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4520
5. Fuscà, D., Germano, S., Zangari, J., Anastasio, M., Calimeri, F., Perri, S.: A framework for easing the development of applications embedding answer set programming. In: Cheney, J., Vidal, G. (eds.) Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016. pp. 38–49. ACM (2016), http://doi.acm.org/10.1145/2967973.2968594
6. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection. AI Commun. 24(2), 107–124 (2011), http://www.mpi-inf.mpg.de/departments/rg1/conferences/deduction10/papers/martin-gebser.pdf
7. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9(3–4), 365–386 (1991)
8. Grasso, G., Leone, N., Manna, M., Ricca, F.: ASP at Work: Spin-off and Applications of the DLV System. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 6565, pp. 432–451. Springer (2011), http://dx.doi.org/10.1007/978-3-642-20832-4_27
9. Ielpa, S.M., Iiritano, S., Leone, N., Ricca, F.: An ASP-Based System for e-Tourism. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5753, pp. 368–381. Springer (2009), http://dx.doi.org/10.1007/978-3-642-04238-6_31
10. Oetsch, J., Pührer, J., Tompits, H.: Extending Object-Oriented Languages by Declarative Specifications of Complex Objects using Answer-Set Programming. CoRR abs/1112.0922 (2011), http://arxiv.org/abs/1112.0922
11. Redl, C.: The DLVHEX system for knowledge representation: Recent advances (system description). Theory and Practice of Logic Programming
12. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the Gioia-Tauro seaport. TPLP 12(3), 361–381 (2012), http://dx.doi.org/10.1017/S147106841100007X
13. Thimm, M.: Tweety: A Comprehensive Collection of Java Libraries for Logical Aspects of Artificial Intelligence and Knowledge Representation. In: Baral, C., Giacomo, G.D., Eiter, T. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014. AAAI Press (2014), http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/7811