Integrating Answer Set Programming with Object-oriented Languages

Jakob Rath and Christoph Redl

jakob.rath@student.tuwien.ac.at, redl@kr.tuwien.ac.at

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Knowledge-Based Systems Group

January 16, 2017

Outline

Motivation

Answer Set Programming

- Answer Set Programming (ASP) is a declarative programming paradigm [**?**].

Motivation

Answer Set Programming

- Answer Set Programming (ASP) is a declarative programming paradigm [**?**].
- Applications: *workforce management* [**?**], *generating holiday plans for tourists* [**?**], cf. [**?**].

Motivation

Answer Set Programming

- Answer Set Programming (ASP) is a declarative programming paradigm [**?**].
- Applications: *workforce management* [**?**], *generating holiday plans for tourists* [**?**], cf. [**?**].

Limitations

- Typical end-user applications contain components which cannot be (easily) solved in ASP:
  - graphical user interfaces
  - presentation of results
  - interfaces to data sources
  - etc.

Motivation

Answer Set Programming

- Answer Set Programming (ASP) is a declarative programming paradigm [**?**].
- Applications: *workforce management* [**?**], *generating holiday plans for tourists* [**?**], cf. [**?**].

Limitations

- Typical end-user applications contain components which cannot be (easily) solved in ASP:
  - graphical user interfaces
  - presentation of results
  - interfaces to data sources
  - etc.
- Realizing such components is in the domain of traditional object-oriented (OOP) languages.

Motivation

Typical approach

- Use ASP programs as components of a larger application.
- The ASP program solves the core computational problem, while other components are implemented in an object-oriented language.

Motivation

Typical approach

- Use ASP programs as components of a larger application.
- The ASP program solves the core computational problem, while other components are implemented in an object-oriented language.
- To this end, object-oriented code
  1. adds input as facts,
  2. evaluates the ASP program, and
  3. interprets the answer sets.

Motivation

Typical approach

- Use ASP programs as components of a larger application.
- The ASP program solves the core computational problem, while other components are implemented in an object-oriented language.
- To this end, object-oriented code
  1. adds input as facts,
  2. evaluates the ASP program, and
  3. interprets the answer sets.
- But: An implementation from scratch is similar for most applications $\Rightarrow$ repetitive work.

Motivation

Contribution

- The ASP program is extended with annotations which specify input/output.
- Input specifications define how objects are mapped to facts.
- Output specifications define how answer sets are mapped back to objects.
- Based on annotations, the integration with object-oriented code is automated.

Motivation

Contribution

- The ASP program is extended with annotations which specify input/output.
- Input specifications define how objects are mapped to facts.
- Output specifications define how answer sets are mapped back to objects.
- Based on annotations, the integration with object-oriented code is automated.
- In contrast to existing approaches,
  ours is independent of a concrete OOP language.
- We provide a prototypical implementation PY-ASPIO for Python.

Outline

Evaluating ASP Programs from Object-Oriented Code

Overview

- We want to use ASP programs similarly to subprocedures:
  an ASP program P performs a computation over input parameters $v_1, \ldots, v_n$,
  each answer set should correspond to a solution object.

Evaluating ASP Programs from Object-Oriented Code

Overview

- We want to use ASP programs similarly to subprocedures:
  an ASP program P performs a computation over input parameters $v_1, \ldots, v_n$,
  each answer set should correspond to a solution object.
  $\Rightarrow$ $eval(P, v_1, \ldots, v_n)$ should return a set of objects (=problem solutions).

Evaluating ASP Programs from Object-Oriented Code

Overview

- We want to use ASP programs similarly to subprocedures:
  an ASP program P performs a computation over input parameters $v_1, \ldots, v_n$,
  each answer set should correspond to a solution object.
  $\Rightarrow$ $eval(P, v_1, \ldots, v_n)$ should return a set of objects (=problem solutions).

- Approach: the ASP program is annotated with input/output specifications.
  Annotations are added as special comments of form `%!` to the ASP code.

Evaluating ASP Programs from Object-Oriented Code

Overview

- We want to use ASP programs similarly to subprocedures:
  an ASP program P performs a computation over input parameters $v_1, \ldots, v_n$,
  each answer set should correspond to a solution object.
  $\Rightarrow$ $eval(P, v_1, \ldots, v_n)$ should return a set of objects (=problem solutions).
- Approach: the ASP program is annotated with input/output specifications.
  Annotations are added as special comments of form `%!` to the ASP code.
- We then provide an interpreter library for evaluating ("calling")
  such an annotated program:
  - it takes an annotated ASP program and a list of input parameters (objects) as input, and
  - returns a set of objects (corresponding the results of the ASP program).

Evaluating ASP Programs from Object-Oriented Code

Evaluation

More precisely, the interpreter library performs the following tasks:

1. Parameters $v_1, \ldots, v_n$ are converted to facts according to input specification ι.
2. These facts along with the ASP program P are passed to the ASP solver.
3. The answer sets are mapped to objects O according to output specification ω.

Evaluating ASP Programs from Object-Oriented Code

Evaluation

More precisely, the interpreter library performs the following tasks:

1. Parameters $v_1, \ldots, v_n$ are converted to facts according to input specification ι.
2. These facts along with the ASP program P are passed to the ASP solver.
3. The answer sets are mapped to objects O according to output specification ω.

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

Evaluating ASP Programs from Object-Oriented Code

Evaluation

More precisely, the interpreter library performs the following tasks:

1. Parameters $v_1, \ldots, v_n$ are converted to facts according to input specification ι.
2. These facts along with the ASP program P are passed to the ASP solver.
3. The answer sets are mapped to objects O according to output specification ω.

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

Language independence

- The specification language is largely independent of a concrete OOP language
  $\Rightarrow$ porting the interpreter library to other OOP languages is easily possible.
  $\Rightarrow$ the same annotated program can be used with multiple OOP languages.

Evaluating ASP Programs from Object-Oriented Code

Evaluation

More precisely, the interpreter library performs the following tasks:

1. Parameters $v_1, \ldots, v_n$ are converted to facts according to input specification ι.
2. These facts along with the ASP program P are passed to the ASP solver.
3. The answer sets are mapped to objects O according to output specification ω.

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

Language independence

- The specification language is largely independent of a concrete OOP language
  $\Rightarrow$ porting the interpreter library to other OOP languages is easily possible.
  $\Rightarrow$ the same annotated program can be used with multiple OOP languages.
- Currently, we provide a prototypical implementation PY-ASPIO for Python.

Assumptions about the Object-Oriented Language

Requirements

The specification language is “largely” independent of the OOP language, we obviously have to presuppose a minimum set of features:

Assumptions about the Object-Oriented Language

Requirements

The specification language is "largely" independent of the OOP language, we obviously have to presuppose a minimum set of features:

1. Data is organized in classes, which consist of named attributes and methods.

Assumptions about the Object-Oriented Language

Requirements

The specification language is "largely" independent of the OOP language, we obviously have to presuppose a minimum set of features:

1. Data is organized in classes, which consist of named attributes and methods.
2. The language must provide the classes *str* and *int*.

Assumptions about the Object-Oriented Language

Requirements

The specification language is "largely" independent of the OOP language, we obviously have to presuppose a minimum set of features:

1. Data is organized in classes, which consist of named attributes and methods.
2. The language must provide the classes *str* and *int*.
3. We presuppose the following collection types:
   - $Set\langle T\rangle$:
     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 $1 \leq i \leq n$.
   - $Sequence\langle T\rangle$:
     a finite ordered sequence containing objects of type T, where elements are addressable by an integer index.

Outline

Input Specification

Example (3-Colorability)

Assume we want to use a 3-colorability program from our object-oriented code.

Let the graph in the object-oriented code be represented by sets of nodes and edges, where

- nodes are instances of class Node with the attribute label as a unique string identifiying the node, and
- edges are instances of class Edge, where the attributes first and second are the nodes at both ends of the edge.

Input Specification

Example (3-Colorability)

Assume we want to use a 3-colorability program from our object-oriented code.

Let the graph in the object-oriented code be represented by sets of nodes and edges, where

- nodes are instances of class Node with the attribute label as a unique string identifiying the node, and
- edges are instances of class Edge, where the attributes first and second are the nodes at both ends of the edge.

Mapping of the input graph to predicates vertex and edge and problem encoding:

```
%! INPUT (Set<Node> nodes, Set<Edge> edges) {
%!     vertex(n.label) for n in nodes;
%!     edge(e.first.label, e.second.label) for e in edges; }
color(X,r) v color(X,g) v color(X,b) :- vertex(X)
:- color(X,C), color(Y,C), edge(X,Y)
```

Input Specification Language

Language Definition

- 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 of types $t_1, \ldots, t_n$,
  $s_1, \ldots, s_k$ are predicate specifications defined as follows.

- Each predicate specification s_i for $1 \leq i \leq k$ is of form
  $$p(x_1, \ldots, x_m)\ \texttt{for}\ w_1\ \texttt{in}\ y_1 \ldots \texttt{for}\ w_\ell\ \texttt{in}\ y_\ell \quad (1)$$
  where
  p ... predicate symbol,
  $x_1, \ldots, x_m$ are objects of any type,
  $w_1, \ldots, w_\ell$ are (iteration) variables,
  $y_1, \ldots, y_\ell$ are collections.

  Here, the constructs $\texttt{for}\ w_i\ \texttt{in}\ y_i$ are iteration clauses which are used to let w_i iterate over the contents of a collection object y_i.

Output Specification

Example (3-Colorability (cont'd))

We want to map the valid 3-colorings back to objects.
Let ColoredNode be an extension of Node with the attribute color.

Output Specification

Example (3-Colorability (cont'd))

We want to map the valid 3-colorings back to objects.
Let ColoredNode be an extension of Node with the attribute color.

```
%! INPUT (Set<Node> nodes, Set<Edge> edges) {
%!    vertex(n.label) for n in nodes;
%!    edge(e.first.label, e.second.label) for e in edges; }
%! OUTPUT {
%!    colorednodes = set { query: color(X,C);
%!                         content: ColoredNode(X,C); }; }
color(X,r) v color(X,g) v color(X,b) :- vertex(X)
:- color(X,C), color(Y,C), edge(X,Y)
```

The output is a set containing instances of ColoredNode, which are created by calling the constructor of the class with arguments X and C for each atom color(X,C) in the current answer set.

Output Specification

Example (3-Colorability (cont'd))

```
%! INPUT (Set<Node> nodes, Set<Edge> edges) {
%!     vertex(n.label) for n in nodes;
%!     edge(e.first.label, e.second.label) for e in edges; }
%! OUTPUT {
%!     colorednodes = set { query: color(X,C);
%!                          content: ColoredNode(X,C); }; }
color(X,r) v color(X,g) v color(X,b) :- vertex(X)
:- color(X,C), color(Y,C), edge(X,Y)
```

This program can be called with two parameters of types Set<Node> resp. Set<Edge> and its output is a set of type Set<ColoredNode>.

Output Specification Language

Language Definition

- Building blocks are (possibly nested) expressions which transform atoms, sets of atoms, and/or results of subexpressions to objects.

  Support types:

  - Basic Expressions are integer and string constants e.
  - 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; }
  - Composite Expressions are instances of custom classes of the object-oriented language.

- 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.

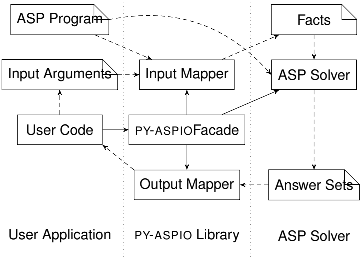
Outline

Implementation

PY-ASPIO

- We implemented an interpreter for our specification language for Python.
- ASP Interface to Object-oriented programs for Python.
- Available at `https://github.com/hexhex/py-aspio`.
- dlvhex is used as ASP solver (switching to other solvers is easily possible).

Implementations for other object-oriented languages are left for future work.



Example: Using the Library from Python

Suppose the ASP program from above is stored in file `coloring.dl`.

Example (3-Colorability (cont'd))

```
from collections import namedtuple
import aspio

# Define classes and create sample data
Node = namedtuple('Node', ['label'])
ColoredNode = namedtuple('ColoredNode', ['label', 'color'])
Edge = namedtuple('Edge', ['first', 'second'])
a, b, c = Node('a'), Node('b'), Node('c')
nodes = {a, b, c}
edges = {Edge(a, b), Edge(a, c), Edge(b, c)}

# Register class names with aspio
aspio.register_dict(globals())

# Load ASP program and input/output specifications from file
prog = aspio.Program(filename='coloring.dl')

# Iterate over all answer sets
for result in prog.solve(nodes, edges):
    print(result.colored_nodes)
```

Applications

Potential industrial applications

- Hard problems occur in the real world: packing, scheduling, optimization, etc.

Applications

Potential industial applications

- Hard problems occur in the real world: packing, scheduling, optimization, etc.
- Users of such applications are not necessarily IT experts:
  - department heads who create working plans,
  - teachers who create timetables,
  - logisticians who create packing plans, etc.

Applications

Potential industrial applications

- Hard problems occur in the real world: packing, scheduling, optimization, etc.
- Users of such applications are not necessarily IT experts:
  - department heads who create working plans,
  - teachers who create timetables,
  - logisticians who create packing plans, etc.

$\Rightarrow$ The direct usage of ASP might be unacceptable.
$\Rightarrow$ A user interface is needed!

Applications

Potential industrial applications

- Hard problems occur in the real world: packing, scheduling, optimization, etc.
- Users of such applications are not necessarily IT experts:
  - department heads who create working plans,
  - teachers who create timetables,
  - logisticians who create packing plans, etc.

  $\Rightarrow$ The direct usage of ASP might be unacceptable.
  $\Rightarrow$ A user interface is needed!
- Interfaces to databases, Web services, etc. might be needed.

Applications

Potential industrial applications

- Hard problems occur in the real world: packing, scheduling, optimization, etc.
- Users of such applications are not necessarily IT experts:
  - department heads who create working plans,
  - teachers who create timetables,
  - logisticians who create packing plans, etc.

$\Rightarrow$ The direct usage of ASP might be unacceptable.
$\Rightarrow$ A user interface is needed!

- Interfaces to databases, Web services, etc. might be needed.

Upcoming research application

- ASP extensions support epistemic negation (quantification over answer sets).
- The evaluation algorithm for such programs is based on evaluating sets of programs and reasoning about their answer sets.
- An implementation of epistemic ASP based on PY-ASPIO is work in progress.

Outline

Conclusion

Integrating ASP with object-oriented code

- ASP is good for solving hard problems.
- Applications often have components which cannot be easily realized in ASP.
- $\Rightarrow$ We integrate ASP with object-oriented languages.

Conclusion

Integrating ASP with object-oriented code

- ASP is good for solving hard problems.
- Applications often have components which cannot be easily realized in ASP.
- $\Rightarrow$ We integrate ASP with object-oriented languages.

Existing approaches

- Focus on a particular language, cf. e.g. [**?**], [**?**], [**?**].
- Tweety [**?**] and PyASP (`https://pypi.python.org/pypi/pyasp`) provide only a generic atom-based interface, but no customizable mapping.

Conclusion

Integrating ASP with object-oriented code

- ASP is good for solving hard problems.
- Applications often have components which cannot be easily realized in ASP.
- $\Rightarrow$ We integrate ASP with object-oriented languages.

Existing approaches

- Focus on a particular language, cf. e.g. [**?**], [**?**], [**?**].
- Tweety [**?**] and PyASP (`https://pypi.python.org/pypi/pyasp`) provide only a generic atom-based interface, but no customizable mapping.

Future work

- Language extensions.
- Implementation for other object-oriented languages.

References I

Erdem, E., Gelfond, M., and Leone, N. (2016).
Applications of answer set programming.
AI Magazine, 37(3):53–68.

Febbraro, O., Leone, N., Grasso, G., and Ricca, F. (2012).
JASP: A Framework for Integrating Answer Set Programming with Java.
In Brewka, G., Eiter, T., and McIlraith, S. A., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press.

Fuscà, D., Germano, S., Zangari, J., Anastasio, M., Calimeri, F., and Perri, S. (2016).
A framework for easing the development of applications embedding answer set programming.
In Cheney, J. and Vidal, G., editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 38–49. ACM.

Gelfond, M. and Lifschitz, V. (1991).
Classical Negation in Logic Programs and Disjunctive Databases.
New Generation Computing, 9(3–4):365–386.

References II

Ielpa, S. M., Iiritano, S., Leone, N., and Ricca, F. (2009).
An ASP-Based System for e-Tourism.
In Erdem, E., Lin, F., and Schaub, T., editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 368–381. Springer.

Oetsch, J., Pührer, J., and Tompits, H. (2011).
Extending Object-Oriented Languages by Declarative Specifications of Complex Objects using Answer-Set Programming.
CoRR, abs/1112.0922.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., and Leone, N. (2012).
Team-building with answer set programming in the Gioia-Tauro seaport.
TPLP, 12(3):361–381.

Thimm, M. (2014).
Tweety: A Comprehensive Collection of Java Libraries for Logical Aspects of Artificial Intelligence and Knowledge Representation.
In Baral, C., Giacomo, G. D., and Eiter, T., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press.