EmbASP Documentation

Release 7.1.0

DeMaCS-Unical

Dec 06, 2020

DOCUMENTATION

A framework for the integration (embedding) of **Logic Programming** in external systems for generic applications. It helps developers at designing and implementing complex reasoning tasks by means of solvers on different platforms.

The framework can be implemented in a object-oriented programming language of choice, easing and guiding the generation of suitable libraries for the use of specific solvers on selected platforms. We currently provide 3 implementations (in Narrative , in Theoretic and in Programmatic) and ready-made libraries for the embedding several logic programming languages (mainly on the Desktop platform). In particular, we provide support for:

- ASP (Answer Set Programming) solvers
  - DLV (also for the Android platform, for the Narrative language)
  - DLV2
  - clingo
  - DLVHEX
- PDDL (Planning Domain Definition Language)
  - Cloud solver Solver.Planning.Domains (also for the Android platform, for the Narrative language).
- Datalog
  - I-DLV

However, the framework has been designed to be easily extensible and adaptable to different solvers and platforms. It is worth to notice that solvers are invoked in different modes; for instance, SPD is invoked via a remote connection, while for the other, binaries are effectively embedded and natively executed.

CHAPTER

ONE

DOCUMENTATION

1.1 Narrative implementation

The following figure provides some details about classes and interfaces of the implementation.

1.1.1 Base module implementation

Each component in the *Base* module has been implemented by means of an abstract class, generic class or interface that will specialize in the following packages.

In particular, the `Handler` class collects `InputProgram` and `OptionDescriptor` objects communicated by the user.

For what the asynchronous mode is concerned, the interface `Service` depends from the interface `CallBack`, since once the reasoning service has terminated, the result of the computation is returned back via a class `CallBack`.

1.1.2 Platforms module implementation

In order to support a new platform, the `Handler` and `Service` components must be adapted.

As for the Android platform, we developed an `AndroidHandler` that handles the execution of an `AndroidService`, which provides facilities to manage the execution of a solver on the Android platform.

Similarly, for the desktop platform we developed a `DesktopHandler` and a `DesktopService`, which generalizes the usage of a solver on the desktop platform, allowing both synchronous and asynchronous execution modes.

1.1.3 Languages module implementation

This module includes specific classes for the management of input and output to ASP, Datalog and PDDL solvers.

The `Mapper` component of the *Languages* module is implemented via a `Mapper` class, that allows to translate input and output into Narrative objects. Such translations are guided by ANTLR4 library and Narrative Annotations , a form of metadata that mark Narrative code and provide information that is not part of the program itself: they have no direct effect on the operation of the code they annotate.

In our setting, we make use of such feature so that it is possible to translate facts into strings and vice-versa via two custom annotations, defined according to the following syntax:

- *@Id (string_name)* : the target must be a class, and defines the predicate name (in the ASP/Datalog case) and the action name (in the PDDL case) the class is mapped to;
- *@Param (integer_position)* : the target must be a field of a class annotated via *@Id*, and defines the term (and its position) in the atom (in the ASP/Datalog case) and in the action (in the PDDL case) the field is mapped to.

By means of the Narrative Reflection mechanisms, annotations are examined at runtime, and taken into account to properly define the translation.

If the classes intended for the translation are not annotated or not correctly annotated, an exception is raised.

In addition to the `Mapper`, this module features three sub-modules which are more strictly related to ASP, PDDL and Datalog.

1.1.4 Specialization module Implementation

The classes `DLVAnswerSets`, `DLV2AnswerSets`, `ClingoAnswerSets`, `DLVHEXAnswerSets` implement specific extensions of the `AnswerSets` class, the `SPDPlan` class extends `Plan`, while `IDLVMinimalModels` extends `MinimalModels` . These classes are in charge of manipulating the output of the respective solvers (e.g. IDLV).

Moreover, this module can contain classes extending `OptionDescriptor` to implement specific options of the solver at hand.

1.1.5 Class Diagram

A complete UML Class Diagram is available here.

For further information, contact embasp@mat.unical.it or visit our website.

1.2 Theoretic implementation

The following figure provides some details about classes and interfaces of the implementation.

1.2.1 Base module implementation

Each component in the *Base* module has been implemented by means of generic class or interface that will specialize in the following packages.

In particular, the `Handler` class collects `InputProgram` and `OptionDescriptor` objects communicated by the user.

For what the asynchronous mode is concerned, the class `Service` depends from the interface `CallBack`, since once the reasoning service has terminated, the result of the computation is returned back via a class `CallBack`.

1.2.2 Platforms module implementation

In order to support a new platform, the `Handler` and `Service` components must be adapted.

For the desktop platform we developed a `DesktopHandler` and a `DesktopService`, which generalizes the usage of a solver on the desktop platform, allowing both synchronous and asynchronous execution modes.

1.2.3 Languages module implementation

This module includes specific classes for the management of input and output to ASP, Datalog and PDDL solvers.

The `Mapper` component of the *Languages* module is implemented via a `Mapper` class, that allows to translate input and output into Theoretic objects. Such translations are guided by ANTLR4 library and `Predicate` abstract class, also present in the module.

To make possible translate facts into strings and vice versa, the classes that want to represent a predicate, must extend the abstract class `Predicate`, and must be implemented by including the following code:

- *predicateName="string_name"* : must be entered as a class field and must contain the predicate name (in the ASP/Datalog case) or the action name (in the PDDL case) to map;
- *[("class_field_name_1", int), ("class_field_name_2"), ...]* : Is a list that must be passed to super in the constructor, and must contain so many tuples how many are the class field, containing the field name, sorted by the position of the terms they represent, and optionally the keyword int if the field represents an integer.

Thanks to the structure of the `Predicate` class, this information is passed to the `Mapper` class, to correctly perform the translation mechanism.

If the classes intended for the translation are not constructed correctly in this way, an exception is raised.

In addition to the `Mapper`, this module features three sub-modules which are more strictly related to ASP, Datalog and PDDL.

1.2.4 Specialization module implementation

The classes `DLVAnswerSets`, `DLV2AnswerSets`, `ClingoAnswerSets`, `DLVHEXAnswerSets` implement specific extensions of the `AnswerSets` class, the `SPDPlan` class extends `Plan`, while `IDLVMinimalModels` extends `MinimalModels` . These classes are in charge of manipulating the output of the respective solvers (e.g. IDLV).

Moreover, this module can contain classes extending `OptionDescriptor` to implement specific options of the solver at hand.

1.2.5 Class Diagram

A complete UML Class Diagram is available here.

For further information, contact embasp@mat.unical.it or visit our website.

1.3 Programmatic implementation

The following figure provides some details about classes and interfaces of the implementation.

1.3.1 Base module implementation

Each component in the *Base* module has been implemented by means of abstract class, generic class or interface that will specialize in the following packages.

In particular, the `Handler` class collects `InputProgram` and `OptionDescriptor` objects communicated by the user.

For what the asynchronous mode is concerned, the class `Service` depends from the interface `CallBack`, since once the reasoning service has terminated, the result of the computation is returned back via a class `CallBack`.

1.3.2 Platforms module implementation

In order to support a new platform, the `Handler` and `Service` components must be adapted.

For the desktop platform we developed a `DesktopHandler` and a `DesktopService`, which generalizes the usage of a solver on the desktop platform, allowing both synchronous and asynchronous execution modes.

1.3.3 Languages module implementation

This module includes specific classes for the management of input and output to ASP, Datalog and PDDL solvers.

The `Mapper` component of the *Languages* module is implemented via a `Mapper` class, that allows to translate input and output into Programmatic objects. Such translations are guided by ANTLR4 library and Programmatic Attributes, a form of metadata that mark Programmatic code and provide information that is not part of the program itself: they have no direct effect on the operation of the code they annotate.

In our setting, we make use of such features so that it is possible to translate facts into strings and vice-versa via two custom attributes, defined according to the following syntax:

- *[Id(string_name)]* : the target must be a class, and defines the predicate name (in the ASP/Datalog case) and the action name (in the PDDL case) the class is mapped to;
- *[Param(integer_position)]* : the target must be a field of a class annotated via *[Id(string_name)]*, and defines the term (and its position) in the atom (in the ASP/Datalog case) and in the action (in the PDDL case) the field is mapped to.

By means of the Programmatic Reflection mechanism, attributes are examined at runtime, and taken into account to properly define the translation.

If the classes intended for the translation are not annotated or not correctly annotated, an exception is raised.

In addition to the `Mapper`, this module features three sub-modules which are more strictly related to ASP, Datalog and PDDL.

1.3.4 Specialization module implementation

The classes `DLVAnswerSets`, `DLV2AnswerSets`, `ClingoAnswerSets`, `DLVHEXAnswerSets` implement specific extensions of the `AnswerSets` class, the `SPDPlan` class extends `Plan`, while `IDLVMinimalModels` extends `MinimalModels` . These classes are in charge of manipulating the output of the respective solvers (e.g. IDLV).

Moreover, this module can contain classes extending `OptionDescriptor` to implement specific options of the solver at hand.

1.3.5 Class Diagram

A complete UML Class Diagram is available here.

For further information, contact embasp@mat.unical.it or visit our website.

1.4 Technical documentation

1.4.1 Doxygen

- Narrative Doxygen documentation
- Theoretic Doxygen documentation
- Programmatic Doxygen documentation

For further information, contact embasp@mat.unical.it or visit our website.

1.5 Implementations

- *Narrative implementation*
- *Theoretic implementation*
- *Programmatic implementation*

1.6 Technical documentation

- *Technical documentation*

CHAPTER

TWO

EXAMPLES

2.1 Shortest-path ASP Narrative

2.1.1 Getting started

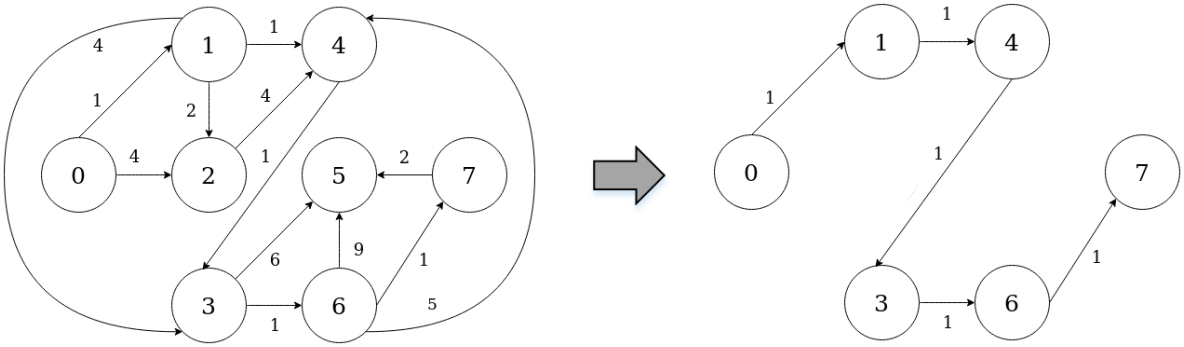
The framework is released as JAR file to be used on a Desktop platform, therefore it can be easily imported and used in any Narrative project.

The framework needs ANTLR4 library for its operation. You can download the JAR and include directly in your project or you can use Gradle or Maven.

2.1.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to solve the shortest-path problem.

The complete code of this example is freely available here.



We will make use of the annotation-guided mapping, in order to create Narrative object constituting ASP predicates.

To this purpose, the following classes are intended to represent possible predicates that an ASP program can use:

```
@Id("edge")
public class Edge {

  @Param(0)
```

(continues on next page)

(continued from previous page)

```
  private int from;

  @Param(1)
  private int to;

  @Param(2)
  private int weight;

  public Edge(int from, int to, int weight) {
    this.from = from;
    this.to = to;
    this.weight = weight;
  }

  [...]
}
```

```
@Id("path")
public class Path {

  @Param(0)
  private int from;

  @Param(1)
  private int to;

  @Param(2)
  private int weight;

  public Path(int from, int to, int weight) {
    this.from = from;
    this.to = to;
    this.weight = weight;
  }

  [...]
}
```

At this point, supposing that we have embedded the DLV2 solver in this project, we can start deploying our application:

```
public class ShortestPath {

  private static int from, to;       // source and destination node
  private static ArrayList<Integer> sortedPath;   // edges in the shortest path␣
↪(sorted)

  public static void main(String[] args) {

    try {

      Handler handler = new DesktopHandler(new DLV2DesktopService("executable/dlv2"));

      ASPMapper.getInstance().registerClass(Edge.class);
      ASPMapper.getInstance().registerClass(Path.class);

      InputProgram input = new ASPInputProgram();
```

(continues on next page)

(continued from previous page)

```
      from = 0;
      to = 7;

      String rules = "from(" + from + "). to(" + to + ")."
         + "path(X,Y,W) | notPath(X,Y,W) :- from(X), edge(X,Y,W)."
         + "path(X,Y,W) | notPath(X,Y,W) :- path(_,X,_), edge(X,Y,W), not to(X)."
         + "visited(X) :- path(_,X,_)."
         + ":- to(X), not visited(X)."
         + ":~ path(X,Y,W). [W@1 ,X,Y]";

      input.addProgram(rules);

      for(Edge edge : getEdges())
        input.addObjectInput(edge);

      handler.addProgram(input);

      AnswerSets answerSets = (AnswerSets) handler.startSync();

      for(AnswerSet answerSet : answerSets.getOptimalAnswerSets()) {

        ArrayList<Path> path = new ArrayList<Path>();  // edges in the shortest path␣
↪(unsorted)
        int sum = 0;        // total weight of the path

        for(Object obj : answerSet.getAtoms()) {
          if(obj instanceof Path) {
            path.add((Path)obj);
            sum += ((Path)obj).getWeight();
          }
        }

        join(from,path,sortedPath);   // sorts the edges
        print(sortedPath,sum);        // shows the path
      }

    } catch (Exception e) {
      e.printStackTrace();
    }

  }

  private static ArrayList<Edge> getEdges() {
    ArrayList<Edge> edges = new ArrayList<Edge>();

    edges.add(new Edge(0,1,1));
    edges.add(new Edge(0,2,4));
    edges.add(new Edge(1,2,2));
    edges.add(new Edge(1,3,4));
    edges.add(new Edge(1,4,1));
    edges.add(new Edge(2,4,4));
    edges.add(new Edge(3,5,6));
    edges.add(new Edge(3,6,1));
    edges.add(new Edge(4,3,1));
    edges.add(new Edge(6,4,5));
    edges.add(new Edge(6,5,9));
```

(continues on next page)

(continued from previous page)

```
    edges.add(new Edge(6,7,1));
    edges.add(new Edge(7,5,2));

    return edges;
  }

  [...]

}
```

The class contains an `Handler` instance as field, that is initialized with a `DesktopHandler` using the parameter `DLV2DesktopService` with a string representing the path to the DLV2 local solver.

The `ASPMapper` registers the classes created before in order to manage the input and output objects.

A string and a list of `Edge` representing facts, rules and constraints of the ASP program are added to an `ASPInputProgram`, and the `ASPInputProgram` is added to the `Handler`.

Finally the solver is invoked, and the output is retrieved.

The output predicates can be managed accordingly to the user's desiderata. In this example the `Path` predicates, that represent the shortest path, are collected, sorted, and printed as well as the total weight of the path.

For further information, contact embasp@mat.unical.it or visit our website.

2.2 Shortest-path ASP Theoretic

2.2.1 Getting started

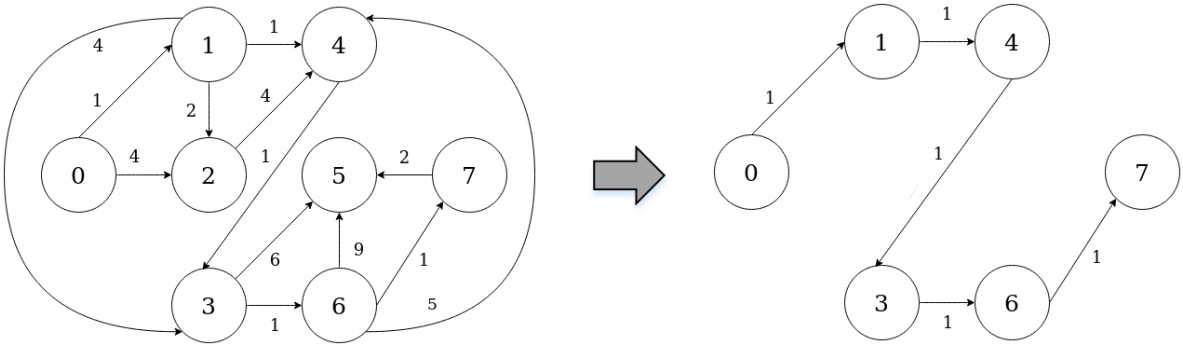
The framework is released as EGG file to be used on a Desktop platform, therefore it can be easily installed in a Theoretic installation.

The framework needs ANTLR4 library for its operation.

2.2.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to solve the shortest-path problem.

The complete code of this example is freely available here.



We will make use of the annotation-guided mapping, in order to create Theoretic object constituting ASP predicates.

To this purpose, the following classes are intended to represent possible predicates that an ASP program can use:

```
class Edge(Predicate):
  predicate_name = "edge"

  def __init__(self, source=None, destination=None, weight=None):
    Predicate.__init__(self, [("source"),("destination"),("weight")])
    self.source = source
    self.destination = destination
    self.weight = weight

  [...]
```

```
class Path(Predicate):
  predicate_name = "path"

  def __init__(self, source=None, destination=None, weight=None):
    Predicate.__init__(self, [("source"),("destination"),("weight")])
    self.source = source
    self.destination = destination
    self.weight = weight

  [...]
```

At this point, supposing that we have embedded the DLV2 solver in this project, we can start deploying our application:

```
def getEdges():

  edges = []

  edges.append(Edge(0,1,1))
  edges.append(Edge(0,2,4))
  edges.append(Edge(1,2,2))
  edges.append(Edge(1,3,4))
  edges.append(Edge(1,4,1))
  edges.append(Edge(2,4,4))
  edges.append(Edge(3,5,6))
  edges.append(Edge(3,6,1))
  edges.append(Edge(4,3,1))
```

(continues on next page)

(continued from previous page)

```
  edges.append(Edge(6,4,5))
  edges.append(Edge(6,5,9))
  edges.append(Edge(6,7,1))
  edges.append(Edge(7,5,2))

  return edges

try:

  handler = DesktopHandler(DLV2DesktopService("../../executable/dlv2"))

  ASPMapper.get_instance().register_class(Edge)
  ASPMapper.get_instance().register_class(Path)

  inputProgram = ASPInputProgram()

  source = 0    # source node
  destination = 7    # destination node

  rules = "source(" + str(self.source) + "). destination(" + str(self.destination) +
↪")."
  rules += "path(X,Y,W) | notPath(X,Y,W) :- source(X), edge(X,Y,W)."
  rules += "path(X,Y,W) | notPath(X,Y,W) :- path(_,X,_), edge(X,Y,W), not to(X)."
  rules += "visited(X) :- path(_,X,_)."
  rules += ":- destination(X), not visited(X)."
  rules += ":~ path(X,Y,W). [W@1 ,X,Y]"

  inputProgram.add_program(rules)
  inputProgram.add_objects_input(self.getEdges())

  handler.add_program(inputProgram)

  answerSets = handler.start_sync()

  for answerSet in answerSets.get_optimal_answer_sets():
      path = []     # edges in the shortest path (unsorted)
      sum_ = 0      # total weight of the path

      for obj in answerSet.get_atoms():
          if isinstance(obj, Path):
              path.append(obj)
              sum_ += int(obj.get_weight())

      sortedPath = []   # edges in the shortest path (sorted)
      join(source, path, sortedPath)   # sorts the edges
      show(sortedPath, sum_)     # shows the path

except Exception as e:
  print(str(e))
```

The class contains an `Handler` instance as field, that is initialized with a `DesktopHandler` using the parameter `DLV2DesktopService` with a string representing the path to the DLV2 local solver.

The `ASPMapper` registers the classes created before in order to manage the input and output objects.

A string and a list of `Edge` representing facts, rules and constraints of the ASP program are added to an `ASPInputProgram`, and the `ASPInputProgram` is added to the `Handler`.

Finally the solver is invoked, and the output is retrieved.

The output predicates can be managed accordingly to the user's desiderata. In this example the `Path` predicates, that represent the shortest path, are collected, sorted, and printed as well as the total weight of the path.

For further information, contact embasp@mat.unical.it or visit our website.

2.3 Shortest-path ASP Programmatic

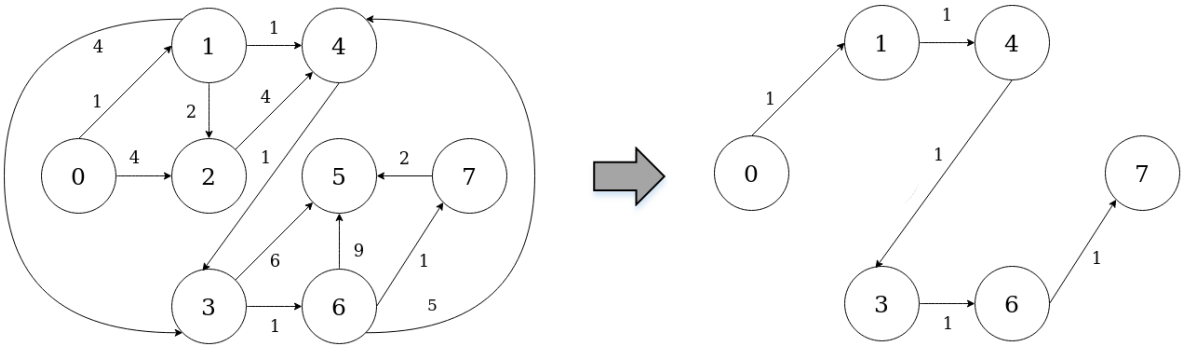
2.3.1 Getting started

The framework is released as DLL file to be used on a Desktop platform, therefore it can be easily added and used in any Programmatic project.

2.3.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to solve the shortest-path problem.

The complete code of this example is freely available here.



We will make use of the annotation-guided mapping, in order to create Programmatic object constituting ASP predicates.

To this purpose, the following classes are intended to represent possible predicates that an ASP program can use:

```
[Id("edge")]
class Edge
{
  [Param(0)]
  private int from;

  [Param(1)]
  private int to;
```

(continues on next page)

(continued from previous page)

```
  [Param(2)]
  private int weight;

  public Edge(int from, int to, int weight)
  {
    this.from = from;
    this.to = to;
    this.weight = weight;
  }
```

```
[Id("path")]
class Path
{
  [Param(0)]
  private int from;

  [Param(1)]
  private int to;

  [Param(2)]
  private int weight;

  public Path(int from, int to, int weight)
  {
    this.from = from;
    this.to = to;
    this.weight = weight;
  }
```

At this point, supposing that we have embedded the DLV2 solver in this project, we can start deploying our application:

```
class ShortestPath
{
  private static int from, to;       // source and destination node
  private static List<int> sortedPath;    // edges in the shorted path (sorted)

  public static void Main(string[] args)
  {
    try
    {
      Handler handler = new DesktopHandler(new DLV2DesktopService("../../../
↪executable/dlv2.win"));

      ASPMapper.Instance.RegisterClass(typeof(Edge));
      ASPMapper.Instance.RegisterClass(typeof(Path));

      InputProgram input = new ASPInputProgram();

      from = 0;
      to = 7;

      String rules = "from(" + from + ").to(" + to + ")." +
         "path(X,Y,W) | notPath(X,Y,W) :- from(X), edge(X,Y,W)." +
         "path(X,Y,W) | notPath(X,Y,W) :- path(_,X,_), edge(X,Y,W), not to(X)." +
         "visited(X) :- path(_,X,_)." +
```

(continues on next page)

(continued from previous page)

```
        ":- to(X), not visited(X)." +
        ":~ path(X,Y,W). [W@1 ,X,Y]";

    input.AddProgram(rules);

    foreach (Edge edge in getEdges())
    {
       input.AddObjectInput(edge);
    }

    handler.AddProgram(input);

    AnswerSets answerSets = (AnswerSets)handler.StartSync();

    foreach (AnswerSet answerSet in answerSets.GetOptimalAnswerSets())
    {
      List<Path> path = new List<Path>();    //  edges in the shortest path␣
↪(unsorted)
      int sum = 0;                           //  total weight of the path

      foreach (object obj in answerSet.Atoms)
      {
        if (typeof(Path).IsInstanceOfType(obj))
        {
          path.Add((Path)obj);
          sum += ((Path)obj).getWeight();
        }
      }

      join(from, path, sortedPath);     // sorts the edges
      print(sortedPath, sum);           // show the result
    }

  }
  catch (Exception e)
  {
    Console.WriteLine(e.Source);
  }
}

private static List<Edge> getEdges()
{
  List<Edge> edges = new List<Edge>();

  edges.Add(new Edge(0, 1, 1));
  edges.Add(new Edge(0, 2, 4));
  edges.Add(new Edge(1, 2, 2));
  edges.Add(new Edge(1, 3, 4));
  edges.Add(new Edge(1, 4, 1));
  edges.Add(new Edge(2, 4, 4));
  edges.Add(new Edge(3, 5, 6));
  edges.Add(new Edge(3, 6, 1));
  edges.Add(new Edge(4, 3, 1));
  edges.Add(new Edge(6, 4, 5));
  edges.Add(new Edge(6, 5, 9));
  edges.Add(new Edge(6, 7, 1));
  edges.Add(new Edge(7, 5, 2));
```

(continues on next page)

(continued from previous page)

```
    return edges;
  }

  [...]

}
```

The class contains an `Handler` instance as field, that is initialized with a `DesktopHandler` using the parameter `DLV2DesktopService` with a string representing the path to the DLV2 local solver.

The `ASPMapper` registers the classes created before in order to manage the input and output objects.

A string and a list of `Edge` representing facts, rules and constraints of the ASP program are added to an `ASPInputProgram`, and the `ASPInputProgram` is added to the `Handler`.

Finally the solver is invoked, and the output is retrieved.

The output predicates can be managed accordingly to the user's desiderata. In this example the `Path` predicates, that represent the shortest path, are collected, sorted, and printed as well as the total weight of the path.

For further information, contact embasp@mat.unical.it or visit our website.

2.4 Blocks-world PDDL Narrative

2.4.1 Getting started

The framework is released as JAR file to be used on a Desktop platform, therefore it can be easily imported and used in any Narrative project.

The framework needs ANTLR4 library for its operation. You can download the JAR and include directly in your project or you can use Gradle or Maven.

2.4.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to solve the blocks-world problem.

The complete code of this example is freely available here.

We will make use of the annotation-guided mapping, in order to retrieve the actions constituting a PDDL plan via Narrative objects.

To this purpose, the following classes are intended to represent possible actions that a blocks-world solution plan can feature:

```
@Id("pick-up")
public class PickUp {

  @Param(0)
  private String block;

  [...]
}
```

```
@Id("put-down")
public class PutDown {

  @Param(0)
  private String block;

  [...]
}
```

```
@Id("stack")
public class Stack {

  @Param(0)
  private String block1;

  @Param(1)
  private String block2;

  [...]
}
```

```
@Id("unstack")
public class Unstack {

  @Param(0)
  private String block1;

  @Param(1)
  private String block2;

  [...]
}
```

At this point, supposing that we are given two files defining the blocks-world domain and a problem instance, we can start deploying our application:

```
public class Blocksworld {

  private static String domainFileName = "domain.pddl";
  private static String problemFileName = "p01.pddl";;

  public static void main(String[] args) {
    Handler handler = new DesktopHandler(new SPDDesktopService());

    final InputProgram inputProgramDomain = new PDDLInputProgram(PDDLProgramType.
↪DOMAIN);
    inputProgramDomain.addFilesPath(domainFileName);
```

(continues on next page)

(continued from previous page)

```
    final InputProgram inputProgramProblem = new PDDLInputProgram(PDDLProgramType.
↪PROBLEM);
    inputProgramProblem.addFilesPath(problemFileName);

    handler.addProgram(inputProgramDomain);
    handler.addProgram(inputProgramProblem);

    try {

      PDDLMapper.getInstance().registerClass(PickUp.class);
      PDDLMapper.getInstance().registerClass(PutDown.class);
      PDDLMapper.getInstance().registerClass(Stack.class);
      PDDLMapper.getInstance().registerClass(Unstack.class);

      Plan plan = (Plan)(handler.startSync());

      for (final Object obj : plan.getActionsObjects())
        if (obj instanceof PickUp || obj instanceof Stack || obj instanceof Unstack␣
↪|| obj instanceof PutDown)
          System.out.println(obj.toString());

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The class contains an `Handler` instance as field, that is initialized with a `DesktopHandler` using the required parameter `SPDDesktopService`.

Then it's set-up the input to the solver; since PDDL requires separate definitions for domain and problem, two `PDDLInputProgram` are created and then given to the handler.

The next lines inform the `PDDLMapper` about what classes are intended to map the output actions.

Finally the solver is invoked, and the output is retrieved.

The output actions can be managed accordingly to the user's desiderata.

For further information, contact embasp@mat.unical.it or visit our website.

2.5 Blocks-world PDDL Theoretic

2.5.1 Getting started

The framework is released as EGG file to be used on a Desktop platform, therefore it can be easily installed in a Theoretic installation.

The framework needs ANTLR4 library for its operation.

2.5.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to solve the blocks-world problem.

The complete code of this example is freely available here.

We will make use of the annotation-guided mapping, in order to retrieve the actions constituting a PDDL plan via Theoretic objects.

To this purpose, the following classes are intended to represent possible actions that a blocks-world solution plan can feature:

```
class PickUp(Predicate):
   predicateName="pick-up"

   def __init__(self, block=None):
      super(PickUp, self).__init__([("block")])
      self.block = block

   [...]
```

```
class PutDown (Predicate):
   predicateName="put-down"

   def __init__(self, block=None):
      super(PutDown, self).__init__([("block")])
      self.block = block

   [...]
```

```
class Stack (Predicate):
   predicateName="stack"

   def __init__(self, block1=None, block2=None):
      super(Stack, self).__init__([("block1"), ("block2")])
      self.block1 = block1
      self.block2 = block2

   [...]
```

```
class Unstack (Predicate):
   predicateName="unstack"

   def __init__(self, block1=None, block2=None):
      super(Unstack, self).__init__([("block1"), ("block2")])
      self.block1 = block1
      self.block2 = block2

   [...]
```

At this point, supposing that we are given two files defining the blocks-world domain and a problem instance, we can start deploying our application:

```
handler  = DesktopHandler(SPDDesktopService())
```

(continues on next page)

(continued from previous page)

```
input_domain = PDDLInputProgram(PDDLProgramType.DOMAIN)
input_domain.add_files_path("../domain.pddl")

input_problem= PDDLInputProgram(PDDLProgramType.PROBLEM)
input_problem.add_files_path("../p01.pddl")

handler.add_program(input_domain)
handler.add_program(input_problem)

PDDLMapper.get_instance().register_class(PickUp)
PDDLMapper.get_instance().register_class(PutDown)
PDDLMapper.get_instance().register_class(Stack)
PDDLMapper.get_instance().register_class(Unstack)

output = handler.start_sync()

for obj in output.get_actions_objects():
   if isinstance(obj, PickUp) | isinstance(obj, PutDown) | isinstance(obj, Stack) |␣
↪isinstance(obj, Unstack) :
      print(obj)
```

The file contains an `Handler` instance as field, that is initialized with a `DesktopHandler` using the required parameter `SPDDesktopService`.

Then it's set-up the input to the solver; since PDDL requires separate definitions for domain and problem, two `PDDLInputProgram` are created and then given to the handler.

The next lines inform the `PDDLMapper` about what classes are intended to map the output actions.

Finally the solver is invoked, and the output is retrieved.

The output actions can be managed accordingly to the user's desiderata.

For further information, contact embasp@mat.unical.it or visit our website.

2.6 Blocks-world PDDL Programmatic

2.6.1 Getting started

The framework is released as DLL file to be used on a Desktop platform, therefore it can be easily added and used in any Programmatic project.

2.6.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to solve the blocks-world problem.

The complete code of this example is freely available here.

We will make use of the annotation-guided mapping, in order to retrieve the actions constituting a PDDL plan via Programmatic objects.

To this purpose, the following classes are intended to represent possible actions that a blocks-world solution plan can feature:

```
[Id("pick-up")]
  class PickUp
  {
      [Param(0)]
      private string block;

      [...]
  }
```

```
[Id("put-down")]
  class PutDown
  {
      [Param(0)]
      private string block;

      [...]
  }
```

```
[Id("stack")]
  class Stack
  {
      [Param(0)]
      private string block1;

      [Param(1)]
      private string block2;

      [...]
  }
```

```
[Id("unstack")]
  class Unstack
  {
      [Param(0)]
      private string block1;

      [Param(1)]
      private string block2;

      [...]
  }
```

At this point, supposing that we are given two files defining the blocks-world domain and a problem instance, we can start deploying our application:

```
class Program
{
    static void Main(string[] args)
    {
        Handler handler = new DesktopHandler(new SPDDesktopService());

        InputProgram inputDomain = new PDDLInputProgram(PDDLProgramType.DOMAIN);
        inputDomain.AddFilesPath("domain.pddl");

        InputProgram inputProblem = new PDDLInputProgram(PDDLProgramType.PROBLEM);
        inputProblem.AddFilesPath("p01.pddl");

        handler.AddProgram(inputDomain);
        handler.AddProgram(inputProblem);

        try
        {
            PDDLMapper.Instance.RegisterClass(typeof(PickUp));
            PDDLMapper.Instance.RegisterClass(typeof(PutDown));
            PDDLMapper.Instance.RegisterClass(typeof(Stack));
            PDDLMapper.Instance.RegisterClass(typeof(Unstack));

            Plan plan = (Plan)handler.StartSync();

            foreach(object obj in plan.ActionsObjects)
            {
                if (typeof(PickUp).IsInstanceOfType(obj) || typeof(PutDown).
↪IsInstanceOfType(obj) ||
                    typeof(Stack).IsInstanceOfType(obj) || typeof(Unstack).
↪IsInstanceOfType(obj))
                {
                   Console.WriteLine(obj.ToString());
                }
            }

        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

The class contains an `Handler` instance as field, that is initialized with a `DesktopHandler` using the required parameter `SPDDesktopService`.

Then it's set-up the input to the solver; since PDDL requires separate definitions for domain and problem, two `PDDLInputProgram` are created and then given to the handler.

The next lines inform the `PDDLMapper` about what classes are intended to map the output actions.

Finally the solver is invoked, and the output is retrieved.

The output actions can be managed accordingly to the user's desiderata.

For further information, contact embasp@mat.unical.it or visit our website.

2.7 Transitive Closure Datalog Narrative

2.7.1 Getting started

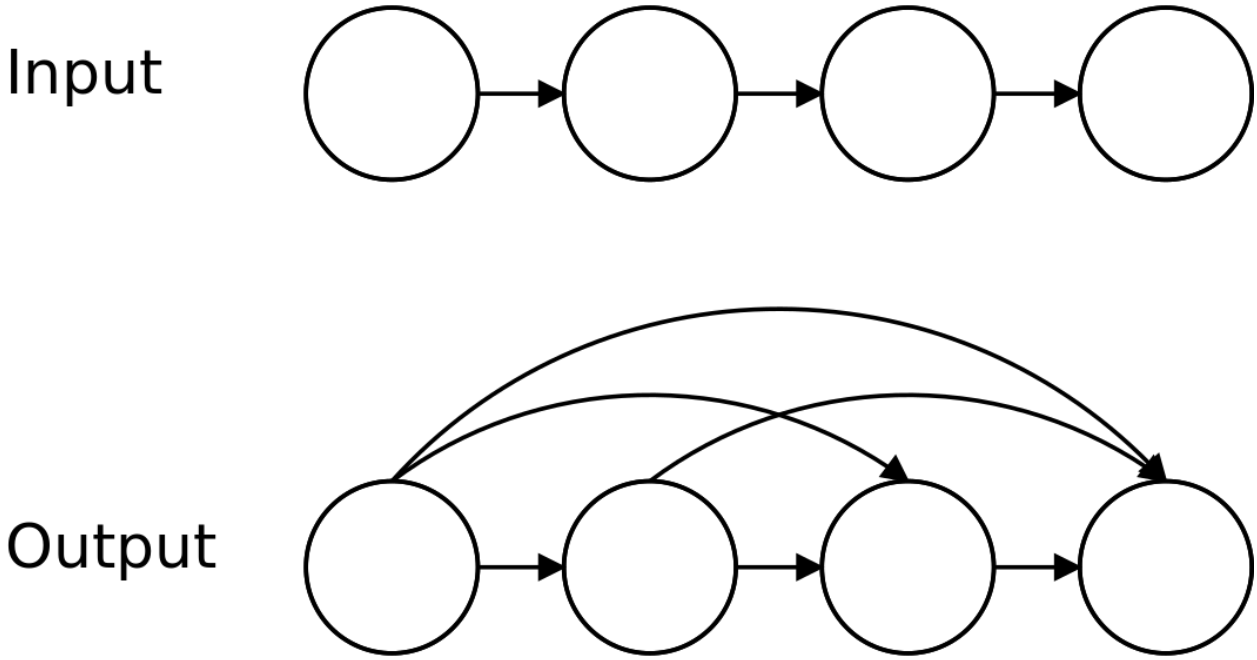
The framework is released as JAR file to be used on a Desktop platform, therefore it can be easily imported and used in any Narrative project.

The framework needs ANTLR4 library for its operation. You can download the JAR and include directly in your project or you can use Gradle or Maven.

2.7.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to compute the transitive closure of a graph.

The complete code of this example is freely available here.



We will make use of the annotation-guided mapping, in order to create Narrative object constituting Datalog predicates.

To this purpose, the following classes are intended to represent possible predicates that a Datalog program can use:

```
@Id("edge")
public class Edge {

  @Param(0)
  private int from;

  @Param(1)
  private int to;
```

(continues on next page)

(continued from previous page)

```
  public Edge(int from, int to) {
    this.from = from;
    this.to = to;
  }

  [...]
}
```

```
@Id("path")
public class Path {

  @Param(0)
  private int from;

  @Param(1)
  private int to;

  public Path(int from, int to) {
    this.from = from;
    this.to = to;
  }

  [...]
}
```

At this point, supposing that we have embedded the IDLV Datalog engine in this project, we can start deploying our application:

```
public class TransitiveClosure {

  public static void main(String[] args) {

    try {
      InputProgram input = new DatalogInputProgram();
      Handler handler = new DesktopHandler(new IDLVDesktopService("executables/idlv
↪"));
      DatalogMapper.getInstance().registerClass(Path.class);
      input.addProgram("path(X,Y) :- edge(X,Y).");
      input.addProgram("path(X,Y) :- path(X,Z), path(Z,Y). ");
      handler.addProgram(input);

      input.addObjectInput(new Edge(1,2));
      input.addObjectInput(new Edge(2,3));
      input.addObjectInput(new Edge(2,4));
      input.addObjectInput(new Edge(3,5));
      input.addObjectInput(new Edge(5,6));

      MinimalModels minimalModels = (MinimalModels) handler.startSync();

      for (MinimalModel m : minimalModels.getMinimalModels()) {
          for (Object a : m.getAtomsAsObjectSet()) {
              if (a instanceof Path) {
                  System.out.println(a);
              }
          }
      }
```

(continues on next page)

(continued from previous page)

```
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

  [...]

}
```

The main method contains an `Handler` instance, that is initialized with a `DesktopHandler` using the parameter `IDLVDesktopService` with a string representing the path to the IDLV local grounder.

The `DatalogMapper` registers the classes created before in order to manage the input and output objects.

A string and a list of `Edge` objects representing facts, rules and constraints of the Datalog program are added to an `DatalogInputProgram`, and the `DatalogInputProgram` is added to the `Handler`.

Finally the solver is invoked, and the output is retrieved.

In this example the `Path` predicates, represent all the arcs in the transitive closure of the starting graph. The output predicates can be managed accordingly to the user's desiderata, as they are simply objects.

For further information, contact embasp@mat.unical.it or visit our website.

2.8 Transitive Closure Datalog Theoretic

2.8.1 Getting started

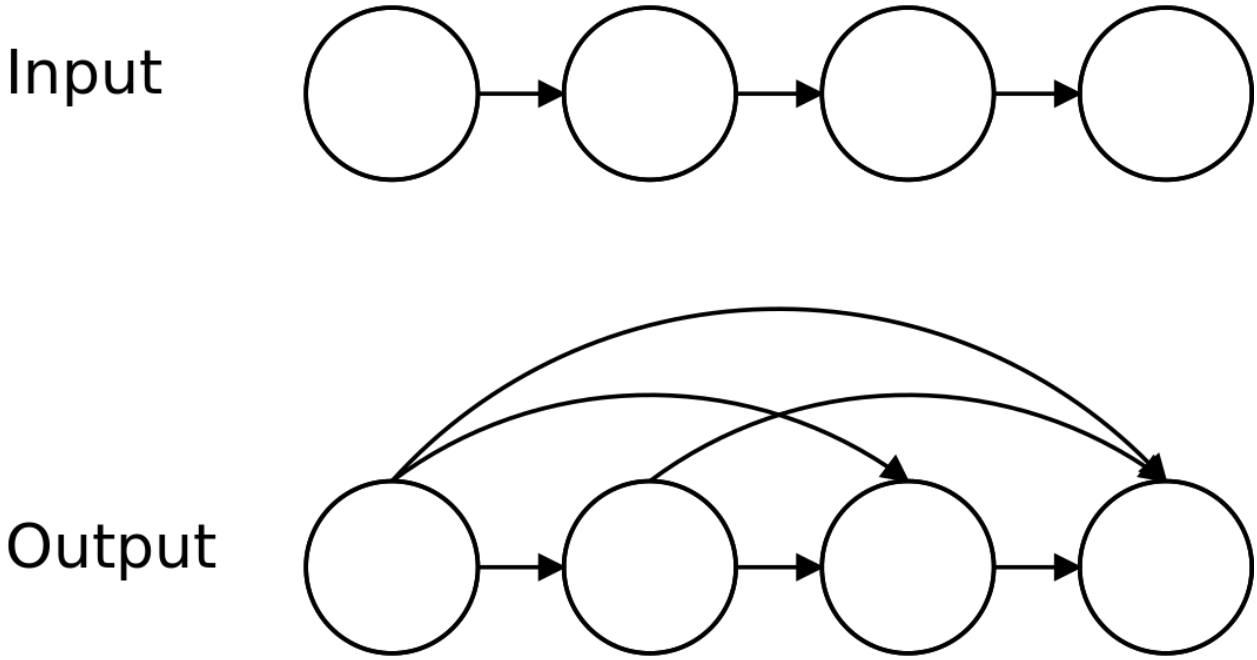
The framework is released as WHL file to be used on a Desktop platform, therefore it can be easily installed via pip.

The framework needs ANTLR4 library for its operation.

2.8.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to compute the transitive closure of a graph.

The complete code of this example is freely available here.



We will make use of the annotation-guided mapping, in order to create Theoretic object constituting Datalog predicates.

To this purpose, the following classes are intended to represent possible predicates that a Datalog program can use:

```
class Edge(Predicate):
    predicate_name = "edge"

    def __init__(self, source=None, destination=None):
        Predicate.__init__(self, [("source"),("destination")])
        self.source = source
        self.destination = destination

    [...]
```

```
class Path(Predicate):
    predicate_name = "path"

    def __init__(self, source=None, destination=None, weight=None):
        Predicate.__init__(self, [("source"),("destination")])
        self.source = source
        self.destination = destination

    [...]
```

At this point, supposing that we have embedded the IDLV Datalog engine in this project, we can start deploying our application:

```
def test_find_reachable_nodes(self):
  try:
      input = DatalogInputProgram()
      handler = DesktopHandler(IDLVDesktopService("executable/idlv"))
      DatalogMapper.get_instance().register_class(Path)
      input.add_program("path(X,Y) :- edge(X,Y).")
      input.add_program("path(X,Y) :- path(X,Z), path(Z,Y). ")
      handler.add_program(input)

      input.add_object_input(Edge(1,2))
      input.add_object_input(Edge(2,3))
      input.add_object_input(Edge(2,4))
      input.add_object_input(Edge(3,5))
      input.add_object_input(Edge(5,6))

      minimalModels = handler.start_sync()

      for o in minimalModels.get_minimal_models().pop().get_atoms_as_objectset():
          if isinstance(o, Path):
              print(o.__str__())

  except Exception as e:
      print(str(e))
```

The main method contains an `Handler` instance, that is initialized with a `DesktopHandler` using the parameter `IDLVDesktopService` with a string representing the path to the IDLV local grounder.

The `DatalogMapper` registers the classes created before in order to manage the input and output objects.

A string and a list of `Edge` objects representing facts, rules and constraints of the Datalog program are added to an `DatalogInputProgram`, and the `DatalogInputProgram` is added to the `Handler`.

Finally the solver is invoked, and the output is retrieved.

In this example the `Path` predicates, represent all the arcs in the transitive closure of the starting graph. The output predicates can be managed accordingly to the user's desiderata, as they are simply objects.

For further information, contact embasp@mat.unical.it or visit our website.

2.9 Transitive Closure Datalog Programmatic

2.9.1 Getting started

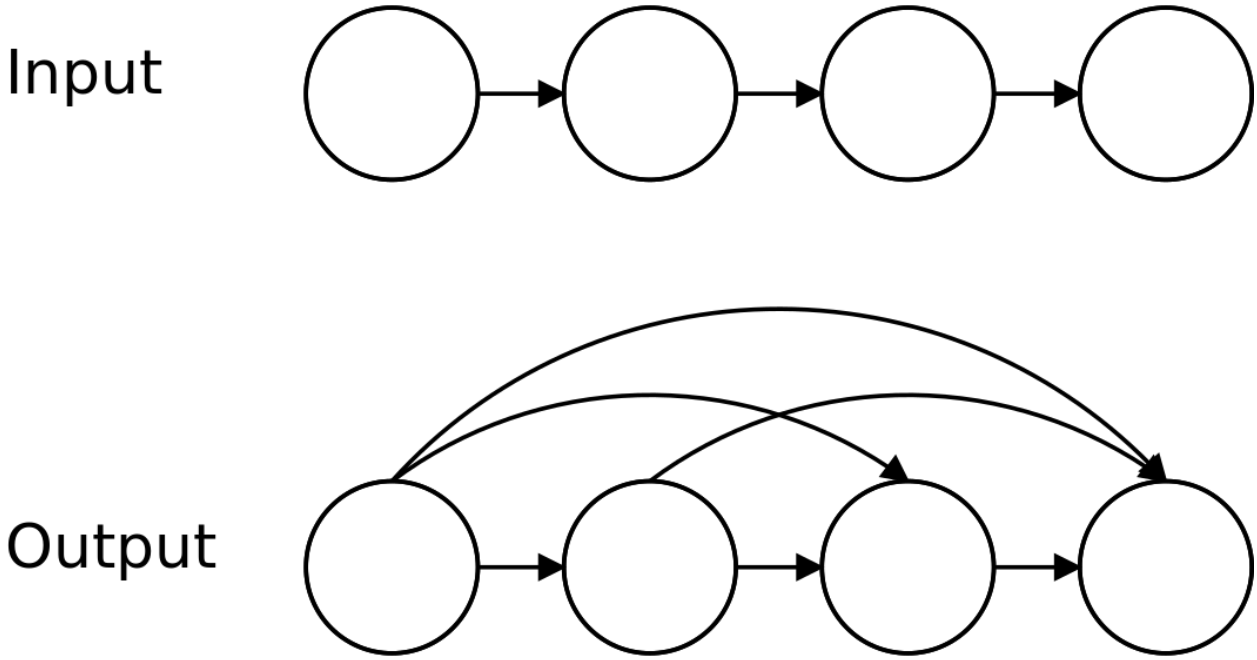
The framework is released as DLL file to be used on a Desktop platform, therefore it can be easily imported and used in any Programmatic project.

The framework needs ANTLR4 library for its operation.

2.9.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Desktop application to compute the transitive closure of a graph.

The complete code of this example is freely available here.



We will make use of the annotation-guided mapping, in order to create Programmatic object constituting Datalog predicates.

To this purpose, the following classes are intended to represent possible predicates that a Datalog program can use:

```
[Id("path")]
class Path
{
    [Param(0)]
    private int from;

    [Param(1)]
    private int to;

    public Path()
    {
        this.from = 0;
        this.to = 0;
    }

    public Path(int from, int to)
    {
        this.from = from;
        this.to = to;
    }

    [...]
}
```

```
[Id("edge")]
class Edge
{
    [Param(0)]
    private int from;

    [Param(1)]
    private int to;

    public Edge()
    {
        this.from = 0;
        this.to = 0;
    }

    public Edge(int from, int to)
    {
        this.from = from;
        this.to = to;
    }

    [...]
}
```

At this point, supposing that we have embedded the IDLV Datalog engine in this project, we can start deploying our application:

```
  public class TransitiveClosure
  {
      public static void Main(string[] args)
      {
          try
          {
              InputProgram input = new DatalogInputProgram();
              DesktopHandler handler = new DesktopHandler(new IDLVDesktopService(
↪"executables/idlv"));
              DatalogMapper.Instance.RegisterClass(typeof(Path));
              input.AddProgram("path(X,Y) :- edge(X,Y).");
              input.AddProgram("path(X,Y) :- path(X,Z), path(Z,Y).");
              handler.AddProgram(input);

              input.AddObjectInput(new Edge(1,2));
              input.AddObjectInput(new Edge(2,3));
              input.AddObjectInput(new Edge(2,4));
              input.AddObjectInput(new Edge(3,5));
              input.AddObjectInput(new Edge(3,6));

              IDLVMinimalModels minimalModels = (IDLVMinimalModels)handler.
↪StartSync();

              foreach (MinimalModel m in minimalModels.Minimalmodels)
              {
                  foreach (object a in m.Atoms)
                  {
                      if (typeof(Path).IsInstanceOfType(a))
                      {
                          Console.WriteLine(a);
```

(continues on next page)

(continued from previous page)

```
                        }
                    }
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }

}
```

The main method contains an `Handler` instance, that is initialized with a `DesktopHandler` using the parameter `IDLVDesktopService` with a string representing the path to the IDLV local grounder.

The `DatalogMapper` registers the classes created before in order to manage the input and output objects.

A string and a list of `Edge` objects representing facts, rules and constraints of the Datalog program are added to an `DatalogInputProgram`, and the `DatalogInputProgram` is added to the `Handler`.

Finally the solver is invoked, and the output is retrieved.

In this example the `Path` predicates, represent all the arcs in the transitive closure of the starting graph. The output predicates can be managed accordingly to the user's desiderata, as they are simply Objects.

For further information, contact embasp@mat.unical.it or visit our website.

2.10 Sudoku Android

2.10.1 Getting started

In order to use the framework in your applications you have to import it as module on Android Studio

1. Import the framework module:

- Download the framework last released module.
- In the project view, right-click on your project New > Module.
- Select Import .JAR/.AAR Package.
- Select the directory in which the module has been downloaded.

2. Set the dependency:

- In the Android Studio menu: File > Project Structure .
- Select your project module (by default called app).
- In the Dependencies Tab add as Module Dependency the previously imported framework.

2.10.2 Using EmbASP

In the following, we describe an actual usage of the framework by means of a running example; as a use case, we will develop a simple Android application for solving Sudoku puzzles.

The complete code of this example is freely available here.

The framework features a annotation-guided mapping, offered by the `ASPMapper` component, for two-way translations between strings recognizable by ASP solvers and objects in the programming language at hand, directly employable within applications. By means of this feature, the ASP-based aspects can be separated from the Narrative coding: the programmer doesn't even necessarily need to be aware of ASP.

Let us think of a user that designed (or has been given) a proper logic program P to solve a sudoku puzzle and has also an initial schema. We assume that the initial schema is well-formed i.e. the complete schema solution exists and is unique. A possible program P is embedded in the complete example, that, coupled with a set of facts F representing the given initial schema, allows to obtain the only admissible solution.

By means of the annotation-guided mapping, the initial schema can be expressed in forms of Narrative objects. To this extent, we define the class `Cell`, aimed at representing the single cell of the sudoku schema, as follows:

```
@Id("cell")
public class Cell {

  @Param(0)
  private int row;

  @Param(1)
  private int column;

  @Param(2)
  private int value;

  [...]

}
```

It is worth noticing how the class has been annotated by two custom annotations, defined according to the following syntax:

- *@Id(string_name)* : the target must be a class, and defines the predicate name the class is mapped to;
- *@Param(integer_position)* : the target must be a field of a class annotated via *@Id*, and defines the term (and its position) in the ASP atom the field is mapped to.

Thanks to these annotations the `ASPMapper` class will be able to map `Cell` objects into strings properly recognizable from the ASP solver as logic facts of the form *cell(Row,Column,Value)*. At this point, we can create an Android Activity Component , and start deploying our sudoku application:

```
public class MainActivity extends AppCompatActivity {

  [...]

  private Handler handler;

  @Override
  protected void onCreate(Bundle bundle) {
    handler = new AndroidHandler(getApplicationContext(), DLVAndroidService.class);
    [...]
  }
```

(continues on next page)

(continued from previous page)

```
  public void onClick(final View view){
    startReasoning();
    [...]
  }

  public void startReasoning() {
    InputProgram inputProgram = new InputProgram();
    for (int i = 0; i < 9; i++){
      for (int j = 0; j < 9; j++)
        try {
          if(sudokuMatrix[i][j]!=0) {
            inputProgram.addObjectInput(new Cell(i, j, sudokuMatrix[i][j]));
          }
        } catch (Exception e) {
          // Handle Exception
        }
    }
    handler.addProgram(inputProgram);

    String sudokuEncoding = getEncodingFromResources();
    handler.addProgram(new InputProgram(sudokuEncoding));

    Callback callback = new MyCallback();
    handler.startAsync(callback);
  }
}
```

The class contains an `Handler` instance as field, that is initialized when the Activity is created as an `AndroidHandler`. Required parameters include the Android Context (an Android utility, needed to start an Android Service Component) and the type of `AndroidService` to use, in our case a `DLVAndroidService`.

In addiction, in order to represent an initial sudoku schema, the class features a matrix of integers as another field where position (i,j) contains the value of cell (i,j) in the initial schema; cells initially empty are represented by positions containing zero.

The method startReasoning is in charge of actually managing the reasoning: in our case, it is invoked in response to a click event that is generated when the user asks for the solution. It is firstly created an `InputProgram` object that is filled with `Cell` objects representing the initial schema, which is then provided to the handler; then it is provided with the sudoku encoding. It could be loaded, for instance, by means of an utility function that retrieves it from the Android Resources folder, which, within Android applications, is typically meant for containing images, sounds, files and resources in general.

At this point, the reasoning process can start; since for Android we provide only the asynchronous execution mode, a `Callback` object is in charge of fetching the output when the ASP system has done.

Finally, once the computation is over, from within the *callback* function the output can be retrieved directly in form of Narrative objects. For instance, in our case an inner class `MyCallback` implements the interface `Callback`:

```
private class MyCallback implements Callback {

  @Override
  public void callback(Output o) {
    if(!(o instanceof AnswerSets))
      return;
    AnswerSets answerSets=(AnswerSets)o;
    if(answerSets.getAnswersets().isEmpty())
```

(continues on next page)

(continued from previous page)

```
      return;
    AnswerSet as = answerSets.getAnswersets().get(0);
    try {
      for(Object obj:as.getAtoms()) {
        Cell cell = (Cell) obj;
        sudokuMatrix[cell.getRow()][cell.getColumn()] = cell.getValue();
      }
    } catch (Exception e) {
      // Handle Exception
    }
    displaySolution();
  }
}
```

For further information, contact embasp@mat.unical.it or visit our website.

2.11 Desktop ASP exemples

- *Shortest-path ASP Narrative*
- *Shortest-path ASP Theoretic*
- *Shortest-path ASP Programmatic*

2.12 Desktop PDDL examples

- *Blocks-world PDDL Narrative*
- *Blocks-world PDDL Theoretic*
- *Blocks-world PDDL Programmatic*

2.13 Desktop Datalog examples

- *Transitive Closure Datalog Narrative*
- *Transitive Closure Datalog Theoretic*
- *Transitive Closure Datalog Programmatic*

2.14 Android example

- *Sudoku Android*

CHAPTER

THREE

CONTACTS

For further information, contact embasp@mat.unical.it or visit our website.