A Framework for Easing the Development of Applications Embedding Answer Set Programming

Davide Fuscà  Stefano Germano  Jessica Zangari
Marco Anastasio  Francesco Calimeri  Simona Perri

Department of Mathematics and Computer Science, University of Calabria
Via Bucci, Cubo 30B
87036 Rende, Italy
embasp@mat.unical.it

ABSTRACT

Answer Set Programming (ASP) is a well-established declarative problem solving paradigm which became widely used in AI and recognized as a powerful tool for knowledge representation and reasoning (KRR), especially for its high expressiveness and the ability to deal also with incomplete knowledge.

Recently, thanks to the availability of a number of robust and efficient implementations, ASP has been increasingly employed in a number of different domains, and used for the development of industrial-level and enterprise applications. This made clear the need for proper development tools and interoperability mechanisms for easing interaction and integration with external systems in the widest range of real-world scenarios, including mobile applications and educational contexts.

In this work we present a framework for integrating the KRR capabilities of ASP into generic applications. We show the use of the framework by illustrating proper specializations for some relevant ASP systems over different platforms, including the mobile setting; furthermore, the potential of the framework for educational purposes is illustrated by means of the development of several ASP-based applications.

CCS Concepts

•Social and professional topics → Computer science education; Computing education; •Computing methodologies → Artificial intelligence; Knowledge representation and reasoning; Logic programming and answer set programming; •Theory of computation

→ Logic; Automated reasoning; •Applied computing → Education; •Software and its engineering → Software notations and tools;

Keywords

Answer Set Programming; Logic Programs; Education; Industrial Applications; Knowledge Representation and Reasoning; Object-Oriented Programming; Software Development; Complex Systems; Embedded Systems; Artificial Intelligence

1. INTRODUCTION

Answer Set Programming (ASP) [1, 2, 9, 10, 18, 23, 24] is a purely declarative formalism for knowledge representation and reasoning developed in the field of logic programming and nonmonotonic reasoning. The language of ASP is based on rules, allowing (in general) for both disjunction in rule heads and nonmonotonic negation in the body.

The idea of answer set programming is to represent a given computational problem by the means of a logic program whose intended models, called answer sets, correspond one-to-one to solutions; hence, an answer set solver can be used in order to actually find such solutions [22]. The term “Answer Set Programming” was introduced by Vladimir Lifschitz to denote a declarative programming methodology [22], concerning terminology, ASP is sometimes used in a somewhat broader sense, referring to any declarative formalism which represents solutions as sets. However, the more frequent understanding is the one adopted in this article, which dates back to [18]. For introductory material on ASP, we refer to [1, 17, 22, 23].

After more than twenty years of research, the theoretical properties of ASP are well understood and the solving technology, as evidenced by the availability of a number of robust and efficient systems [7], is mature for practical applications: ASP has been increasingly employed in many different domains, and also used for the development of industrial-level and enterprise applications [8, 21]. Notably, this is spreading ASP teaching in universities worldwide, and, interestingly, is moving the focus from a strict theoretical scope to more practical aspects. Moreover, it makes clear the need for proper tools and interoperability mechanisms that ease the
development of ASP-based applications, in both educational and real-world contexts.

In this work, we present a framework for the integration of ASP in external systems for generic applications; it consists of an abstract architecture, implementable in a programming language of choice, that easily allows for proper specializations to different platforms and ASP reasoners.

The framework features explicit mechanisms for two-way translations between strings recognizable by ASP solvers and objects in the programming language at hand, directly employable within applications. This gives developers the possibility to work separately on ASP-based modules and on applications that makes use of them, and keeps things simple when developing complex applications. Let us think, for instance, of a scenario in which different figures are involved, such as Android/Java developers and KRR experts. Both figures can take advantage from the fact that the knowledge base and the reasoning modules can be designed and developed independently from the rest of the Java-based application.

In order to illustrate the use of the framework, we present here an actual Java implementation; in addition, we introduce two specialized libraries for DLV [20] and clingo [15], two state-of-the-art ASP systems, on mobile and desktop platforms, respectively. Furthermore, we show some applications developed in an educational context, that prove the effectiveness of the framework.

2. ANSWER SET PROGRAMMING

In this section, we briefly recall syntax and semantics of Answer Set Programming.

It is worth recalling that a significant amount of work has been carried out by the scientific community for extending the basic language, in order to increase the expressive power and improve usability of the formalism. This has led to a variety of ASP “dialects”, supported by a corresponding variety of ASP systems, that only share a portion of the basic language. Notably, the community recently agreed on the definition of a standard input language for ASP systems, namely ASP-Core-2 [4], which is also the official language of the ASP Competition series [16]; it features most of the advanced constructs and mechanisms with a well-defined semantics that have been introduced and implemented in the latest years.

For the sake of simplicity, we focus next on the basic aspects of the language; for a complete reference to the ASP-Core-2 standard, and further details about advanced ASP features, we refer the reader to [1] and the vast literature.

2.1 Syntax

A variable or a constant is a term. An atom is \(a(t_1, \ldots, t_n)\), where \(a\) is a predicate of arity \(n\) and \(t_1, \ldots, t_n\) are terms. A literal is either a positive literal or a negative literal \(\neg p\), where \(p\) is an atom. A disjunctive rule (rule, for short) \(r\) is a formula

\[
a_1 \mid \cdots \mid a_n \leftarrow b_1, \ldots, b_k, \not b_{k+1}, \ldots, \not b_m.
\]

where \(a_1, \ldots, a_n, b_1, \ldots, b_k, b_{k+1}, \ldots, b_m\) are atoms and \(n \geq 0, m \geq k \geq 0\). The disjunction \(a_1 \mid \cdots \mid a_n\) is the head of \(r\), while the conjunction \(b_1, \ldots, b_k, \not b_{k+1}, \ldots, \not b_m\) is the body of \(r\). A rule without head literals (i.e. \(n = 0\)) is usually referred to as an integrity constraint. If the body is empty (i.e. \(k = m = 0\)), it is called a fact.

\(H(r)\) denotes the set \(\{a_1, \ldots, a_n\}\) of the head atoms, and by \(B(r)\) the set \(\{b_1, \ldots, b_k, \not b_{k+1}, \ldots, \not b_m\}\) of the body literals. \(B^+(r)\) (resp., \(B^-(r)\)) denotes the set of atoms occurring positively (resp., negatively) in \(B(r)\). A rule \(r\) is safe if each variable appearing in \(r\) appears also in some positive body literal of \(r\).

An ASP program \(\mathcal{P}\) is a finite set of safe rules. An atom, a literal, a rule, or a program is ground if no variables appear in it. Accordingly with the database terminology, a predicate occurring only in facts is referred to as an EDB predicate, all others as IDB predicates; the set of facts of \(\mathcal{P}\) is denoted by \(EDB(\mathcal{P})\).

2.2 Semantics

Let \(\mathcal{P}\) be a program. The Herbrand Universe and the Herbrand Base of \(\mathcal{P}\) are defined in the standard way and denoted by \(UP\) and \(BP\), respectively.

Given a rule \(r\) occurring in \(\mathcal{P}\), a ground instance of \(r\) is a rule obtained from \(r\) by replacing every variable \(X\) in \(r\) by \(\sigma(X)\), where \(\sigma\) is a substitution mapping the variables occurring in \(r\) to constants in \(UP\); \(ground(\mathcal{P})\) denotes the set of all the ground instances of the rules occurring in \(\mathcal{P}\).

An interpretation for \(\mathcal{P}\) is a set of ground atoms, that is, an interpretation is a subset \(I\) of \(BP\). A ground positive literal \(A\) is true (resp., false) w.r.t. \(I\) if \(A \in I\) (resp., \(A \notin I\)). A ground negative literal \(\not A\) is true w.r.t. \(I\) if \(A\) is false w.r.t. \(I\); otherwise \(\not A\) is false w.r.t. \(I\). Let \(r\) be a rule in \(\mathcal{P}\); \(r\) is true w.r.t. \(I\) if \(H(r)\) is true w.r.t. \(I\) if \(H(r)\) is false w.r.t. \(I\) if \(I\) is a model for \(\mathcal{P}\) for \(\mathcal{P}\) such that every rule \(r\) in \(ground(\mathcal{P})\) is true w.r.t. \(M\). A model \(M\) for \(\mathcal{P}\) is minimal if no model \(N\) for \(\mathcal{P}\) exists such that \(N\) is a proper subset of \(M\). The set of all minimal models for \(\mathcal{P}\) is denoted by \(MM(\mathcal{P})\).

Given a ground program \(\mathcal{P}\) and an interpretation \(I\), the reduct of \(\mathcal{P}\) w.r.t. \(I\) is the subset \(\mathcal{P}^I\) of \(\mathcal{P}\), which is obtained from \(\mathcal{P}\) by deleting rules in which a body literal is false w.r.t. \(I\). Note that the above definition of reduct, proposed in [11], simplifies the original definition of Gelfond-Lifschitz (GL) transform [18], but is fully equivalent to the GL transform for the definition of answer sets [11].

Let \(I\) be an interpretation for a program \(\mathcal{P}\). \(I\) is an answer set (or stable model) for \(\mathcal{P}\) if \(I \in MM(\mathcal{P}^I)\) (i.e., \(I\) is a minimal model for the program \(\mathcal{P}^I\)) [26] [18]. The set of all answer sets for \(\mathcal{P}\) is denoted by \(ANS(\mathcal{P})\).

2.3 Knowledge Representation and Reasoning with ASP

In the following, we briefly introduce the use of ASP as a tool for knowledge representation and reasoning, and show how its fully declarative nature allows to encode a large variety of problems via simple and elegant logic programs.

The examples below have been implemented adhering to the “Guess&Check” (GC) paradigm [9], one of the most common ASP programming methodology. In summary, a GC program features 2 modules:

- a Guessing Part, that defines the search space (for
instance, by means of disjunctive rules);

- a Checking Part (optional), that checks solution admissibility (usually, by means of integrity constraints).

When dealing with optimization problems, the methodology can be further extended to match a “Guess/Check/Optimize” (GCO) paradigm: ad-hoc means for expressing preferences among answer sets are employed, such as weak constraints, thus implementing the

- Optimizing Part (optional), that specifies preference criteria.

[3-COL] As a first example, let us consider the well-known problem of 3-colorability, which consists of the assignment of three colors to the nodes of a graph in such a way that adjacent nodes always have different colors. This problem is known to be NP-complete.

Suppose that the nodes and the arcs are represented by a set $F$ of facts with predicates node (unary) and arc (binary), respectively. Then, the following ASP program allows us to determine the admissible ways of coloring the given graph.

$$
\begin{align*}
    r_1 : & \quad \text{color}(X,r) \land \text{color}(X,y) \land \text{color}(X,g) \leftarrow \text{node}(X). \\
    r_2 : & \quad \neg \text{arc}(X,Y), \text{color}(X,C), \text{color}(Y,C).
\end{align*}
$$

Rule $r_1$ (guess) above states that every node of the graph must be colored as red or yellow or green; $r_2$ (check) forbids the assignment of the same color to any couple of adjacent nodes. The minimality of answer sets guarantees that every node is assigned only one color. Thus, there is a one-to-one correspondence between the solutions of the 3-coloring problem for the instance at hand and the answer sets of $F \cup \{r_1,r_2\}$: the graph represented by $F$ is 3-colorable if and only if $F \cup \{r_1,r_2\}$ has some answer set.

We have shown how it is possible to deal with a problem by means of an ASP program such that the instance at hand has some solution if and only if the ASP program as some answer set; in the following, we show an ASP program whose answer sets witness that a property does not hold, i.e., the property at hand holds if and only if the program has no answer sets.

[RAMSEY] The Ramsey Number $R(k, m)$ is the least integer $n$ such that, no matter how we color the arcs of the complete graph (clique) with $n$ nodes using two colors, say red and blue, there is a red clique with $k$ nodes (a red $k$-clique) or a blue clique with $m$ nodes (a blue $m$-clique).

Similarly to what already described above, let $F$ be the collection of facts for input predicates node (unary) and edge (binary), encoding a complete graph with $n$ nodes; then, the following ASP program $P_{R(3,4)}$ allows to determine whether a given integer $n$ is the Ramsey Number $R(3,4)$, knowing that no integer smaller than $n$ is $R(3,4)$.

$$
\begin{align*}
    r_1 : & \quad \text{blue}(X,Y) \lor \text{red}(X,Y) \leftarrow \text{edge}(X,Y). \\
    r_2 : & \quad \neg \text{red}(X,Y), \text{red}(X,Z), \text{red}(Y,Z). \\
    r_3 : & \quad \neg \text{blue}(X,Y), \text{blue}(X,Z), \text{blue}(Y,Z), \\
             & \quad \text{blue}(X,W), \text{blue}(Y,W), \text{blue}(Z,W).
\end{align*}
$$

Intuitively, the disjunctive rule $r_1$ guesses a color for each edge. The first constraint $r_2$ eliminates the colorings containing a red complete graph (i.e., a clique) on 3 nodes; the second constraint $r_3$ eliminates the colorings containing a blue clique on 4 nodes. The program $P_{R(3,4)} \cup F$ has an answer set if and only if there is a coloring of the edges of the complete graph on $n$ nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular $n$, then $n$ is not $R(3,4)$, that is, $n < R(3,4)$. The smallest $n$ such that no answer set is found is the Ramsey Number $R(3,4)$.

Eventually, let us show how ASP can be applied for solving puzzles.

[SUDOKU] A classic Sudoku puzzle consists of a tableau featuring 81 cells, or positions, arranged in a 9×9 grid, which is divided into nine sub-tableaux (regions, or blocks) containing nine positions each. Initially, a number of positions (between 17 and 35) are filled with a number picked up in the range 1…9. The aim of the game is to check whether every empty position can be filled with a number between 1 and 9 in such a way that each row, column and block show all digits from 1 to 9 exactly once.

Let us suppose that a set of facts $F$ is given, representing the schema to be completed; in particular, a binary predicate pos encodes possible position coordinates; symbol is a unary predicate encoding possible symbols (numbers); facts of the form sameblock($x_1,y_1,x_2,y_2$) state that two positions ($x_1,y_1$) and ($x_2,y_2$) are within the same block; facts of the form cell($x,y,n$) represent a position ($x,y$) is filled with symbol $n$.

We show next an ASP program $P_{sudoku}$ such that the answer sets of $P_{sudoku} \cup F$ correspond to the solutions of the Sudoku schema at hand; note that, in general, well-founded sudoku instances have only one solution, and thus $P_{sudoku} \cup F$ will have a single answer set.

$$
\begin{align*}
    r_1 : & \quad \text{cell}(X,Y,N) \land \neg \text{cell}(X,Y,N) \leftarrow \text{pos}(X), \\
             & \quad \text{pos}(Y), \text{symbol}(N). \\
    r_2 : & \quad \neg \text{cell}(X,Y,N), \text{cell}(X,Y,N),N1 \leftrightarrow N. \\
    r_3 : & \quad \text{assigned}(X,Y) \leftarrow \text{cell}(X,Y,N). \\
    r_4 : & \quad \neg \text{pos}(X), \text{pos}(Y), \text{not-assigned}(X,Y). \\
    r_5 : & \quad \neg \text{cell}(X,Y,Z), \text{cell}(X,Y,Z),Y1 \leftrightarrow Y2. \\
    r_6 : & \quad \text{cell}(X,Y,Z), \text{cell}(X,Y,Z),X1 \leftrightarrow X2. \\
    r_7 : & \quad \text{cell}(X_1,Y_1,Z), \text{cell}(X_2,Y_2,Z),Y1 \leftrightarrow Y2, \\
             & \quad \text{sameblock}(X_1,Y_1,X_2,Y_2). \\
    r_8 : & \quad \text{cell}(X_1,Y_1,Z), \text{cell}(X_2,Y_2,Z),X1 \leftrightarrow X2, \\
             & \quad \text{sameblock}(X_1,Y_1,X_2,Y_2).
\end{align*}
$$

Rules $r_1$ – $r_4$ guess the number for each cell, ensuring that each cell is filled exactly one number (symbol); note that the guessed values for the positions complete the extension of the predicate cell for which some values have been already provided in $F$. Rules $r_5$ and $r_6$ check that a number does not occur more than once in the same row or column, respectively; rules $r_7$ and $r_8$, finally, ensure that two different cells in the same block don’t have the same number.
3. THE FRAMEWORK

In this section we introduce EMBASP, an abstract framework for the integration of ASP in external systems for generic applications; then, we propose a Java implementation.

The general architecture of EMBASP is depicted in Figure 1: it defines an abstract framework to be implemented in some object-oriented programming language. Due to its abstract nature, Figure 1 just reports the general dependencies among the main modules. Nevertheless, each concrete implementation might require specific dependencies among the inner components of each module, as can be observed in Figure 2, which is related to a concrete Java implementation and will be discussed hereafter.

It is worth noting that the framework design is intended to ease and guide the generation of suitable libraries for the use of specific solvers on particular platforms; resulting applications manage ASP solvers as “black boxes”. On the one hand, this might lead to issues arising from users demanding for a more interactive white-box usage; on the other hand, this made us able to keep a clean design that grants an intuitive usage and an architecture which is general and easily adaptable to different platforms and reasoners. The resulting libraries can hence be used in order to effectively embed ASP reasoning modules, handled by the ASP system(s) at hand, within any kind of application developed for the targeted platforms. In addition, as already discussed above, the framework is meant to give developers the possibility to work separately on ASP-based modules and on the applications that makes use of them, thus keeping things simple when developing complex applications. Additional specific advantages/disadvantages might arise depending on the programming language chosen for deploying libraries and on the target platform; special features, indeed, can make implementation, and in turn extensions and usage, easier or more difficult, to different extents. We will briefly discuss these issues later on.

Abstract Architecture

The framework architecture has been designed by means of four modules: Core, Platforms, ASP Language, and Systems, whose indented behaviour is described next.

Core Module. The Core module defines the basic components of the Framework.
The Handler component mediates the communication between the Framework and the user that can provide it with the input program(s) via the component Input Program, along with any desired solver’s option(s) via the component Option Descriptor. A Service component is meant for managing the chosen ASP solver executions.

Two different execution modes can be made available: synchronous or asynchronous. While in the synchronous mode any call to the execution of the ASP solver is blocking (i.e., the caller waits until the reasoning task is completed), in asynchronous mode the call is non-blocking: a Callback component notifies the caller once the reasoning task is completed. The result of the execution (i.e., the output of the ASP system) is handled by the Output component, in both modes.

Platforms Module. The Platforms module is meant for containing what is platform-dependent; in particular, the Handler and Service components from the Core module that should be adapted according to the platform at hand, since they take care of practically launching solvers.

ASP Language Module. The ASP Language module defines specific facilities for ASP; in particular, components AnswerSet and AnswerSets adapt Output component to the ASP case. Moreover, an additional component, namely the ASPMapper, is conceived as an utility for managing input and output via objects, if the programming language at hand permits it.

Systems Module. The Systems module defines what is system-dependent; in particular, the Input Program, Output and Option Descriptor components from the Core module should be adapted in order to effectively interact with the ASP system at hand.

Implementing embASP

In the following, we propose a Java implementation of the architecture described above, along with proper specializations for two of the state-of-the-art ASP systems. In particular, we implemented the main modules by means of classes or interfaces, and we created specialized libraries that permit the use of DLV on Android and clingo on desktop (i.e., any java-enabled desktop for which clingo is available).

Figure 2 provides some details about classes and interfaces of the implementation. For the sake of presentation, we do not report the complete UML class diagram, which is quite involved; rather, we illustrate a simplified version. Although methods inside classes have been omitted to further improve readability, adopted connectors follow UML syntax. In order to better outline correspondences with the abstract architecture of Figure 1, classes belonging to a module have been grouped together. The complete UML class diagram is available online at [6].

Core module implementation

Each component in the Core module has been implemented by means of an homonymous class or interface. 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.

Platforms module implementation

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

1https://www.oracle.com/java
2http://developer.android.com
As for the Android platform, we developed an Android-Handler that handles the execution of an AndroidService, which provides facilities to manage the execution of an ASP reasoner on the Android platform.

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

While both synchronous and asynchronous modes are provided in the desktop setting, we stick to the asynchronous one on Android: indeed, mobile users are familiar with apps featuring constantly reactive graphic interfaces, and according to this native asynchronous execution policy, we want to discourage a blocking execution.

**ASP Language module implementation**

This module includes specific classes for the management of input and output to ASP solvers. In particular, ASPInputProgram extends InputProgram to the ASP case. In addition, since the “result” of an ASP solver execution consists of answer sets, the Output class has been extended by the AnswerSets class that is composed by a set of AnswerSet objects.

Moreover, the module features an ASPMapper class, that acts like a translator, providing proper means for a two-way translation between strings recognizable by the ASP solver and Java objects directly employable within the application. The ASPMapper is intended at translating ASP input and output from and to objects: thus has a dependency from ASPInputProgram and AnswerSets classes.

In our implementation, such translations are guided by Java Annotations, a form of metadata that mark Java 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. They have a number of uses, such as directions to the compiler, compile-time and deployment-time processing, or runtime processing. For more details, we refer the reader to the Java documentation.

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:

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

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

The user has to register all its annotated classes to the ASPMapper, although classes involved in input translation are automatically detected. If the classes intended for the translation are not annotated or not correctly annotated, an exception is raised. Other problems might occur if once that the solver output is returned, the user asks for a translation into objects of not annotated classes: in this case a warning is raised and the request is ignored.

Notably, such feature is meant to give developers the possibility to work separately on the ASP-based modules and on the Java side. The mapper acts like a middleware that enables the communication among the modules, and eases the burden of developers by means of an explicit, ready-made mapping between Java objects and the logic modules.

Further insights about this feature are illustrated thanks to an example in the next section.

**Systems Module Implementation**

The classes DLVAnswerSets and ClingoAnswerSets implement specific extensions of the AnswerSets class, in charge of manipulating the output of the respective solvers.

Moreover, this module also contains classes extending OptionDescriptor to implement specific options of the solver at hand. For instance, the class DLVFilter is a utility class representing the filter option of DLV.

**Specializing the Framework**

We implemented two libraries derived from EMBASP, allowing the embedding of ASP reasoning modules handled by DLV and clingo, from within Android and desktop applications, respectively.

The classes DLVAndroidService and ClingoDesktopService are in charge of this task.

DLVAndroidService is a specific version of AndroidService for the execution of DLV on Android. It is worth noting that DLV was not available for Android; furthermore, it is natively implemented in C++, while the standard development process on Android is based on Java. To this end, DLV has been on purpose rebuilt using the NDK (Native Development Kit) and has been linked to the Java code using the JNI (Java Native Interface). This grants the access to the APIs provided by the Android NDK, and in turn accelerates to the DLV exposed functionalities directly from the Java code of an Android application.

Similarly, ClingoDesktopService is a specific version tailored for the clingo reasoner on the desktop platform, extending the DesktopService with proper functions needed to invoke clingo. In this case, different versions of the solver for several desktop OSes were already available online.

**4. EMBEDDING ASP PROGRAMS**

In the following we show the use of the specialized Java libraries generated via EMBASP for developing Android applications; we report some considerations about programming languages different from Java at the end of the section.

As a use case, we will refer to an application for solving Sudoku puzzles. We will report the code related to the EMB-ASP usage; the complete code is available online.

Notably, thanks to the annotation-guided mapping, the ASP-based aspects can be separated from the Java coding: the programmer does not 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

3

http://docs.oracle.com/javase/8/docs/technotes/guides/jni

4

[25]

[14]

4
is well-formed i.e. the complete schema solution exists and is unique. For instance, \( P \) can correspond to the logic program presented in Section 2.3 so that, coupled with a set of facts \( F \) representing the given initial schema, allows to obtain the only admissible solution (i.e., a single answer set).

It is worth remembering that, in case of less usual sudoku schemata featuring multiple solutions, the ASP program features multiple answer sets, one-to-one corresponding to such solutions.

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

```java
public class Cell {
    private int row;
    private int column;
    private int value;
}
```

It is worth noticing how the class has been annotated by two custom annotations, as introduced above. Thanks to these annotations the `ASPMapper` 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 `ActivityComponent` and start deploying our sudoku application:

```java
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_main);
    }

    @Override
    public void onClick(View view){
        startReasoning();
    }

    public void startReasoning() {
        InputProgram inputProgram = new ASPIInputProgram();
        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);
    }
}
```

The class contains a `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 addition, 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. Lines 19–32 create an `InputProgram` object that is filled with `Cell` objects representing the initial schema, which is then served to the handler; lines 34–37 provide it with the sudoku encoding. It could be loaded, for instance, by means of a 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 (Lines 39–40).

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

```java
private class MyCallback implements Callback {
    @Override
    public void callback(Output o) {
        if(!((o instanceof AnswerSets))) {
            return;
        } else {
            AnswerSets answerSets = (AnswerSets) o;
            if(answerSets.getAnswerSets().isEmpty()) {
                return;
            } else {
                AnswerSet as = answerSets.getAnswerSets().get(0);
                try {
                    for(Object obj: as.getAtoms()) {
                        if(obj instanceof Cell) {
                            Cell cell = (Cell) obj;
                            try {
                                if(cell.getValue() == 0) {
                                    inputProgram.addObjectInput(
                                        new Cell(i, j, sudokuMatrix[i][j]));
                                } catch (Exception e) {
                                    // Handle Exception
                                }
                            } catch (Exception e) {
                                // Handle Exception
                            }
                        }
                    }
                } catch (Exception e) {
                    // Handle Exception
                }
            }
        }
    }
}
```

4.1 Other Language Implementations of EMB-ASP

The implementation illustrated above relies on Java. Besides the fact that it represents a very popular, solid and re-
liable programming language, the choice was also motivated by the intention to foster the use of ASP in new scenarios, and in particular in the mobile one; Android is by far the most widespread mobile platform, and its development and deployment models heavily rely on Java. However, as already stated, the abstract architecture of emBASP can be made concrete by means of other object-oriented programming languages. A thorough discussion about different language implementations is out of the scope of this work; however, we briefly discuss in the following some interesting possible approaches.

Most of components in the herein presented Java implementation have been accomplished thanks to features that are typical of any object-oriented language, such as inheritance and polymorphism. The unique exception is represented by the ASPMapper component, implemented by means of Java peculiar features, such as annotations and reflection. In case of other languages that feature similar constructs, such as C++, the approach can resemble the herein presented Java implementation.

With different languages that lack such features, the mapping mechanism can still be implemented with a simulation via inheritance and polymorphism and applying typical Software Engineering patterns [13]. As a matter of example, one possible implementation can be accomplished using the Prototype design pattern, that results well-suited to our purposes, as it allows to “specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype” [13]. Such pattern can be the key to simulate the dynamical loading of classes in languages that do not support it natively, as it happens with C++. Indeed, the run-time environment can make use of it in order to automatically create an instance of each class when it’s loaded, and then register the instance with a prototype manager – in our case, represented by the ASPMapper component. All classes that in Java (or similar languages) would make use of reflection and annotations, can be defined by extending a properly defined Prototype class and then specify how to map predicates and terms. Moreover, a class ASPMapper would still be needed, with a behaviour quite similar to the Java case.

5. ASP-BASED APPLICATIONS: SOME EXAMPLES IN THE EDUCATIONAL SETTING

In this section we describe some ASP-based applications developed by means of emBASP for educational purposes, and, in particular, in the context of a university course that covers ASP topics: it is worth noting that such applications have been developed by some of the course attendants, i.e., undergraduate students. The educational aspect here is two-folded. The most relevant is the engagement of university (under)graduate students in ASP capabilities, in order to make them able to take advantage from it when solving problem and designing solutions, in the broadest sense. Furthermore, ASP looks well-fitted for the use in the development of educational/training software, as, for instance, the DLVEdu app introduced below; a deeper study of such aspects, however, is out of the scope of the present work.

In the following, we first briefly introduce three applications; then, in order to further clarify the emBASP use, especially in the mobile setting, we describe the DLVfit Android App more in detail.

GuessAndCheckers

GuessAndCheckers is a native mobile application that works as an helper for users that play “live” games of the (Italian) checkers (i.e., by means of physical board and pieces). The app, that runs on Android, can help a player at any time; by means of the device camera a picture of the board is taken, and the information about the current status of the game is properly inferred thanks to OpenCV[9] an open source computer vision and machine learning software; an ASP-based artificial intelligence module then suggests the move.

Thanks to emBASP and the use of ASP, GuessAndCheckers features a fully-declarative approach that made easy to develop and improve several different strategies, also experimenting with many combinations thereof.

The source code of this application along with the Android Application Package (APK) are available online; more details can be found at [6].

DLVEdu

DLVEdu is an educational Android App for children, that integrates well-established mobile technologies, such as voice or drawn text recognition, with the modeling capabilities of ASP. In particular, it is able to guide the child through-out the learning tasks, by proposing a series of educational games, and developing a personalized educational path. The games are divided into four macro-areas: Logic, Numeric-Mathematical, Memory, and Verbal Language. The usage of ASP allows the application to adapt to the game experiences fulfilled by the user, her formative gap, and the obtained improvements.

The application continuously profiles the user by recording mistakes and successes, and dynamically builds and updates a customized educational path along the different games.

The application features a “Parent Area”, that allows parents to monitor child’s achievements and to express some preferences, such as desired express directions in order to grant/forbid access to some games or educational areas.

Connect4

The popular turn-based Connect Four game is played on a vertical 7*6 rectangular board, where two opponents drop their disks with the aim of creating a line of four, either horizontally, vertically, or diagonally.

The Connect4 application allows a user to play the game (also known as Four-in-a-Row) against an ASP-based artificial player. Notably, the declarative nature of ASP, its expressive power, and the possibility to compose programs by selecting proper rules, allowed to design and implement different AIs, ranging from the most powerful one, that implements advanced techniques for the perfect play, to the simplest one, that relies on some classical heuristic strategies. Furthermore, by using emBASP, two different versions of the same app have been built: one for Android, making use of DLV, and one for java-enabled desktop platforms, making use of clingo.


**DLVfit**

The DLVfit Android App was the first application making use of the framework; it was conceived as a proof of concept, in order to show the framework features and capabilities. To our knowledge, it is also the first mobile app natively running an ASP solver.

DLVfit is a health app that aims at suggesting the owner of a mobile device the “best” way to achieve some fitness goals. The app lets the user express her own goals and preferences in a very customizable way along many combinable dimensions: calories to burn, time to spend, differentiation over several physical activities, time constraints, etc. Then, it monitors her actual activity throughout the day and, upon request, it computes one or more plans meant, if accomplished, to make her meet the aforementioned goals the way she would have preferred.

More in detail, the app constantly detects the current user activity (running, walking, cycling, etc.) and (at a customizable frequency) stores some information (activity type, timestamps, calories burned up to the present time, etc.). Activity detection is performed by means of the Google Activity Recognition APIs [19], a de-facto standard on Android, thus relying on these for the accuracy of detection. As already mentioned, the user might ask, at any time, for a suggestion about a plan for the rest of the day; the reasoning module hence prepares a (set of) proper workout plans complying with the very personal goals and preferences previously expressed.

The user interacts with the app via a standard graphical interface; the reasoning module is actually in charge of building a proper ASP program, which is in turn fed to DLV via embASP. Such program matches the classical “Guess/Check/Optimize” paradigm introduced in Section 2 and thus resulting easy to understand, enrich and customize:

- the “guess” part chooses how much time to spend on each exercise;
- the “check” part forces the resulting plan to be admissible: burning the remaining amount of desired calories, do not exceed the time constraints, etc.;
- the “optimize” part, eventually, expresses preferences: minimize total time spent exercising, number of activities to perform, maximize the number of different activity types, avoid activities around a given time of the day, etc.

The logic program used takes as “input” (i.e., a set of facts as instances of proper predicates):

- **calories_burnt_per_activity(A, C)**
  the calories burnt (C), in each unit of time, per each Activity (A);
- **remaining_calories_to_burn(R)**
  the remaining calories to burn in the rest of the current day;
- **how_long(A, D)**
  the amount of time that can be spent for each activity A (in order to reach the goal of burn all the remaining calories);
- **max_time(T)**
  the duration of the workout (max: the remaining time to the end of day);

surplus(C)

the maximum surplus of calories to burn with the suggested workouts;

**optimize(O, W, P)**

the specific optimization operation(s) that the user wants to perform; each direction is assigned a weight (W) and a preference order (P).

An example of the basic input concepts described above is the following:

```logic
how_long("RUNNING", 20).
how_long("WALKING", 20).
how_long("WALKING", 10).
how_long("RUNNING", 10).
how_long("RUNNING", 20).
max_time (20).
surplus (100).
```

In this example the activities that can be performed ("ON_BICYCLE", "WALKING" and "RUNNING") are specified along with the calories they allow to burn per unit of time; then, the amount of time spent for each activity is reported. Moreover, there are pieces of information about the calories that remain to burn in the current day (at least 200, and up to 300 due to the surplus) and the maximum time that the user wants to spend on the workouts (20).

Custom optimization preferences are typically represented as follows:

```logic
optimize("RUNNING", 1, 3).
optimize("ON_BICYCLE", 3, 3).
optimize("WALKING", 2, 3).
```

Solutions, in this context, are actually workouts suggestions to the user. The optimize predicate is of arity 3, and third argument is supposed to express the “importance” of the statement (the higher the number, the more the importance). In this example, the ASP code models that: (i) the user wants (preference level: 3) to maximize the number of favourite activities to perform, and provides an order ("RUNNING" first, then "WALKING" and finally "ON_BICYCLE"); (ii) if more than one admissible workout is found featuring the same favourite activities, she wants to minimize the total time spent exercising (preference level: 2); also, (iii) if there are workouts that have the same favourite activities and the same time, she wants to minimize the total number of activities (preference level: 1).

The logic program is able to find the combinations of activities that should be performed in order to burn the remaining calories. Obviously, this goal can be achieved, in general, in many different ways, each of them modelled by a
different answer set. Part of the rules of the program that we used are reported hereafter; full program is available online.

\begin{verbatim}
% %%%%% Guess Part %%%%%%
activity_to_do(A, HL) | not_activity_to_do(A, HL) :-
how_long(A, HL).

% %%%%% Check Part %%%%%%
:- activity_to_do(A, HL1), activity_to_do(A, HL2),
HL1 != HL2.

:- remaining_calories_to_burn(RC),
total_calories_activity_to_do(CB), RC > CB.

:- remaining_calories_to_burn(RC),
total_calories_activity_to_do(CB),
CB > RC + surplus, RC + surplus = RC + surplus.

:- max_time(MTS), MTS < TS,
total_time_activity_to_do(TS).

% %%%%% Optimize Part %%%%%%
:- optimize (activities, _, P), # int (HM),
# count {A, HL : activity_to_do (A, HL )} = HM. [HM:P]


optimizing :-(activities, _, P), activity_to_do(_, HL). [HL:P]

optimizing :-(activities, _, P), # int (HM),
# count(A, HL : activity_to_do(A, HL)) = HM. [HM:P]
\end{verbatim}

The Guess Part chooses how much time to spend on each exercise. The Check Part checks that each activity selected has one specific amount of time, it ensures that all the remaining calories are burnt and that not more calories than the remaining (with the surplus) are burnt and it ensures to not exceed the maximum time that the user wants to spend on the workouts. The Optimize Part makes use of weak constraints\cite{3,4}, in case the user specified preferences about activities, tries to select the favourite ones; in case she specified preferences about the time spent exercising, tries to minimize it; if she specified preferences about the number of different activities, tries to minimize it.

There is a wide range of customization possibilities in this setting: thanks to the modeling capabilities and the declarative nature of ASP, adding new features to DLVfit, such as new exercises or new kind of preferences, is straightforward, and sums up to adding a few lines to the logic program. It is also worth noting that the ASP program is dynamically built, thus providing the developer (and, in turn, the final user) with great customization and flexibility capabilities. Indeed, we plan to actually take advantage from this in the future versions of the prototype, contemplating a higher number of rules and sub-programs to be dynamically fed to DLV.

6. RELATED WORK

The problem of embedding ASP reasoning modules into external systems and/or externally controlling an ASP system has been already investigated in the literature; to our knowledge, the more widespread solutions are the DLV Java Wrapper \cite{28}, JDLV \cite{12}, Tweety \cite{29}, and the scripting facilities featured by clingo\cite{15}, which allow, to different extents, the interaction and the control of ASP solvers from external applications.

In clingo4, the scripting languages lua and python enable a form of control over the computational tasks of the embedded solver clingo, with the main purpose of supporting also dynamic and incremental reasoning; on the other hand, EMBASP, similarly to the Java Wrapper and JDLV, acts like a versatile “wrapper” wherewith the developers can interact with the solver. However, differently from the Java Wrapper, EMBASP features a Mapper that, in the Java implementation, makes use of annotations, a form of metadata that can be examined at runtime, thus allowing an easy mapping of input/output to Java Objects; and differently from JDLV, that uses JPA annotations for defining how Java classes map to relations similarly to ORM frameworks, EMBASP straightforwardly uses custom annotations, almost effortless to define, to deal with the mapping.

Moreover, our framework is not specifically bound to a single or specific solver; rather, it can be easily extended for dealing with different solvers; in addition, it allows to build applications that can run different solvers, and different instances, at the same time; none of the mentioned systems exposes this feature. Finally, to our knowledge, the specialization of EMBASP for DLV on Android has been the first actual attempt to port ASP solvers to mobile systems reported in literature; indeed, the preliminary version of EMBASP was originally explicitly tailored to the mobile scenario \cite{5}.

Tweety is an open source framework for experimenting on logical aspects of artificial intelligence; it consists of a set of Java libraries that allow to make use of several knowledge representation systems supporting different logic formalisms, ranging from classical logics, over logic programming and computational models for argumentation, to probabilistic modelling approaches, including ASP. Tweety and EMBASP cover a wide range of applications, and the use is very similar: at the bottom line, both provide libraries to incorporate proper calls to external declarative systems from within “traditional” applications. Currently, Tweety implementation is already very rich, covering a wide range of KR formalisms, yet looking less general, as the more abstract level is conceived as a coherent structure of Java libraries; also, it currently misses the mobile focus. EMBASP is mainly focused on fostering the use of ASP in the widest range of contexts, as evidenced by the specialization for the mobile setting; nevertheless, the framework core is very abstract, and has been conceived in order to create libraries for different programming languages, platforms and formalisms.

7. CONCLUSIONS

In this paper we presented a framework for embedding ASP reasoning and modeling capabilities into external systems. The fully abstract architecture makes the framework general enough to be adapted to a wide range of scenarios; indeed, it can be implemented in any programming language, grounded to different platforms, and can make use of different ASP solvers. We herein presented an actual Java implementation and two specialized libraries for embedding DLV on Android applications and clingo on any Java-based desktop application. The framework has been tested within some university courses featuring ASP topics, for implementing a set of applications, ranging from AI-based games to educative apps; it proved to be an effective set of tools and interoperability mechanisms able to ease the development of ASP-based applications, in both educational and real-world contexts.

It is worth noting that, although the framework has been mainly conceived for fostering the usage of ASP, its abstract
core makes it also adaptable to other declarative knowledge representation formalisms.

The framework, documentation, an application showcase and further details are freely available online [6].

8. ACKNOWLEDGEMENTS

Francesco Calimeri has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No. 690974 for the project “MIREL: Mining and REasoning with Legal texts”.

9. REFERENCES


