Extending ASPIDE with User-defined Plugins

Onofrio Febbraro[1], Nicola Leone[2], Kristian Reale[2], and Francesco Ricca[2]

[1] DLVSystem s.r.l. - P.zza Vermicelli, Polo Tecnologico, 87036 Rende, Italy
febbraro@dlvsystem.com
[2]Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{leone,reale,ricca}@mat.unical.it

Abstract. ASPIDE is the most comprehensive IDE for Answer Set Programming. We describe how to extend ASPIDE with user-defined plugins by means of three examples for: (i) handling the ASP RuleML input format, (ii) performing disjunctive program shifting, and (iii) generating custom XML output.

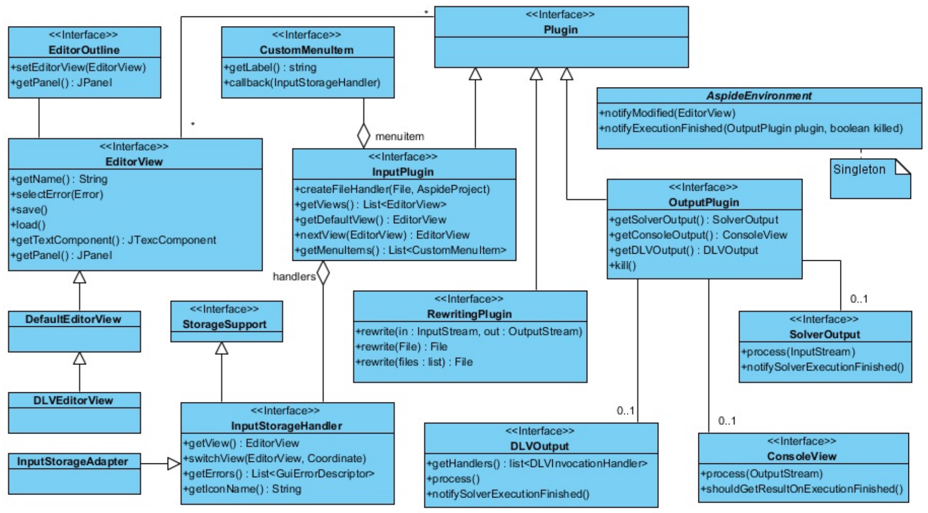
1 Introduction

Answer Set Programming (ASP) [1] is a declarative programming paradigm which has been proposed in the area of non-monotonic reasoning and logic programming. A computational problem is represented in ASP by a logic program (set of rules) whose answer sets (also called stable models [1]) correspond to problem's solutions, which can be, thus, effectively computed by an ASP solver [2]. The availability of some efficient ASP systems made ASP a powerful tool for developing advanced applications, and, also, the exploitation of ASP in industry has started [3].

As for the most diffused programming languages, the availability of a complete IDE can simplify significantly both programming and maintenance tasks. In order to facilitate the design of ASP applications, some tools for ASP-program development were proposed and the first Integrated Development Environments (IDEs) were introduced [4–6]. Among them, one of the most comprehensive is *ASPIDE* [6]. *ASPIDE* includes a cutting-edge *editing tool* with a collection of user-friendly *graphical tools* for program composition, debugging, testing, profiling, DBMS access, solver execution configuration and output-handling. Nonetheless, we describe in this paper a further significant extension of *ASPIDE* devised with the goal of improving the support to application development: user-defined plugins.

In real-world applications, input data is usually not encoded in ASP, and the results of a reasoning task specified by an ASP program is expected to be saved in an application-specific format. In addition, during the development of an ASP program, the developer might have the need to apply "refactoring", which often means "rewriting some rule" (e.g., by applying magic sets, disjunctive rule shifting, etc.), for optimizing performance, for compliance with solver formats or for modeling purposes. Having this in mind, we have introduced in *ASPIDE* the possibility to extend it with user-defined plugins. Developers can create libraries for extending *ASPIDE* with: (i) new input formats, (ii) program rewritings, and even (iii) customizing the format of solver results.

In the remainder of this paper we present the SDK, composed by Java interfaces and classes, for extending ASPIDE by describing three examples plugins for: (i) handling



<<Interface>> Plugin
<<Interface>> EditorOutline
+setEditorView(EditorView)
+getPanel() : JPanel
<<Interface>> CustomMenuItem
+getLabel() : string
+callback(InputStorageHandler)
menuitem
<<Interface>> EditorView
+getName() : String
+selectError(Error)
+save()
+load()
+getTextComponent() : JTexcComponent
+getPanel() : JPanel
<<Interface>> InputPlugin
+createFileHandler(File, AspideProject)
+getViews() : List<EditorView>
+getDefaultView() : EditorView
+nextView(EditorView) : EditorView
+getMenuItems() : List<CustomMenuItem>
AspideEnvironment
+notifyModified(EditorView)
+notifyExecutionFinished(OutputPlugin plugin, boolean killed)
Singleton
<<Interface>> OutputPlugin
+getSolverOutput() : SolverOutput
+getConsoleOutput() : ConsoleView
+getDLVOutput() : DLVOutput
+kill()
handlers
DefaultEditorView
<<Interface>> StorageSupport
<<Interface>> RewritingPlugin
+rewrite(in : InputStream, out : OutputStream)
+rewrite(File) : File
+rewrite(files : list) : File
0..1
<<Interface>> SolverOutput
+process(InputStream)
+notifySolverExecutionFinished()
DLVEditorView
<<Interface>> InputStorageHandler
+getView() : EditorView
+switchView(EditorView, Coordinate)
+getErrors() : List<GuiErrorDescriptor>
+getIconName() : String
InputStorageAdapter
0..1
<<Interface>> DLVOutput
+getHandlers() : list<DLVInvocationHandler>
+process()
+notifySolverExecutionFinished()
0..1
<<Interface>> ConsoleView
+process(OutputStream)
+shouldGetResultOnExecutionFinished()

Fig. 1. The Java class diagram for plugin development

the ASP RuleML input format, (ii) performing disjunctive program shifting, and (iii) generating custom XML output.

2 An Input Plugin for ASP RuleML

We design a new input plugin for handling and loading files containing program written in the *ASP RuleML* syntax [7]. The goal is to enrich *ASPIDE* with tools to open and edit *ASP RuleML* programs in two modalities: (i) in the original format, so that an editor, working as XML editor, shows the content of the file; (ii) in its ASP version, so that an editor, working as ASP editor, shows the program using the usual ASP syntax. When this plugin is installed in *ASPIDE*, a user can identify easily, on the *workspace explorer* panel, *ASP RuleML* files by the corresponding icon specified in the plugin.

Typical input-plugin usage scenario. The user opens one of the available *ASP RuleML* files with a simple editor of the plugin and *ASPIDE* shows the pure content of the file. When the user switches to the ASP editor of the plugin, the current program is translated in the ASP version and shown to the user. In this way the user has the possibility to edit the program in the DLV syntax editor. Finally, the user saves the program and, in this case, the ASP version of the program is translated again to the *ASP RuleML* syntax and stored in the original format of the source file.

Creating the plugin. To define this plugin we develop a new class *RuleMLPlugin* that implements the interface *InputPlugin* (fig. 1). We specify also a new class *RuleMLFile* that implements *InputStorageHandler* and represents our *ASP RuleML* input file. To define the editors mentioned on the scenario, we create two classes named *SimpleEditorXML* and *SimpleEditorASP* that extend *DefaultEditorView* and *DLVEditorView* (both implementing the interface *EditorView*) respectively. In *RuleMLPlugin*, we implement the functions *getViews*, returning the editors, and *nextView*, returning the next editor to be loaded when the user switches to the next editor. When the user opens or saves the content of an editor view, the *load* and *save* methods of *EditorView* are called. Consequently, we allow the *save* (*load*) method of *SimpleEditorXML* to store (load) the pure content of the editor and the ones of *SimpleEditorASP* to store (load) the content of the editor translated from ASP to *ASP RuleML* format (and vice-versa); for this last purpose we create a class *RuleMLPlugin* that implements *RewritingPlugin* (fig. 1) with a rewriting procedure (see next section for rewriting plugin creation).

Usage in ASPIDE. We now show the usage of the *ASP RuleML* plugin in *ASPIDE*. The user creates a new project that includes an XML file containing the well known *MaximalClique* encoding in the *ASP RuleML* format. The user chooses to open it by selecting *Simple RuleML Editor*. *ASPIDE* shows the editor with the content of the file. To switch to the ASP editor for showing the program in the DLV format the user clicks the switch button. *ASPIDE* opens the *Simple ASP Editor* showing the translated version of the program (in ASP). In this editor the user modifies the program by adding facts, and switches again (after saving) to the *ASP RuleML Editor*. The editor will show the modified version of the file that includes also the facts that the user has added.

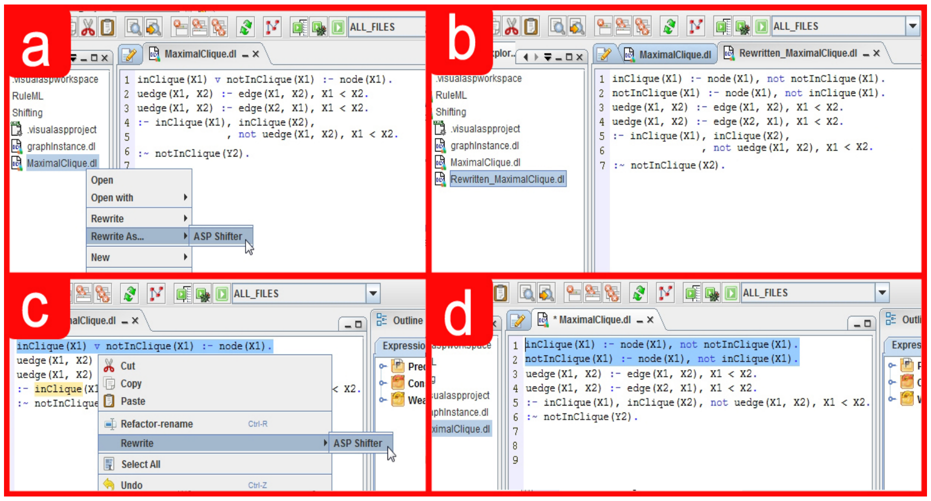
3 A Rewriting Plugin for Shifting ASP Rules

In this section we describe the creation of a *rewriting plugin* for enriching *ASPIDE* with a new functionality that allows one to rewrite disjunctive rules in order to obtain a "shifted" version, where disjunction is replaced by cyclic non-monotonic negation (this rewriting produces an equivalent encoding in case of Head-Cycle Free programs). For example, rule $a \vee b$ is rewritten in $a \coloneq not\ b$ and $b \coloneq not\ a$, the intuition here is that "disjunction is shifted in the body".

Typical rewriting-plugin usage scenario. The user can apply a rewriting procedure to ASP files or to a part of them by selecting a DLV file on the workspace explorer panel, and using the shifting function of the *rewriting plugin*. In this case, the whole file is analyzed and rewritten in a shifted version. This operation can also be directly used on a part of the program by selecting interested rules in the DLV editor; in this way only the selected rules are translated to the shifted version.

Creating the plugin. We specify a new class *ShifterPlugin* that implements the interface *RewritingPlugin* (fig. 1). The programmer must override three methods used for dealing with an entire file, with multiple files and with ASP code usually corresponding to some rules selected in the editor. The results of the rewriting procedure will be stored to a file or written to the *OutputStream* parameter.

Usage in ASPIDE. We now show the usage of the *ASP Shifter* plugin in *ASPIDE*. After its installation, new menu items appear automatically in *ASPIDE*, allowing one to use the rewriting functionality on DLV files. A first way of using the *ASP Shifter*



```
inClique(X1) v notInClique(X1) :- node(X1).
uedge(X1, X2) :- edge(X1, X2), X1 < X2.
uedge(X1, X2) :- edge(X2, X1), X1 < X2.
:- inClique(X1), inClique(X2),
                , not uedge(X1, X2), X1 < X2.
:~ notInClique(Y2).
```

```
inClique(X1) :- node(X1), not notInClique(X1).
notInClique(X1) :- node(X1), not inClique(X1).
uedge(X1, X2) :- edge(X1, X2), X1 < X2.
uedge(X1, X2) :- edge(X2, X1), X1 < X2.
:- inClique(X1), inClique(X2),
                , not uedge(X1, X2), X1 < X2.
:~ notInClique(X2).
```

```
inClique(X1) :- node(X1), not notInClique(X1).
notInClique(X1) :- node(X1), not inClique(X1).
uedge(X1, X2) :- edge(X1, X2), X1 < X2.
uedge(X1, X2) :- edge(X2, X1), X1 < X2.
:- inClique(X1), inClique(X2), not uedge(X1, X2), X1 < X2.
:~ notInClique(Y2).
```

Fig. 2. Shifter plugin at work.

plugin consists in exploiting the popup menu on a DLV file (in this example *MaximalClique.dl*); the user selects the menu item *Rewrite As, ASP shifter* (fig. 2a). The result of this action is an automatic creation of a new DLV file (fig. 2b). The *ASP Shifter* plugin can be directly used on a DLV editor, in this case selecting a set of rules and using the command *Rewrite, ASP Shifter* (fig. 2c); selected code is rewritten and replaced with the corresponding shifted code (fig. 2d).

4 An Output Plugin for Formatting Solver Output in XML

We now describe the creation process of an output plugin for handling the solver output, generated on the execution process, and writing it in a new format. The output can be used for many purposes like: (i) redirecting it to some other solver, (ii) translating it to some other output format to be shown, for example, in the output console or in a new user-defined window.

Typical output-plugin usage scenario. The user opens the Run Configuration dialog by adding program files that resolve the *Maximal Clique* problem. Exploiting the window, the user chooses to visualize the results using the plugin, so that, after the execution, the Console Window is open with answer sets formatted to the custom XML format.

Creating the plugin. We start by designing the class *CustomXMLOutput* that implements the interface *OutputPlugin* (fig. 1). This class allows one to: (i) capture the answer sets of the DLV solver, (ii) rewrite the answer sets in the XML format, (iii) print the result to the Console Window, (iv) notify *ASPIDE* that the execution is finished. The DLV Wrapper library [8] offers Java classes that represents models and atoms, so we can exploit it for handling the DLV output by means of Java objects without any need to create specific parsers. For the writing procedure to the console we create a new class that implements *ConsoleView* (fig. 1) and we override the method *process(OutputStream)* making it to write the custom XML output to the *OutputStream*; *ASPIDE* redirects the *OutputStream* to the Console Window on the execution phase. When the execution of the solver and the rewriting procedure are finished, the method *notifyExecutionFinished* to the singleton class *AspideEnvironment* must be called so that *ASPIDE* can show the result window.

Usage in ASPIDE. In *ASPIDE* the user opens the Run Configuration window by the *Execute* menu. After the setting of the solver and the specification of the *Maximal Clique* encoding, the user chooses to show the results in the *Custom ASP-XML Output* format and click on the Run button. The solver is executed, the output of the solver is passed to the plugin (activating the rewriting procedure) and the result is shown to the console.

5 Conclusion and Future Work

Since its first presentation at LPNMR'11 in Vancouver, *ASPIDE* has improved in several respects. In this paper we have described one of the new remarkable additions of *ASPIDE*: the support for user-defined plugins. This new feature allows one to write an entire ASP-based application with minimal (or no) need for external conversion tools.

The Java interfaces herein described belong to *aspidePluginKit.jar*, a JAR library that provides an SDK distributed under the LGPL license and available at the *ASPIDE* web site `https://www.mat.unical.it/ricca/aspide/`.

Acknowledgments. This work has been partially supported by the Calabrian Region under PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) project DLVSYSTEM approved in BURC n. 20 parte III del 15/05/2009 - DR n. 7373 del 06/05/2009.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Lifschitz, V.: Answer Set Planning. In: ICLP'99) 23–37
3. Grasso, G., Leone, N., Manna, M., Ricca, F.: ASP at Work: Spin-off and Applications of the DLV System Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of M. Gelfond. LNCS 6565 (2011)
4. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog* Environment. In: SEA 07 101–115
5. Oetsch, J., Pührer, J., Tompits, H.: The sealion has landed: An ide for answer-set programming—preliminary report. In: INAP2011/WLP2011. Volume abs/1109.3989. (2011)
6. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated Development Environment for Answer Set Programming. In: LPNMR 2011. LNCS 6645, (2011) 317–330
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: H.: A ruleml syntax for answer-set programming. In: Informal Proceedings of the Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS06). (2006) 107108
8. Ricca, F.: The DLV Java Wrapper. In: ASP'03, Messina, Italy (2003) 305–316 Online at `http://CEUR-WS.org/Vol-78/`.