Università della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

xxv ciclo

Settore Disciplinare INF/01 – INFORMATICA

Tesi di Dottorato

ASPIDE: Integrated Development Environment for Answer Set Programming

Kristian Reale

Supervisori
Prof. Francesco Ricca

Coordinatore
Prof. Nicola Leone

A.A. 2011 – 2012

Università della Calabria

Dipartimento di Matematica

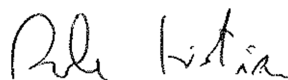
Dottorato di Ricerca in Matematica ed Informatica

XXV CICLO

Settore Disciplinare INF/01 – INFORMATICA

Tesi di Dottorato

ASPIDE:
Integrated Development Environment for Answer Set Programming

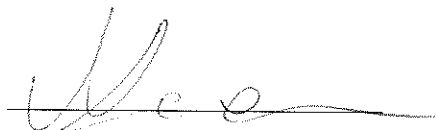


Kristian Reale

Supervisore

Prof. Francesco Ricca

Coordinatore

Prof. Nicola Leone 

A.A. 2011 – 2012

Acknowledgments

I would like to express my sincere gratitude to my supervisor Francesco Ricca, who has encouraged me to do my work during these years with patient guidance and constant support. Through his energy and enthusiasm in research and in my work, he gave me all the inspiration and motivation to achieve the results presented in this thesis in a care way and supporting me on the development phase of ASPIDE.

I am deeply grateful to Nicola Leone particularly because he has supported me in these years and for all the opportunities for personal and professional growth he ensured me.

I would like to thank also the research group of the Department of Mathematics at Unical, which has been a source of friendships as well as good advice and collaboration. Among them, a sincere and grateful thanks goes to the colleagues who shared the office with me, especially Pierfrancesco Veltri and Onofrio Febbraro for their friendship and constant help.

Particular thanks to Onofrio Febbraro because his collaboration for the development of ASPIDE has been crucial and many implementation solutions have been possible thanks to his professional skills.

I wish to thank also all the long-standing friends for always keeping in contact and do not let the miles set us apart. Their friendship has helped me to get through tough periods. Lastly, and most importantly, I owe my deepest gratitude to my mother, my father and my brother for their care, moral support, and love.

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.

Abstract

Answer Set Programming (ASP) is a truly-declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming. The successful application of ASP in a number of advanced projects, has renewed the interest in ASP-based systems for developing real-world applications. Nonetheless, to boost the adoption of ASP-based technologies in the scientific community and especially in industry, it is important to provide effective programming tools, supporting the activities of researchers and implementors, and simplifying user interactions with ASP solvers. In the last few years, several tools for ASP-program development have been proposed, including (more or less advanced) editors and debuggers. However, ASP still lacks an Integrated Development Environment (IDE) supporting the entire life-cycle of ASP development, from (assisted) programs editing to application deployment.

In this thesis we present *ASPIDE*, a comprehensive IDE for ASP. It integrates a cutting-edge editing tool (featuring dynamic syntax highlighting, online syntax correction, auto-completion, code-templates, quick-fixes, refactoring, etc.) with a collection of user-friendly graphical tools for program composition, debugging, profiling, database access, solver execution configuration and output-handling.

A comprehensive feature-wise comparison with existing environments for developing logic programs is also reported in this thesis, which shows that *ASPIDE* is a step forward in the present state of the art of tools for ASP programs development.

Sommario

L'Answer Set Programming (ASP) è un paradigma di programmazione totalmente dichiarativo proposto nell'area del ragionamento non-monotono e della programmazione logica. Il successo dell'applicazione di ASP in un vasto numero di progetti avanzati, ha rinnovato l'interesse di sistemi basati su di esso per lo sviluppo di applicazioni nel mondo reale. Ciononostante, per incentivare l'adozione di tecnologie basate su ASP nella comunità scientifica e specialmente in ambito industriale, è importante fornire efficaci strumenti di sviluppo che supportino le attività di ricercatori e sviluppatori e che semplifichino l'interazione degli utenti con i solver ASP.

Negli ultimi anni sono stati proposti diversi tool per lo sviluppo di programmi ASP, compresi editor e debugger (più o meno avanzati). Tuttavia, ASP è carente di un ambiente di sviluppo integrato (IDE) per lo sviluppo di programmi ASP nel loro intero ciclo di vita, dalla scrittura (assistita) di programmi, al deployment delle applicazioni.

In questo lavoro di tesi presentiamo il sistema *ASPIDE*, un ambiente di sviluppo completo per ASP. *ASPIDE* integra un innovativo strumento di editing ed offre caratteristiche come la sintassi colorata, l'evidenziazione automatica degli errori di sintassi, l'auto-completamento, i code template, i quick-fix, il refactoring, ecc. Inotre, *ASPIDE* include una collezione di strumenti grafici ed intuitivi per la composizione di programmi, il debugging, il profiling, l'accesso ai database, la configurazione dell'esecuzione di un solver e la gestione dei risultati.

La tesi riporta anche un confronto dettagliato delle caratteristiche del sistema con quelle degli ambienti esistenti per lo sviluppo di programmi logici, mostrando che *ASPIDE* rappresenta un importante passo avanti nel presente stato dell'arte per gli strumenti di sviluppo di programmi ASP.

Contents

1 Introduction **1**

2 Integrated Development Environments **5**

2.1 Overview 5
2.2 A broad classification of IDEs 6
2.2.1 Language-centered environments 7
2.2.2 Structure-oriented environments 7
2.2.3 Toolkit environments 8
2.2.4 Method-based environments 8
2.3 Microsoft Visual Studio 8
2.3.1 Main Features 9
2.4 Eclipse 10
2.4.1 Main Features 11
2.5 Common features of IDEs 12

3 Answer Set Programming **15**

3.1 ASP Language 15
3.1.1 Syntax 15
3.1.2 Semantics 17
3.2 Aggregates and Language Extension 19
3.2.1 Aggregate Functions 19
3.2.2 Weak contraints 20
3.2.3 Language extensions for Database Management 20
3.3 Dependency Graphs 21
3.4 Relevant Sub-Classes 22
3.4.1 Stratified Programs 22
3.4.2 Head-Cycle Free Programs 23
3.5 Modularity aspects 23
3.5.1 Splitting an ASP program 24
3.5.2 DLP-functions 24
3.6 Knowledge Representation 26
3.6.1 Reachability 27
3.6.2 Hamiltonian Path 28
3.6.3 Maximal Clique 29
3.6.4 Maze Generation 29

4 ASPIDE **33**
4.1 *ASPIDE* Graphical Interface Overview 33
4.2 System Features 34
4.2.1 Workspace organization 35
4.2.2 Advanced text editor 38
4.2.3 Automatic completion 38
4.2.4 Code template 41
4.2.5 Annotation management for ASP programs 49
4.2.6 Outline navigation 50
4.2.7 Errors and Warnings management 53
4.2.8 Dynamic code checking and errors highlighting 54
4.2.9 Quick fix 55
4.2.10 Dependency graph 59
4.2.11 Configuration of the execution 59
4.2.12 Presentation of the results 63
4.2.13 Debugger and Profiler 67
4.3 System Architecture and Implementation 72

5 Visual Editor for drawing logic programs **75**
5.1 Visual Editor overview 76
5.2 The Visual Editor by a Use Case Example 77
5.2.1 Drawing Rule r_1 79
5.2.2 Drawing Rules r_2 and r_3 81
5.2.3 Drawing Constraint r_4 83
5.2.4 Drawing Constraint r_5 and r_6 86
5.2.5 Switching from Visual Mode to Textual Mode 88
5.2.6 Specifying DLV Directives and comments 88
5.2.7 Other features of the *Visual Editor* 90

6 Unit Testing in ASPIDE **93**
6.1 Context and Motivation 93
6.2 Contribution 94
6.3 Unit Testing in ASP 95
6.3.1 Testing Language 96
6.3.2 Test case example 98
6.3.3 Modularity aspects 100
6.3.4 Testing methodology 101
6.4 Implementation in *ASPIDE* 102
6.4.1 Unit testing in *ASPIDE* 102
6.4.2 Visual Editor for building Test Suites 105

7 Extending ASPIDE with user-defined Plugins **109**
7.1 Motivation and Contribution 109
7.2 Input Plugin 110
7.2.1 An Input Plugin for ASP RuleML 110
7.3 Rewriting Plugin 114
7.3.1 A Rewriting Plugin for Shifting ASP Rules 114
7.4 Output Plugin 116
7.4.1 An Output Plugin for a Custom XML Output 117
7.5 The SDK Library for Plugins Development 119

7.5.1 Input Plugins classes description 119
7.5.2 Rewriting Plugins classes description 123
7.5.3 Output plugins classes description 127
7.5.4 The *AspideEnvironment* Java interface 128
7.5.5 Implementation, Deploy and Installation of *ASPIDE* plugins 128

8 Database Management in *ASPIDE* 131
8.1 Schema Management and Table Mappings 132
8.1.1 TYP Files 132
8.1.2 Import/Export Directives 135
8.1.3 Schema Annotations 136
8.2 Database Interaction Plugin 138
8.3 Use Case: A Data Integration Scenario 139

9 Related Work 143
9.1 IDEs for Declarative and Logic Programming Languages 143
9.1.1 Logic and Database-based environments 144
9.1.2 Ontology-based environments 145
9.1.3 Prolog-based envoronments 146
9.1.4 Answer Set Programming based environments 148
9.2 Comparison with *ASPIDE* 150

10 Conclusion 155

Bibliography 157

List of Tables

9.1 System Comparison for General Features. 151
9.2 System Comparison for the Project Management Features. 152
9.3 System Comparison for the Text Editor Features. 152
9.4 System Comparison for the Visual Editor Features. 153
9.5 System Comparison for Supported Languages and Solvers. 154

List of Figures

2.1 Taxonomy on Integrated Development Environments. 6
2.2 *Visual Studio* IDE overview for C#. 9
2.3 Eclipse for Java IDE overview. 11

3.1 *Graphs (a)* $DG(\mathcal{P}_4)$, *and (b)* $DG(\mathcal{P}_5)$ 21
3.2 *Graphs (a)* $DG^{+}(\mathcal{P}_5)$, *and (b)* $SCC(DG^{+}(\mathcal{P}_5))$ 22

4.1 The *ASPIDE* graphical user interface. 33
4.2 Project and Workspace Explorer panels. 36
4.3 Select the workspace. 37
4.4 Create a new project. 37
4.5 Create a new DLV File. 38
4.6 On-line auto-completion of a disjunction. 39
4.7 On-line auto-completion of an atom. 39
4.8 On-line auto-completion of a variable. 40
4.9 Atom auto-completion using a new predicate. 41
4.10 On-request auto-completion of a variable. 41
4.11 Auto-completion using the keywork DIS for writing a disjunction. 42
4.12 Code template using the keyword *DIS3inPath2* for building a disjunction. 42
4.13 Code template using the keyword *guess* for writing a disjunction that guesses paths. 43
4.14 Code template using the keyword *guessStrict* for writing a disjunction containing a normal atom and its true negated version. . 44
4.15 Code template for writing the aggregate `#count`. 44
4.16 Code template for writing a key constraint. 45
4.17 Code template for writing an inclusion constraint. 46
4.18 Code template for writing an *AtMost* constraint. 46
4.19 Code template for defining a transitive closure. 47
4.20 Template creation and definition. 48
4.21 Specify the input predicates of the template `path`. 48
4.22 Create a new template. 49
4.23 Outline for a *TYP file* on the left and for a *TEST file* on the right. 51
4.24 Different views for the outline representing a program. 51
4.25 Outline for the Maximal Clique program and accessing to a line of code. 53
4.26 Select the rule with the safety error. 54
4.27 Error and Warning highlighting. 55

4.28 Applying a quick fix for a safety error. 55
4.29 Complete, Positive and Connected Components Dependency Graphs of the Hamiltonian Path program. 59
4.30 Run Configuration Dialog. 60
4.31 Run Button. 61
4.32 Quick Run of files. 61
4.33 Workflow execution Editor. 62
4.34 Tabular results for the *Hamiltonian Path* program. 63
4.35 *Hamiltonian Path* program solver output printed to the Console. 64
4.36 Graphical visualization of a Maze Generation result. 64
4.37 Open the Query Window. 65
4.38 The Query Window. 65
4.39 Debug the current program. 68
4.40 Applicable and blocked rules on the debugging process. 68
4.41 Check rule abnormalities for rules. 69
4.42 Check supporting atoms abnormalities. 70
4.43 Check unfounded atoms abnormalities. 70
4.44 Run the *DLV Profiler* for the current Run Configuration. 71
4.45 *DLV Profiler* GUI in *ASPIDE*. 71
4.46 System architecture of *ASPIDE*. 72

5.1 Overview of the *Visual Editor*. 76
5.2 Creating a new Predicate. 77
5.3 The predicate `node` in the *Outline* and the attribute *NodeLabel* inserted in `node`. 78
5.4 Insert facts in `edge`. 79
5.5 Dialog window for setting the head of the new disjunctive rule. . 79
5.6 Dragging of the predicate `edge` from the *Outline* to the *Body Graph*. 80
5.7 Projecting the attribute *Target* of the predicate `edge`. 80
5.8 Join the predicates `inPath` e `reached`. 81
5.9 Join the attribute *Source* of the predicate `inPath` and *Node* of the predicate `reached`. 82
5.10 Details of the join. 82
5.11 Applying a quick fix for a Safety Error. 83
5.12 Specifying a name to the constraint. 84
5.13 Created a new constraint and made a join between the predicates `node` and `reached` inserted in the body. 84
5.14 Negation of the predicate reached. 85
5.15 Creating a quick negation for the predicate `start`. 85
5.16 Defining an inequality for attributes. 86
5.17 Aggregation of the predicate *inPath*. 87
5.18 Result of the aggregation of the predicate `inPath`. 87
5.19 Result of the aggregation of the predicate `inPath`. 88
5.20 Specifying an *Import* Directive. 89
5.21 Specifying a comment to the *Visual Editor*. 89
5.22 A rule with built-in literals. 90
5.23 Collapsing two body literals. 91

6.1 Input graphs of the Maximal Clique program. 98
6.2 Test case creation. 103

6.3 Test case execution and assertion management. 104
6.4 Visual Editor for *TEST File* definition. 106
6.5 Creating a test case by exploiting the Visual Editor. 107

7.1 Input Plugin interfaces Diagram. 111
7.2 *ASP RuleML* plugin at work. 113
7.3 Rewriting Plugin interfaces Diagram. 115
7.4 *Shifter Plugin* at work. 116
7.5 Output Plugin interfaces Diagram. 117
7.6 *Custom XML* plugin at work. 119
7.7 Opening of an *ASP RuleML* file using the DLV Editor. 126
7.8 *ASPIDE* Script Executor plugin at work. 126
7.9 *AspideEnvironment* Java interface. 128
7.10 Installing a new plugin in *ASPIDE*. 130
7.11 Installing a new plugin in *ASPIDE* by checking available plugins on the web. 130

8.1 Text Editor and outline of a *TYP file*. 134
8.2 Visual Editor of a *TYP file*. 134
8.3 Schema visualization in the *Outline* panel. 137
8.4 Database Plugin Architecture. 138
8.5 Creating a new source. 140
8.6 Mapping of database tables. 140
8.7 Switch among mapping rewritings. 141
8.8 Use mapped tables. 141
8.9 Execution of the global schema. 142
8.10 Results of the global schema execution. 142

Chapter 1

Introduction

Context and Motivation. Answer Set Programming (ASP) [51] is a truly-declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming. The language of ASP is very expressive [30] and the idea consists in representing a given computational problem by a logic program whose answer sets correspond to solutions and then using a solver to find such solutions [66]. ASP applications belong to several fields, from Artificial Intelligence [47, 4, 7, 8, 44, 43, 71] to Information Integration [62], and Knowledge Management [5, 9, 52]. These applications of ASP have confirmed the viability of the exploitation in real application settings and, very recently, stimulated some interest also in industry [53]. Furthermore, the availability of some efficient ASP systems [63, 82, 83, 94, 68, 3, 46, 58, 65, 28, 47, 26] make ASP a powerful tool for developing advanced applications.

The mentioned ASP applications, however, have evidenced the lack of "complete" and "effective" development environments capable of supporting the programmers in managing large and complex projects [27]. Nowadays it is recognized [27] that this may discourage the usage of the ASP programming paradigm, even if it could provide the needed reasoning capabilities at a lower (implementation) price than traditional imperative languages. Note also that, the most diffused programming languages always come with the support of Integrated Development Environments (IDEs) featuring a rich set of tools that significantly simplify both programming and maintenance tasks. The general goal of an IDE is to help programmers in easy writing programs in an assisted way and, mostly, to support the entire life-cycle of software development from (assisted) programs editing to application deployment. Many IDEs exploit tools for graphical composition of entire programs or portions of code, for example visual editors for drawing window forms available in some IDEs for imperative programming languages.

In order to facilitate the design of ASP applications, some tools for ASP-program development have recently been proposed that range from specialized editors [54, 74, 85, 75, 73, 80] to debuggers [15, 13, 33, 73, 14]. The task of designing the programs consists, currently, of writing text files (more or less computer-assisted) and this task might be uncomfortable for novice users; moreover, programmers often have to know the details of a specific ASP input dialect. This, together with the intrinsic differences between commonly-employed imperative languages and a purely declarative one, makes writing ASP programs an

activity for ASP-experts (or, even worse, for experts in a specific ASP-system). Faced with a similar problem in the field of databases, for facilitating the specification of queries, researchers proposed a number of tools and graphical user interfaces [93, 77, 76, 81] starting from the 70s. Moreover, people from the logic programming community, and especially Prolog programmers are already exploiting tools for assisted program development. Some support to the development of logic programs is also present in environments conceived for logic-based ontology languages, which, besides graphical ontology development, also allow for writing logic programs (e.g., to reason on top of the knowledge base). However, these tools cannot be completely ported to ASP because of its different approach.

This thesis deals with the mentioned issues by the implementation of an IDE for ASP which embraces a wide number of features already implemented in the most diffused IDEs for imperative languages. The IDEs, in fact, offer a wide set of features helping programmers with composing, editing, fixing errors in a guided way, executing, testing, debugging and more and more.

Contribution. The work presented in this thesis provides a contribution in this setting, through the realization of *ASPIDE*, a comprehensive and advanced IDE for Answer Set Programming. The system *ASPIDE* is a result of several analyses and considerations on existing IDEs for imperative programming languages. Nonetheless, the approach for writing ASP programs is different from imperative languages because of the declarative nature of ASP and, consequently, features which are already and commonly exploited by other IDEs have been re-designed to deal with ASP programs.

The following are the main features of *ASPIDE* that we have developed and are described in this work:

1. **Kernel Editing Features**: they consist of features which are basically provided by almost all IDEs. The features, offering a minimal support for writing programs, are listed below:
    - *Projects organization*: organizing several files in projects is useful in the case where a program should be organized in modules. This organization helps programmers when a software is big and needs some project organization.
    - *Text Editor*: it offers features which ease writing programs like dynamic syntax highlighting, on-line syntax correction, auto-completion, code-templates and so on.
    - *Error Detecting*: a feature which automatically detects syntax errors and allows the user to apply some quick fixes. Detected errors are signaled directly to the editor and collected to an Error Console.
2. **Visual Editor**: IDEs for imperative programming languages, as well as tools for database querying, offer visual editors useful for drawing parts of code or for graphical composition of queries. For example, Visual Studio and Eclipse offer a Visual Editor for drawing window forms. Regarding databases, many commercial and free relational database query tools offer fully graphical Query By Example (QBE) interfaces for facilitating the end approach of users to systems and languages. The practical relevance

of graphic tools is now well-recognized: a QBE interface is, indeed, the default in the user-oriented Microsoft Access. ASP still lacks also these kinds of tool, which might serve for reducing the difficulty of producing ASP programs for both novice and inexperienced programmers, and easing the encoding tasks for experts that prefer graphic tools. In this setting we have defined a visual language, inspired by QBE editors, which supports all the constructs available in ASP, and we have implemented a Visual Editor which exploits the language. In this way users can draw ASP programs on the screen in a full graphical environment.

3. **Program execution**: the following features regard program execution in its various aspects:

   - *Execution and presentation of results*: an important feature offered by IDEs for imperative programming languages is the possibility of executing programs inside the same environment. However, executing an ASP programs has a different effect because the result of the execution consists in showing, textually or visually, answer sets or query results. *ASPIDE* allows users to call external solvers by pre-configuring the execution with files to be executed and solver options. Results are visualized in the environment in different ways like textual way, graphical way and customized way. Moreover, results can be rewritten to some other formats or saved to some files for subsequent execution. For query execution, having different reasoning modes (brave and cautious reasoning) [30], results are displayed in a comfortable view and a specific output for Epistemic queries [48, 49] is available.
   - *Debugging*: every IDE for software development always come with a debugger. However, the task of debugging ASP programs needs a totally different approach compared with the ones available for imperative programming languages. Since many solutions were proposed in literature, we give a contribution by embedding the existing debugging tool *spock* [15] and providing a user-interface.
   - *Tracing* and *Profiling*: on the execution phase of a program, tracing consists in recording information about the execution for debugging or optimization purposes. Also in this setting we provide a contribution by embedding the graphical tool proposed in [20] which allows the DLV solver to be traced in the execution phase.

4. **Unit Testing**: testing programs consists in checking whether a program behaves as expected. However, the crucial task of testing ASP programs received less attention in the literature and the current proposals [56, 57, 72, 90] do not support users on programs development. In *ASPIDE*, a solution inspired by the JUnit framework for Java was implemented by generalizing previous definitions of Test Case of ASP programs and introducing the concept of "Unit" in ASP programs.

5. **DBMS access**: in our setting, database interactions allows the import of schemas and metadata, retrieval of data, definition of mappings between predicates and database tables, exploitation of the language directives of DLV^{DB} [88] and interaction with it. We offer a graphical interface that

helps to interact with databases in an intuitive way. For example, the user can easily map tables in predicates in order to import facts from the table. With these features, a data integration scenario [62] can be implemented.

6. **Extensibility via 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 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. To deal with these purposes we have implemented an SDK which allows users to introduce plugins in *ASPIDE* which allow dealing with new imput formats, performing program rewriting and even customizing the format of solver results. Note that our setting of plugins is different compared with the definition of Eclipse plugins which allow extension of the IDE with new general features. Plugins in *ASPIDE* also extend the IDE, but for specific purposes related to logic programming.

Finally, *ASPIDE* is able to load and store ASP programs in the syntax of the ASP system DLV [63], and supports the `ASPCore` language profile employed in the ASP System Competition 2011 [18].

In this thesis we also show, by a comprehensive feature-wise comparison with existing environments for developing logic programs, that *ASPIDE* is a step forward in the present state of the art of tools for ASP programs development.

Organization. The remainder of the thesis is structured as follows. Chapter 2 describes a brief introduction to Integrated Development Environments, with a description of some of the well-known ones. Chapter 3 presents an overview of Answer Set Programming. Chapter 4 describes all the main features of *ASPIDE* with use case examples and screenshots. In Chapter 5 the Visual Editor for drawing ASP programs is described, and Chapter 6 provides a detailed description of the Unit Testing methodology proposed in *ASPIDE*. Chapter 7 describes the proposed SDK for implementing and using user-defined plugins. Chapter 8 describes the use of *ASPIDE* for schema and database management. Chapter 9 shows a comprehensive feature-wise comparison with existing environments for developing declarative programs and, finally conclusions are reported.

Chapter 2

Integrated Development Environments

2.1 Overview

In Software Engineering environments refer to a collection of software tools used by programmers to build (more or less complex) software systems. The terms *Programming Environment* and *Software Development Environment* are generally used as synonym, but in reality they should be distinct. In particular the first term refers to environments suitable for building programs only, whereas the second one considers environments supporting the entire life-cycle of the software development process, consisting also of the project management phase, debugging, testing and validation, prototypes definitions, version control and so on.

An *Integrated Development Environment (IDE)* consists in a set of (integrated) tools supporting program creation, modification, execution and debugging. This is a well-known classical definition proposed in the 1980s [24] that emphasizes the importance of integrating different tools in a unique environment. The basic tool of an IDE consists of a *Text Editor* used for the program composition phase. Programming languages, in fact, are defined by a text-based grammar and allow the programmer to compose a set of text based instructions (at lower or high level) that a computer must execute for purposes like automatic solving of problems. Another important tool that an IDE should have is a *debugger*, used to help the programmer to detect the reason for wrong behavior of the program on the execution phase.

Currently, available IDEs have been expanded to include many other tools supporting, for example, testing, advanced run configurations, advanced syntax error detections on the editing phase, and version control of files. However, having different tools inside an environment is not enough in case they are not integrated with each other. Integrating the tools to enable them to share the same data structures and purposes, is crucial from the programmer's point of view because he can have total control in the entire life-cycle of software development.

In this Chapter, a first classification of Integrated Development Environments is outlined by evidencing general roles that IDEs currently have. Im-

perative programming languages come with a wide assortment of IDEs offering different tools which help programmers to design, compile and build any kind of sofware like stand alone softwares, web applications, smart phones applications and so on. Since *Microsoft Visual Studio* and *Eclipse* are the two most important and well-known IDEs for imperative programming, they are described in detail in this Chapter because the features included in them give a clear view on what users accustomed to IDEs would expect in an IDE for Answer Set Programming. Finally, a list of common features included in various IDEs is described.

2.2 A broad classification of IDEs

Integrated development environments evolved, in the history, with the introduction of new important features that are oriented to the entire life-cycle of software development. The first environments were console oriented from the user interface point of view and focussed on the program composition and execution only. Commands written on console windows, were used to exploit compiling, error checking, execution and debugging. Given that the evolution of IDEs moved forward a lot, a broad classification of them has been proposed in [23] just for describing trends that they had in history. The classification consists of four categories:

- Language-centered environments;
- Structure-oriented environments;
- Toolkit environments;
- Method-based environments.

That classification can be seen as a taxonomy depicted in Figure 2.1.

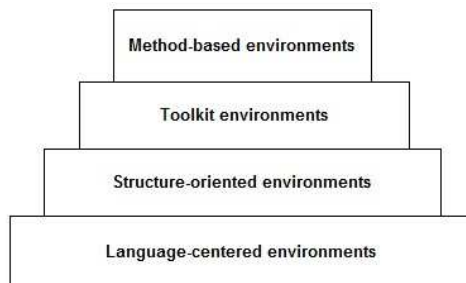


Figure 2.1: Taxonomy on Integrated Development Environments.

The base of the taxonomy is represented by *Language-centered environments* because all IDEs are generally oriented to at least one language. The top of the taxonomy represents *Method-based environments* which include the features of the other (lower) environments of the taxonomy and, moreover, support the entire life-cycle of software development process.

2.2.1 Language-centered environments

Language-centered environments are completely customized for a specific language. They propose a minimal set of tools for easy program composition by exploiting a text editor generally colored to highlight specific keywords used by the language. Most of them offer also some auto-completion features, can detect syntax errors and can offer to the user semantic information related to the language. Programs can be immediately executed in the same environment (development and runtime environments are the same) and can be halted every time a programmer needs to take debugging actions.

Those kinds of environment are very useful in the case where programmers would like to build small application fast or to implement prototypes to be presented to the final user (before the real implementation of the entire software). These environments encourage especially the incremental software development method. This method allows programmers to write program in an incremental way by writing, executing and continuing writing once the correctness of the code portion is verified through the execution. Programmers make programming decisions according to the passing or the failing of a specific execution. Some environments for the languages *Lisp*, *Smalltalk* and *ADA* are examples of language-centered environments.

2.2.2 Structure-oriented environments

A first attempt to move away from environments that are strictly language oriented is represented by structure-oriented environments. These environments consist in building a structure representation of the program. In particular, blocks of code like *if* and *for* are internally inserted in a data structures and are used to suggest to the programmer possible auto-completion and advanced editing ways to fill missing values in blocks. For example a *for* block can be filled using variables and collections that the interface suggests to the user using a dedicated panel or the editor itself. These kinds of suggestion can be activated by special key combinations or by exploiting commands offered by the user interface. The user interface of some environments offers also various ways to introduce entire portions of code fastly. For example, available templates for building common portion of codes that follow a pattern can be exploited and customized.

An inportant advantage of these environments is the possibility of building different views of the same program. Exploiting trees or dependency graphs, can be a good way to represent some programs; programmers are allowed to better read programs and edit them without any need to act on the original programming language. In this way environments are more language independent.

Structures on environment allow a sophisticated semantic checking of the code. Environments need only to verify the correctness of the data structure and to build semantic information to be shown to the user. Possible quick fix suggestions can be made in case the structure contains errors.

Structure-oriented environments have been accepted primarily as teaching aids in universities, but little acceptance has been found in industry.

2.2.3 Toolkit environments

Toolkit environments are a set of tools that support the programmer in the coding phase and in other features like compiling, linking and debugging. They are strictly based on operating systems that offer some support for installing tools, updating existing ones and using them together. Starting from tools for simple editing and compiling, new tools can be easily installed to support other advanced features like file versioning and specialized editors. However, these environments offer less support for integration of tools; consequently, the task of integrating them to work together in the correct way is up to the programmer who, in general, has to control information exchange between the tools.

The scenario just described offers extensibility and also portability between other environments because of the using of simple and uniform data modeling that helps the communication between different kinds of tool. Other tools that support the same data model can easily be introduced and, consequently, the extensibility is made easier.

The environments just described, however, do not greatly assist the maintenance of large software systems because of the independence between tools and because they can frequently change.

2.2.4 Method-based environments

A challenge to the software development process consists in controlling the entire software development cycle by exploiting graphical tools. In software engineering, building a software is an activity consisting of different phases: requirements analysis, design, validation and verification, and reuse. The purpose of method-based environments consists in supporting a team of developers in those single phases. For every phase, different languages and (also graphical) representation formalisms (informal, semi-formal and formal) were proposed. These formalisms ease communication with end-users, in the requirements analysis phase, and boost team collaboration in the software designing phase. An example of formalisms is UML (Unified Modeling Language), exploited for the requirements analysis and designing phases, and E-R schemas, exploited for the designing of relational databases.

The set of tools that aim to support developers in the mentioned phases, define well-known *Computer-Aided Software Engineering* (CASE) tools. By exploiting these tools, team of developers can easily design, for example, software architectures using UML. Difficulties for CASE tools consist in integrating them into a single environment; this is motivated in the case where developers have different tasks (e.g. modeling or database designing phase) and do not need some tools or prefer certain tools rather than others. The problem is now faced through tools which enable users to add functionalities by defining plugins (see for example Eclipse); in this way developers can customize environments according to their preferences.

2.3 Microsoft Visual Studio

Microsoft Visual Studio[1] is a commercial IDE that supports the entire life-

[1] http://www.microsoft.com/visualstudio

cycle of software development oriented to imperative languages. In particular, it allows development of the following kind of both console and graphical user interface applications:

- Windows Forms applications;
- Web sites, Web Applications, and Web Services.

The IDE is customized for Microsoft Windows environments and includes a code editor supporting *IntelliSense* as well as code refactoring. Visual Studio supports different programming languages like *C/C++* (via *Visual C++*), *VB.NET* (via *Visual Basic .NET*), *C#* (via *Visual C#*), and supports also other languages such as Python and web oriented languages like *XML/XSLT*, *HTML/XHTML*, *JavaScript* and *CSS*.

Figure 2.2 shows an overview of the *Visual C#* tool included in *Visual Studio*.

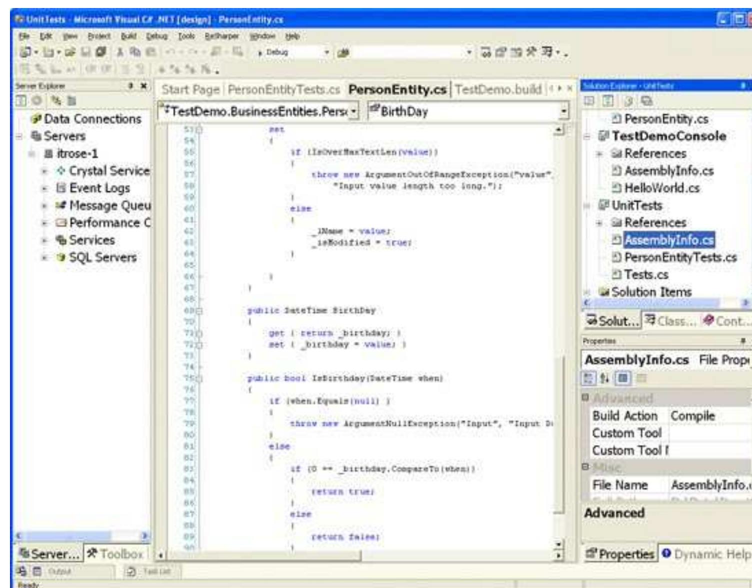


Figure 2.2: *Visual Studio* IDE overview for C#.

The figure shows panels useful for workspace organization (organizing file in projects), for editing the code by exploiting a colored text editor, for showing a tree representation of elements of code and project through an outline, and for visualizing and editing properties.

2.3.1 Main Features

Visual Studio includes a **Code Editor** for syntax highlighting and code completion (using IntelliSense) for variables, functions and methods as well as loops and queries. Auto-completions are shown in a popup list box, appearing on top of the code editor. The editor also supports setting bookmarks in code for quick navigation, offers an advanced search, allows code blocks to be collapsed and includes a task list.

Refactoring is also included in the editor. It allows for parameter reordering, variable and method renaming, interface extraction and encapsulation of class members inside properties, among others.

The editor provides feedback about syntax and compilation errors, which are flagged with a red underline; warnings are marked with a green underline.

Visual Studio provides a **Debugger** that can be used for debugging applications written in any language supported by *Visual Studio*. In the running phase, the debugger can display the code as it is being run. In this case a step-by-step code execution can be exploited by setting breakpoints, which allows execution to be stopped temporarily, with the purpose of monitoring the values of variables at the current state of the execution. Breakpoints can be triggered when conditions are met in the execution phase. The code can be also modified in the debugging phase; in this case the program is automatically re-run from the modified points.

Visual Studio offers the following tools of visual designing:

- *Windows Forms Designer* : is used to build graphical user interface applications by exploiting a visual editor for drawing windows forms;
- *Web designer/development* : a web-site editor and designer;
- *Class designer* : is used to edit classes, including its members, using UML modeling. The UML diagrams are translated into the source code. Recent releases of *Visual Studio* offer the reverse-reengineering feature from source code to UML diagrams;
- *Data designer* : can be used to graphically edit database schemas, typed tables, primary and foreign keys and constraints. It can also be exploited to design queries from the graphical view;
- *Mapping designer* : is used to design mappings between database schemas and classes that encapsulate data.

Visual Studio allows developers also to write plugins for extending its functionality. Packages of plugins are created using the *Visual Studio SDK* providing tools and allowing also other programming languages to be extended.

2.4 Eclipse

The *Eclipse Platform*[2] can be seen as a “set” (not just one) of integrated development environments that can be installed, expanded and configured inside the platform. In particular, the system can be entirely configured to introduce a complete IDE for writing software programs using different languages like *C++*, *Perl*, *Python* and so on. The flexibility of the architecture allows also arbitrary tools to be introduced which can potentially do anything.

Importantly, *Eclipse* is frequently used to develop Java applications. To this end it offers a wide set of tools supporting Java developers on the entire software development process. Figure 2.3 shows an overview of *Eclipse* in the Java perspective.

[2] `http://www.eclipse.org`

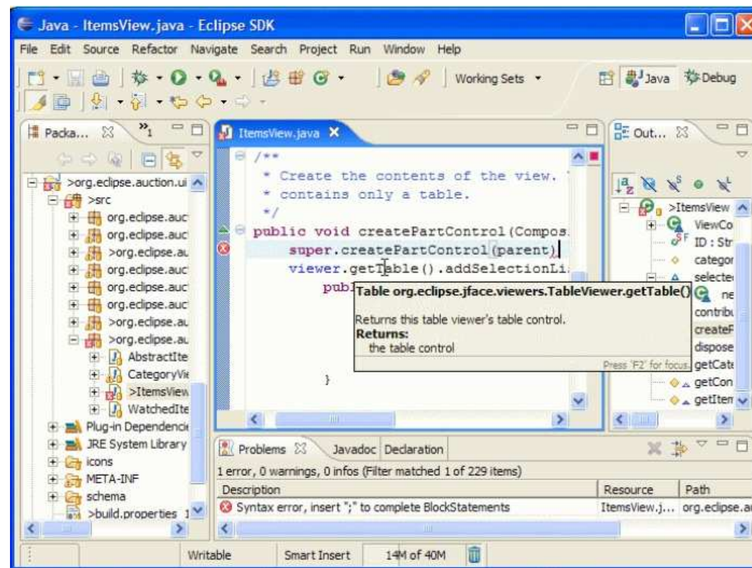


Figure 2.3: Eclipse for Java IDE overview.

2.4.1 Main Features

For a description of the main features of *Eclipse*, we refer especially to *Eclipse for Java* because it offers a wider set of interesting features compared to the versions of *Eclipse* for other languages.

Eclipse for Java

The main features of *Eclipse for Java* are summarized in the following:

- *Project Organization* : allows organization of Java source files laid out in Java package directories. Moreover, unrestricted other files, such as program resources and design documentation can be managed by the environment. Users can easy browse Java projects in terms of Java-specific elements: packages, types, methods, and fields;

- *Java Source Code Editor* : allows editing of Java files in an advanced editor that offers keyword and syntax coloring, code formatter, automatic indentation, and code completion for legal completions of method, variables and so on. The editor offers also "fast auto-completion" when users write keywords; for example, if the user writes `syso` and makes a keystroke on CTRL+SPACE, the editor writes `System.out.println` automatically;

- *Visual Editor* : useful for drawing windows and panels in a graphical way;

- *Outline Navigation* : shows declaration structure, represented by a tree, of the program. In the outline, methods, static fields, as well as visibility of objects (private, protected, public), are presented to users in the graphical way. The outline is updated automatically while editing;

- *Error Management and Quick Fixes* : errors are shown in different places: (i) on an Error Console, (ii) on the editor near the wrong part of the code, (iii) and on the outline nearness wrong Java methods. Errors are detected while writing and can be immediately fixed by exploiting Quick Fixes;

- *Annotations and Javadoc Management* : Java annotations and Javadocs are easy managed by *Eclipse*. Javadocs are extracted from source files and used to be shown to the user in an easy way by exploiting specific panels;

- *Refactoring* : allows editing of parts of code in a controlled way for improving code structure without changing behavior; for example, renaming of methods and update of references can be made in a safe way. A preview utility allows users to see the results of the refactoring before of performing it;

- *Search* : the search utility allows users to find declarations of references to packages, types, methods, and fields. Matches are highlighted as annotations in the editor;

- *Compare* : useful to compare Java compilation units showing, for example, the changes to individual Java methods;

- *Run* : a Run Configuration can be set up to Run, from the environment, the Java program in a separate target Java virtual machine. The results of the execution can be printed to the internal console which provides standard output (*stdout*), standard input (*stdin*) and standard error (*stderr*) streams;

- *Debug* : debug of Java programs is performed by exploiting graphical tools and views. The tools allow users to view threads and stack frames, set breakpoints and step through method source code and inspect and modify fields and local variables. The program code can be edited in the debugging phase; in this case classes are dynamically reloaded.

Other Features and Extensibility

Eclipse offers also other features, not mentioned in the previous list, which are in general available in all versions of *Eclipse* accustomized to other programming languages.

The creation and the installation of new plugins allows, finally the installation of more and more features and tools. In particular, plugins allow the introduction of new types of editors and views and set new perspectives which arrange old and new views in order to suit new user tasks. Moreover, the available standard tools of *Eclipse* can be extended by introducing new actions, pop-up content menu, action sets and shortcuts.

2.5 Common features of IDEs

Integrated Development Environments usually present a wide variety of common features that are used so much by novice and domain expert users. Managing large project softwares, for example, requires a structured organization of the

project elements (like files).In the following, common features present in imperative and declarative IDEs are presented:

- *Workspace organization* : for easy management of a large number of files that probably need some compiling/building;
- *Automatic completion* : is activated in the writing phase of a program and consists in asking to the IDE suggestions for completing the code the user is currently writing. For example, in the case where the user started to write a variable, the system can suggest the entire variable to be completed automatically;
- *Code Template* : is also an automatic completion but, in this case, the system tries to recognize whether the code written by the user, belongs to some common pattern. For example if the user starts to write a "`for`" cycle, the system recognizes that it is a `for` and can ask the user just to fill some field composing the `for` (e.g. counters); the system, finally, will automatically write the entire code;
- *Error checking* : helping users to discover errors easily;
- *Syntax highlighting* : offered in text editors for keywords coloring (e.g. a different color for the "`if`" keyword in Java) and for highlighting line containing errors;
- *Quick fixes* : offering a list of suggestions for automatic fixing of errors;
- *Testing Tools* : offering a suite of tools to help the user for automatic errors discovery by asserting an expected result on the execution;
- *Debugger* : a crucial feature for detecting why the program has an unexpected behaviour. In Imperative Programming Languages different debugging techniques were proposed like a *step-by-step* execution where for each step, a line (or a set of lines) of code is (are) executed and the state of the program (e.g. variables values at the current step) is shown. Declarative programming languages have fewer debugging techniques and tools;
- *Visual Editor* : helping programmers to write programs fast by exploiting a full graphical interface. For example, *Eclipse* and *Visual Studio* offer a visual editor to easily draw window forms, while some tools for the declarative language SQL offer a *QBE* editor for query composition;
- *Configuration of the execution* : allows the user to set the execution properties before running the program. It is useful in the case where a user wants to pass to the executable some execution options or wants to configure piping between execution processes;
- *Execution in the same environment* : this feature is very useful because the user is not obliged to compile the program in an external executable file but can ask the IDE to execute the program directly. The feature is important for checking the correctess of the program at any moment or for debugging purposes;

- *Refactoring* : is an important feature that the user exploits every time he needs to edit something in the code and wants to be sure that the editing does not affect other parts of the program. For example, renaming a class in Java means that other files that use this class must refer to the class using the new name;
- *Customizable user interface and tools* : useful to allow the user to add and organize internal or external (plugins) tools that he needs.

The features listed above were object of several analisys and considerations in our work. To face with the ASP approach, since it is different from imperative languages because of the declarative nature of ASP, these features have been re-designed to deal with ASP programs.

Chapter 3

Answer Set Programming

In this Chapter some preliminaries notions concerning Answer Set Programming (ASP) [51] are first presented with a brief introduction of some common language extensions used in the examples of this thesis. Some relevant modularity properties of ASP programs are introduced and, finally, the use of ASP as a tool for Knowledge Representation (KR) is described. We assume the reader to be familiar with logic programming, and refer to both paper [5] and [50] for some introductory material on ASP.

3.1 ASP Language

3.1.1 Syntax

Let $\mathcal{V}$ be a set of *variables*, $\mathcal{C}$ be a set of *constants*, and $\mathcal{S}$ be a set of *predicates symbols*. We assume variables to be strings starting with uppercase letters and constants to be non-negative integers or strings starting with lowercase letters. Moreover, predicates (represented by strings starting with lowercase letters) have each one an associated *arity* (non-negative integer) representing the number of terms contained in the predicate. Moreover, the language allows the use of built-in predicates (i.e., predicates with a fixed meaning) for the common arithmetic operations (i.e., $=$, $\leq$, $\geq$, $+$, $\times$, etc.; usually written in infix notation).

A variable or a constant is a *term*. A *standard atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. An atom $p(t_1, \ldots, t_n)$ is ground if $t_1, \ldots, t_n$ are constants.

A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Terms : Conj\}$, where *Terms* is a list of terms (variables or constants) and *Conj* is a conjunction of standard atoms, that is, *Conj* is of the form $a_1, \ldots, a_k$ and each a_i $(1 \leq i \leq k)$ is a standard atom.

Example 3.1.1. For example, consider the set term

$$\{\mathtt{X}:\mathtt{p}(\mathtt{X},\mathtt{a}),\mathtt{q}(\mathtt{X})\}.$$

Suppose that ground atoms `p(1,a)`, `p(2,a)`, `q(1)` and `q(2)` are true.

The set term stands for the set of all the possible values of the variable X such that the conjunction `a(X,a), p(X)` is true, i.e., $\{X \,|\, p(X,a)$ *and* $q(X)$ *are*

true}. In this case, the possible values are

$$\{\mathtt{X}\mathtt{:}\mathtt{p}(\mathtt{X},\mathtt{a}),\mathtt{q}(\mathtt{X})\} \quad \mathtt{X} \in \{\mathtt{1},\mathtt{2}\}.$$

□

A *ground set* is a set of pairs of the form $\langle consts : conj \rangle$, where *consts* is a list of constants and *conj* is a conjunction of ground standard atoms.

A *literal* is either a standard atom, or a standard atom preceded by the *negation as failure* symbol `not`, or an aggregate atom (see Section 3.2). Complementary standard literals are of the form `a` and `not a`, where `a` is a standard atom. For a standard literal ℓ, we denote by $\neg.\ell$ the complement of ℓ. If L is a set of standard literals, we denote with $\neg.L$, by a little abuse of notation, the set $\{\neg.\ell \mid \ \ell \in L\}$.

Example 3.1.2. Example of literals are `person(joe)`, `father(joe,john)`, `not father(joe,joseph)`, `#max{X: age(X,Y), person(Y)} < 18`.

□

A *rule* r is a construct of the form

$$\mathtt{a_1}\ \mathtt{v} \cdots \mathtt{v}\ \mathtt{a_n}\ \text{:-}\ \ell_1,\ \cdots,\ \ell_\mathtt{m}.$$

where $\mathtt{a_1}\ \cdots\ \mathtt{a_n}$ are standard atoms, $\ell_1,\ \ldots,\ \ell_m$ are literals, $n \geq 0$, and $m \geq 0$. The disjunction $\mathtt{a_1}\ \mathtt{v} \cdots \mathtt{v}\ \mathtt{a_n}$ is the *head* of r, and the conjunction $\ell_1,\ \ldots,\ \ell_m$ is the *body* of r. If the body is empty ($m = 0$), the rule is called *fact*, while if the head is empty ($n = 0$), the rule is called *integrity constraint* (or simply *constraint*). We denote the set of head atoms by

$$H(r) = \{\mathtt{a_1},\ \ldots,\ \mathtt{a_n}\},$$

and the set of body literals by

$$B(r) = \{\ell_1,\ \ldots,\ \ell_m\}.$$

Moreover, the set of positive standard body literals is denoted by $B^+(r)$ and the set of negative standard body literals by $B^-(r)$. We denote also the set of atoms of the body as $At(B(r))$; the set of atoms of a rule are denoted by

$$At(r) = H(r) \cup At(B(r))$$

A rule r is ground if all the literals in $H(r)$ and in $B(r)$ are ground.

A *program* $\mathcal{P}$ is a set of rules and it is ground if all its rules are ground. In this case $At(\mathcal{P})$ are all atoms contained in $\mathcal{P}$. Accordingly with the database terminology, a predicate occurring only in facts is an *EDB* predicate, while all the other ones are *IDB* predicates; the set of facts of $\mathcal{P}$ is denoted by $EDB(\mathcal{P})$.

The variables of a rule can be *local* when they appear solely in sets terms of r, or *global* otherwise. A rule r is *safe* if both the following conditions hold:

1. for each global variable X of r there is a positive standard literal $\ell \in B^+(r)$ such that X appears in ℓ;
2. each local variable of r appearing in a symbolic set $\{Terms : Conj\}$ also appears in $Conj$.

Note that the first condition is the standard safety condition adopted in Logic Programming to guarantee that the variables are range restricted [89]. A program is considered safe if all its rules are safe.

Example 3.1.3. Consider the following rules:

```
person(X) :- father(X,Y).
animal(X) v mineral(X) :- not plant(X).
p(X) :- q(X,Y,V), #max{Z : r(Z), a(Z,V)} > Y.
p(X) :- q(X,Y,V), #sum{Z : r(X), a(X,S)} > Y.
```

The first rule is safe, while the second is not because it violates the first condition due to variable X. The third rule is safe, while the fourth is not because variable Z violates the second condition. □

3.1.2 Semantics

Given an ASP program $\mathcal{P}$, the *universe* of $\mathcal{P}$, denoted by $\mathcal{U}_{\mathcal{P}}$, is the set of all constants appearing in $\mathcal{P}$. The *base* of $\mathcal{P}$, denoted by $\mathcal{B}_{\mathcal{P}}$, is the set of standard atoms constructible from predicates of $\mathcal{P}$ with constants in $\mathcal{U}_{\mathcal{P}}$.

A *substitution* is a mapping from a set of variables to $\mathcal{U}_{\mathcal{P}}$. A *global substitution* for a rule r consists in substituting the set of the global variables of r to $\mathcal{U}_{\mathcal{P}}$; a *local substitution* for a rule r consists on substituting the set of local variables of r to $\mathcal{U}_{\mathcal{P}}$. Given a set term without global variables $S = \{Terms : Conj\}$, the *instantiation of* S is the following ground set:

$$inst(S) = \{\langle \sigma(Terms) : \sigma(Conj)\rangle \mid \sigma \text{ is a local substitution for } S\}.$$

A *ground instance* of a rule r is obtained by applying a global substitution σ for r, and then replacing every set term S in $r\sigma$ by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of instances of all the rules in $\mathcal{P}$.

Example 3.1.4. Consider the following program $\mathcal{P}_1$:

```
q(1) :- not p(2,2).        q(2) :- not p(2,1).
p(2,2) :- not q(1).        p(2,1) :- not q(2).
t(X) :- q(X), #sum{Y : p(X,Y)} > 1.
```

The instantiation $Ground(\mathcal{P}_1)$ of $\mathcal{P}_1$ is the following program:

```
q(1) :- not p(2,2).        q(2) :- not p(2,1).
p(2,2) :- not q(1).        p(2,1) :- not q(2).
t(1) :- q(1), #sum{⟨1 : p(1,1)⟩, ⟨2 : p(1,2)⟩} > 1.
t(2) :- q(2), #sum{⟨1 : p(2,1)⟩, ⟨2 : p(2,2)⟩} > 1.
```

□

An *interpretation* I for an ASP program $\mathcal{P}$ is a consistent set of standard

ground literals, that is, $I \subseteq \mathcal{B}_{\mathcal{P}} \cup \neg.\mathcal{B}_{\mathcal{P}}$. The set of all the interpretations of $\mathcal{P}$ is denoted by $\mathcal{I}_{\mathcal{P}}$.

The evaluation of a standard literal ℓ with respect to an interpretation I results in one of the following alternatives:

- if $\ell \in I$, then ℓ is true with respect to I;
- if $\neg.\ell \in I$, then ℓ is false with respect to I;
- otherwise, if $\ell \notin I$ and $\neg.\ell \notin I$, then ℓ is undefined with respect to I.

Also Set Terms can be evaluated with respect to an interpretation, giving rise to a multiset, a value, and a truth value, respectively.

An interpretation I satisfies a rule r if at least one of the following conditions is satisfied:

- $H(r)$ is true with respect to I;
- a literal in $B(r)$ is false with respect to I;
- all the atoms in $H(r)$ and a literal in $B(r)$ are undefined with respect to I.

An interpretation M is a *model* of an ASP program $\mathcal{P}$ if all the rules r in $Ground(\mathcal{P})$ are satisfied with respect to M. A model M for $\mathcal{P}$ is minimal if no model N for $\mathcal{P}$ exists such that $N^+ \subset M^+$.

Example 3.1.5. Consider again the program $\mathcal{P}_1$ of Example 3.1.4. Let I_1 be an interpretation for $\mathcal{P}_1$ such that $I_1 = \{q(2), p(2,2), t(2)\}$. Then I_1 is a minimal model of $\mathcal{P}_1$. □

Definition 3.1.6 ([34]). Given a ground ASP program $\mathcal{P}$ and an interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting all rules in which a body literal is false w.r.t. I. I is an *answer set* of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.

Example 3.1.7. Consider the following program:

$$\mathcal{P}_2 : \{\mathtt{p(a)} :- \mathtt{p(X)}.\}$$

then, we have that

$$Ground(\mathcal{P}_2) = \{\mathtt{p(a)} :- \mathtt{p(a)}.\}$$

and two interpretation $I_1 = \{\mathtt{p(a)}\}$, $I_2 = \emptyset$. Consequently, $Ground(\mathcal{P}_2)^{I_1} = Ground(\mathcal{P}_2)$ and $Ground(\mathcal{P}_2)^{I_2} = \emptyset$ hold. I_2 is the only answer set of $\mathcal{P}_2$ (because I_1 is not a minimal model of $Ground(\mathcal{P}_2)^{I_1}$).

□

Note that any answer set A of $\mathcal{P}$ is also a model of $\mathcal{P}$ because $Ground(\mathcal{P})^A \subseteq Ground(\mathcal{P})$, and rules in $Ground(\mathcal{P}) - Ground(\mathcal{P})^A$ are satisfied w.r.t. A.

Another possible characterization is given by the notion of *supportedness*. Given an interpretation I for a ground program $\mathcal{P}$, we say that a ground atom

A is *supported* in I if there exists a *supporting* rule r in the ground instantiation of $\mathcal{P}$ such that the body of r is true w.r.t. I and A is the only true atom in the head of r.

Proposition 3.1.8. *[69, 64, 6]* If M is an answer set of a program $\mathcal{P}$, then all atoms in M are supported.

In the remainder of this thesis, the set of all answer sets of a program $\mathcal{P}$ will be denoted with $ANS(\mathcal{P})$.

3.2 Aggregates and Language Extension

In this Section we describe aggregate functions supported by ASP and extended constructs to solve optimization problems and interact with databases. These constructs are supported in the DLV [63] system as well as in other ASP systems.

3.2.1 Aggregate Functions

An *aggregate function* in DLV [25] is an expression of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant. According to the notation of the DLV system for representing aggregates, S is a set term of the form $\{\,Vars : Conj\,\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard atoms, and f is an *aggregate function symbol*.

The most common aggregate functions are listed below. All these functions consider the element of the set term as integers:

- `#min`: The function identifies the minimal term among the elements of the set term. This function is undefined for the empty set;
- `#max`: The function identifies the maximal term among the elements of the set term. This function is undefined for the empty set;
- `#count`: The function determines the number of elements of the set term;
- `#sum`: The function determines the sum of the elements of the set term;
- `#times`, The function determines the product of the elements of the set term;
- `#avg`, The function determines the average value of the elements of the set term. This function is undefined for the empty set.

An *aggregate atom* is a structure of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq\}$ is a comparison operator, and T is a term (variable or constant). An aggregate atom $f(S) \prec T$ is ground if T is a constant and S is a ground set.

Example 3.2.1. The following is a rule with an aggregate atom that counts the number of true instances of predicate `p`:

$$\texttt{numP(X)} \;{:-}\; \#\texttt{count}\{\texttt{X} : \texttt{p(X)}\} = \texttt{X}.$$

□

3.2.2 Weak contraints

Weak constraints [16] allow one to express desiderata, on the contrary to standard constraints that always have to be satisfied, that is, to express conditions that *should* be satisfied. They are useful to formulate several optimization problems in an easy and natural way.

Weak constraints can be weighted according to their importance. In particular a higher weight indicates a more important constraint. In the presence of weights, best models minimize the sum of the weights of the violated weak constraints. Weak constraints can also be prioritized; in this case the semantics minimizes the violation of the constraints of the highest priority level first; then the lower priority levels are considered one after the other in descending order.

Weak constraints are specified as follows.

$$\leftsquigarrow \texttt{Conj}.[\texttt{Weight} : \texttt{Level}]$$

where `Conj` is a conjunction of literals, while `Weight` and `Level` are positive integers. `Weight` and `Level` can also be variables that also appear in a positive literal of `Conj`. The weight or the priority or both can be also omitted.

The informal meaning of a $\leftsquigarrow B$ is "try to falsify B," or "B should preferably be false." Intuitively, the semantics coincides with the answer sets minimizing the number of violated (unsatisfied) weak constraints.

3.2.3 Language extensions for Database Management

Predicates used in ASP programs can be (possibly complex) views on database tables, which are stored in different DBMSs. In this case the facts of a program can be imported from an external database and solvers can use DBMSs also for performing reasoning that needs a large quantity of memory and for storing the results of the execution. Database functionalities were implemented in DLV and DLV^{DB} [88] which support the following features:

- *Import/Export directives*: `Import` and `Export` commands are used for interacting with databases. The `Import` command retrieves data from a table through the query specified by the user in the SQL syntax and creates one atom for each retrieved tuple. The `Export` command generates a new tuple into a table for each new truth value derived for that predicate by the program evaluation;

- *TYP files*: store advanced database mappings through definition of *TYP directives*. *TYP files* are supported by the DLV^{DB} solver.

DLV^{DB} has also the ability to evaluate logic programs directly on databases with a very limited use of main-memory resources. Program predicates can be mapped to (possibly complex and distributed) database views and data can be easily specified as input or output for the program. The evaluation requires some explicit specifications for the mappings between input and output data and program predicates, as well as proper indications for the temporary relations possibly needed for the mass-memory evaluation.

Import/Export directives and *TYP directives* will be illustrated in details in Chapter 8 by introducing syntax and mapping examples.

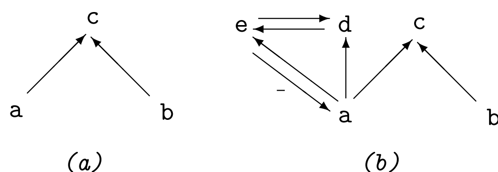


Figure 3.1: *Graphs (a) $DG(\mathcal{P}_4)$, and (b) $DG(\mathcal{P}_5)$*

3.3 Dependency Graphs

In this Section, relevant properties are described with syntactic sub-classes of ASP programs; moreover, modularity aspects consideration of programs are faced. The definition of *Dependency Graph* is given first.

Definition 3.3.1 (Dependency Graph). A *Dependency Graph* of a ground program $\mathcal{P}$ is a directed graph $DG(\mathcal{P}) = (N, E)$ of $\mathcal{P}$, where:

1. the nodes in N are the atoms of $\mathcal{P}$;
2. there is an edge in E from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that b appears in $H(r)$ and a appears in $B^+(r)$;
3. there is an edge, "marked" with a stroke, in E from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that b appears in $H(r)$ and a appears in $B^-(r)$;

The graph $DG(\mathcal{P})$ singles out the dependencies of the head atoms of a rule r from the atoms in its body.

Example 3.3.2. Consider the following two programs:

$$\mathcal{P}_4 = \{\texttt{a v b. c :- a. c :- b.}\}$$
$$\mathcal{P}_5 = \mathcal{P}_4 \cup \{\texttt{d v e :- a. d :- e. e :- d. a :- not e.}\}$$

The dependency graph $DG(\mathcal{P}_4)$ is depicted in Figure 3.1 (a), while the dependency graph $DG(\mathcal{P}_5)$ is depicted in Figure 3.1 (b).

□

Definition 3.3.3 (Positive Dependency Graph). A *Positive Dependency Graph*, denoted as $DG^+(\mathcal{P})$, is a dependency graph having only the properties 1 and 2 of the Definition 3.3.1.

The graph $DG^+(\mathcal{P})$ singles out the dependencies of the head atoms of a rule r from the positive atoms in its body. Negatives edges in this case are not represented in the graph.

Definition 3.3.4 (Strongly Connected Components). Given a program $\mathcal{P}$ and its dependency graph $DG(\mathcal{P})$, a *Strongly Connected Component (SCC)* of $DG(\mathcal{P})$, denoted by $SCC(DG(\mathcal{P}))$, is a maximal set of atoms $S \subset At(\mathcal{P})$ such that $a \leftarrow b$ for every $a, b \in S$.

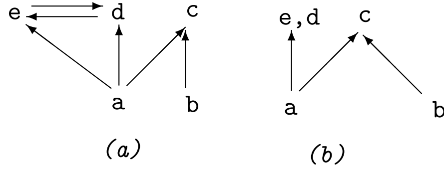


Figure 3.2: *Graphs (a) $DG^{+}(\mathcal{P}_5)$, and (b) $SCC(DG^{+}(\mathcal{P}_5))$*

$SCC(DG(\mathcal{P}))$ singles out that all atoms belonging to a *Component* depend upon each other.

Example 3.3.5. Consider the program $\mathcal{P}_5$ of the Example 3.3.2. The positive dependency graph $DG^{+}(\mathcal{P}_5)$ is depicted in Figure 3.2 (a), while the Strongly Connected Components of positive dependency graph $SCC(DG^{+}(\mathcal{P}_5))$ is depicted in Figure 3.2 (b). Note that the SCC groups the nodes e and d because there is a cycle between them.

□

The dependency graphs presented here are "atom-oriented" (the nodes of the graph are atoms). However, in particular for non-ground programs, a *Predicate Dependency Graph* can be considered, having the same properties as the atom dependency graph. In this case nodes are represented by predicate symbols instead of atoms. For example, in the rule

$$\mathtt{a(X)} :- \mathtt{c(X)}, \mathtt{d(Y)}$$

the nodes are `a`, `c`, and `d` and there is an edge from `c` to `a` and an edge from `d` to `a`.

3.4 Relevant Sub-Classes

The class of *stratified* programs and the class of *head-cycle free* programs are now presented along with some examples.

3.4.1 Stratified Programs

Definition 3.4.1. Functions from $\| \| : \mathcal{B}_{\mathcal{P}} \rightarrow \{0, 1, \dots\}$ from the ground set of literal $\mathcal{B}_{\mathcal{P}}$ to finite ordinals are called level mappings of $\mathcal{P}$.

Level mapping is used now for defining (locally) stratified programs.

Definition 3.4.2. A disjunctive ASP program $\mathcal{P}$ is called *(locally) stratified* [2, 78] if there is a level mapping$\| \|_s$ of $\mathcal{P}$ such that, for every rule r of $Ground(\mathcal{P})$,

(1) for any $l \in B^{+}(r)$, and for any $l' \in H(r)$, $\| l\|_s \leq \| l'\|_s$;

(2) for any $l \in B^{-}(r)$, and for any $l' \in H(r)$, $\| l\|_s < \| l'\|_s$;

(3) for any $l,\ l' \in H(r)$, $\|\ l\|_s = \|\ l'\|_s$.

Example 3.4.3. Consider the following two programs.

```
P6: p(a) v p(c) :- not q(a).   P7: p(a) v p(c) :- not q(b).
    p(b) :- not q(b).               q(b) :- not q(a).
```

It is easy to see that program $\mathcal{P}_6$ is stratified, while program $\mathcal{P}_7$ is not. A suitable level mapping for $\mathcal{P}_6$ is the following:

$$\|p(a)\|_s = 2\ \|p(b)\|_s = 2\ \|p(c)\|_s = 2\ \|q(a)\|_s = 1\ \|q(b)\|_s = 1\ \|q(c)\|_s = 1$$

As for $\mathcal{P}_6$, an admissible level mapping would need to satisfy $\|p(a)\|_s < \|q(b)\|_s$ and $\|q(b)\|_s < \|p(a)\|_s$, which is impossible. □

An important property of locally stratified ASP programs is given by the following proposition.

Proposition 3.4.4. *[50]* A locally stratified normal (non-disjunctive) ASP programs has at most one answer set.

It is worthwhile noting that the presence of disjunction invalidates Proposition 3.4.4. Indeed, the program {`a v b.`} has two answer sets, namely {`a`} and {`b`}.

3.4.2 Head-Cycle Free Programs

Another relevant property of disjunctive ASP programs is head-cycle freeness (HCF) [10]; the dependency graphs allow us to define HCF programs.

Definition 3.4.5. A program $\mathcal{P}$ is *HCF* iff there is no rule r in $\mathcal{P}$ such that two atoms occurring in the head of r occur in a single cycle of $G(\mathcal{P})$.

Example 3.4.6. The dependency graphs given in Figure 3.1 reveal that program $\mathcal{P}_4$ of Example 3.3.2 is HCF and that program $\mathcal{P}_5$ is not HCF, as rule `d v e :- a.` contains in its head two atoms belonging to the same cycle of $\mathcal{G}(5)$. ▮ □

It has been shown that HCF programs are computationally easier than general (non-HCF) programs.

Proposition 3.4.7. *[10, 30]* Deciding whether an atom belongs to some answer set of a ground HCF program $\mathcal{P}$ is NP-complete, while deciding whether an atom belongs to some answer set of a ground (non-HCF) program $\mathcal{P}$ is Σ_2^P-complete.

3.5 Modularity aspects

Programming languages, in general, allows the programmer to split a program into several modules which interact through well-defined input/output interfaces. Consequently the entire program can be viewed as a composition of its component modules. For Answer Set Programming, various modularity investigations have been exploited in order to decompose ASP programs in modules that are logically linked by an input/output interface. The modularity techniques presented here consists in splitting an ASP program into two modules and, a more complex operation, decomposing a ground ASP program into different modules by exploiting the *DLP-functions* definition [59].

3.5.1 Splitting an ASP program

An important result presented in [67] allows an ASP program $\mathcal{P}$ to be splitted into two modules that have a clear interface.

Definition 3.5.1. (Splitting Set). A *splitting set* [67] of a program $\mathcal{P}$ is any subset U of atoms in $\mathcal{P}$ such that, for each rule $r \in \mathcal{P}$, if $H(r) \cap U \neq \emptyset$ then $At(r) \subset U$.

In this case we say that U splits $\mathcal{P}$ into two distinct sub-programs $\mathcal{P}_b$ and $\mathcal{P}_t$. $\mathcal{P}_b$, called the *bottom* of $\mathcal{P}$ w.r.t. U, consists of all the rules that satisfy the property of Definition 3.5.1, and $\mathcal{P}_t$, called the *top* of $\mathcal{P}$ w.r.t. U, is composed of all the rules contained in $\mathcal{P} \setminus \mathcal{P}_b$. A consequence is the fact that all the atoms contained in the head of some rule of $\mathcal{P}_t$ are not contained in U.

Theorem 3.5.2. (Splitting Theorem). *Let a program $\mathcal{P}$ with a splitting set U of $\mathcal{P}$, let $\mathcal{P}_b$ the* bottom *of $\mathcal{P}$ w.r.t. U, and $\mathcal{P}_t$ the* top *of $\mathcal{P}$ w.r.t. U. A set M of atoms is a* consistent answer set *for $\mathcal{P}$ iff $M = X \cup Y$ where X is an answer set of $\mathcal{P}_b$ and Y is an answer set of $\mathcal{P}_t \cup X$.*

In other words, this theorem says that an answer set of $\mathcal{P}$ can be found by calculating an answer set X of $\mathcal{P}_b$ and giving X as input to $\mathcal{P}_t$ for calculating an answer set of $\mathcal{P}_t$.

The splitting theorem by Lifschitz and Turner [67] was also generalized to the non-ground case [31] by considering, as splitting sets, predicates of the program instead of ground atoms and taking in consideration the *Predicate Dependency Graph* of the program.

3.5.2 DLP-functions

Definition 3.5.3. (DLP-function). A DLP-function Π [59], is a quadruple $\langle R, I, O, H \rangle$ where I, O, and H are pairwise distinct sets of *input atoms, output atoms* and *hidden atoms*, respectively and R is a (disjunctive) ground program such that for each rule $r \in R$,

- $At(r) \subseteq I \cup O \cup H$, and
- if $H(r) \neq \emptyset$, then $H(r) \cap (O \cup H) \neq \emptyset$.

The atoms $I \cup O$ are *visible atoms* and accessible by other DLP-functions. The atoms H are *not visible* and, consequently, not accessible by other DLP-functions.

The second property of the definition 3.5.3 ensures that at least one atom of the head must be either an output atom or a hidden atom; this is to ensure that no rules must interfere with the input. In this way, when a head contains some output or hidden atom, and also some input atom, the input atoms act very much as atoms contained in negative body.

Definition 3.5.4. Given two DLP-functions $\Pi_1 = \langle R_1, I_1, O_1, H_1 \rangle$ and $\Pi_2 = \langle R_2, I_2, O_2, H_2 \rangle$, they respect the *input/output interfaces* of each other iff the following properties are respected:

- $(I_1 \cup O_1 \cup H_1) \cap H_2 = \emptyset$

- $(I_2 \cup O_2 \cup H_2) \cap H_1 = \emptyset$
- $(O_1 \cap O_2) = \emptyset$
- $Def_{R_1}(O_1) = Def_{R_1 \cup R_2}(O_1)$
- $Def_{R_2}(O_2) = Def_{R_1 \cup R_2}(O_2)$

In such a way, the composition of Π_1 and Π_2 consists of:

$$\Pi_1 \oplus \Pi_2 = \langle R_1 \cup R_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2 \rangle.$$

The first two properties of definition 3.5.4 say that the hidden atoms of Π_2 must not interfere with the atoms of Π_1 and vice-versa. The third property assumes that the output atoms of Π_1 and Π_2 are disjoint. The last two properties say that the output atoms O_1 are defined by rules contained in R_1 and the output atoms O_2 are defined by rules contained in R_2. When two DLP-functions are composed together, the result is also a DLP-function composed by the rules R_1 and R_2. From the input atoms of one function, it is important to remove the output atoms of the other function and vice-versa.

Definition 3.5.5. Given two DLP-functions Π_1 and Π_2 defined as before, given the composition $\Pi_c = \Pi_1 \oplus \Pi_2$, and given $DG^+(\Pi_c)$ the positive dependency graph of Π_c, the DLP-functions Π_1 and Π_2 are *mutually independent* iff does not exist a *strongly connected component* S in $DG^+(\Pi_c)$ such that $S \cap O_1 \neq \emptyset$ and $S \cap O_2 \neq \emptyset$. In this case, the *join* $\Pi_u = \Pi_1 \sqcup \Pi_2 = \Pi_c$ is defined.

The following definitions are useful for calculating the answer sets of a program, by calculating the answer sets of the respective DLP-functions.

Definition 3.5.6. Given two DLP-functions Π_1 and Π_2, two interpretations $M_1 \subseteq At(\Pi_1)$ and $M_2 \subseteq At(\Pi_2)$ are *mutually compatible* w.r.t. Π_1 and Π_2 if

$$M_1 \cap (O_2 \cup H_2) = M_2 \cap (O_1 \cup H_1).$$

Definition 3.5.7. Given two DLP-functions Π_1 and Π_2 such that $\Pi_1 \sqcup \Pi_2$ is defined. Given any sets of interpretations A_1 of Π_1 and A_2 of Π_2, the *natural join* of A_1 and A_2, denoted by $A_1 \bowtie A_2$, is the set of interpretations

$A_1 \bowtie A_2 = \{M_1 \cup M_2 | M_1 \in A_1, M_2 \in A_2$, and M_1 and M_2 are mutually compatible. $\}$

Definition 3.5.8. For DLP-functions, the following semantic operator is given:

$SM(F) = \{M \subseteq At(\Pi) | M \in MM(\Pi^M)\}$ where Π is a DLP-function.

Using the definitions just introduced, the following *module theorem* is defined.

Theorem 3.5.9. (Module Theorem [59]). *For all DLP-functions* Π_1 *and* Π_2 *such that* $\Pi_1 \sqcup \Pi_2$ *is defined, and for any mutually compatible interpretations* $M_1 \subseteq At(\Pi_1)$ *and* $M_2 \subseteq At(\Pi_2)$*,* $M_1 \cup M_2$ *is an answer set of* $\Pi_1 \sqcup \Pi_2$*, if and only if* M_1 *is an answer set of* Π_1 *and* M_2 *is an answer set of* Π_2*.*

3.6 Knowledge Representation

Answer Set Programming is a tool for knowledge representation and reasoning applied in several application domains, from classical deductive databases to artificial intelligence. ASP is particular useful for handling incomplete knowledge and non-monotonic reasoning, and allows for encoding problems in a declarative way. Writing an ASP program is easy as describing the problem domain and the complexity of the reasoning task is hidden by using a dedicated ASP system. Moreover, the (optional) separation of a fixed non-ground program from an input database allows one to obtain uniform solutions over varying instances.

ASP is a powerful formalisms, and allows complex problems to be expressed. The expressiveness of ASP captures all problems belonging to the second level of the polynomial hierarchy (the complexity class Σ_2^P). This high expressive power is significantly relevant for approaching hard problems like, for example, solving planning and diagnosis problems, or, in the field of Artificial Intelligence, for solving problems not reducible to SAT instances.

ASP allows the encoding of problems in an intuitive and concise way following a *Guess&Check* programming methodology (originally introduced in [29] and refined in [63]). According to this approach a program $\mathcal{P}$ which encodes a problem $\mathbf{P}$ consists of the following parts:

Input Instance: An instance F of the problem $\mathbf{P}$ is specified in input using a database of facts.

Guess Part: A set of disjunctive rules $G \subseteq \mathcal{P}$, referred to the *guessing part*, is used the define the search space.

Check Part: The search space is then pruned by the *checking part*, consisting of a set of constraints $C \subseteq \mathcal{P}$ which impose some properties to be verified.

Basically the input instance and the guessing part, represent the *candidate solutions* to the problem. By adding the check part those solutions are filtered in order to guarantee that the answer sets of the resulting program represent exactly the admissible solutions for the input instance. The following example represents the typical application of the *Guess&Check* methodology.

Example 3.6.1. Suppose that we want to partition a set of people into two groups, but we also know that some pairs of people dislike each other, thus we have to keep those two in different groups. Assume that the input instance consists of the following facts:

$$\texttt{person(bob).}\ \texttt{person(eve).}\ \texttt{dislike(bob,eve).}$$

Applying the guess&check methodology, the guess part would model the possible partition of persons to groups:

$$\texttt{group(P,1)}\ \texttt{v}\ \texttt{group(P,2)}\ \ {:}{-}\ \ \texttt{person(P).}$$

The resulting program (input instance + guess) produces the following answer

sets:

$\{\texttt{person}(\texttt{bob}),\ \texttt{person}(\texttt{eve}),\ \texttt{dislike}(\texttt{bob},\texttt{eve}),\ \texttt{group}(\texttt{bob},1),\ \texttt{group}(\texttt{eve},1)\}$
$\{\texttt{person}(\texttt{bob}),\ \texttt{person}(\texttt{eve}),\ \texttt{dislike}(\texttt{bob},\texttt{eve}),\ \texttt{group}(\texttt{bob},1),\ \texttt{group}(\texttt{eve},2)\}$
$\{\texttt{person}(\texttt{bob}),\ \texttt{person}(\texttt{eve}),\ \texttt{dislike}(\texttt{bob},\texttt{eve}),\ \texttt{group}(\texttt{bob},2),\ \texttt{group}(\texttt{eve},1)\}$
$\{\texttt{person}(\texttt{bob}),\ \texttt{person}(\texttt{eve}),\ \texttt{dislike}(\texttt{bob},\texttt{eve}),\ \texttt{group}(\texttt{bob},2),\ \texttt{group}(\texttt{eve},2)\}$

However, we want to discard assignments in which people that dislike each other belong to the same group. To this end, we add the checking part by writing the following constraint:

$$:-\ \texttt{group}(\texttt{P1},\texttt{G}),\ \texttt{group}(\texttt{P2},\texttt{G}),\ \texttt{dislike}(\texttt{P1},\texttt{P2}).$$

Now, by adding the constraint to the original program we obtain the intended answer sets (the checking part acted as a sort of filter):

$\{\texttt{person}(\texttt{bob}),\ \texttt{person}(\texttt{eve}),\ \texttt{dislike}(\texttt{bob},\texttt{eve}),\ \texttt{group}(\texttt{bob},1),\ \texttt{group}(\texttt{eve},2)\}$
$\{\texttt{person}(\texttt{bob}),\ \texttt{person}(\texttt{eve}),\ \texttt{dislike}(\texttt{bob},\texttt{eve}),\ \texttt{group}(\texttt{bob},2),\ \texttt{group}(\texttt{eve},1)\}$

□

In the following, we illustrate some ASP programs examples referred to knowledge representation. More in details, we present some problems that can be naturally encoded using ASP. The programs will be used in the remainder of this thesis.

3.6.1 Reachability

Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of edges in A. In different terms, the problem amounts to computing the transitive closure of the relation A.

The input graph is encoded by assuming that A is represented by the binary predicate `edge(X,Y)`, where a fact `edge(a,b)` means that G contains an edge from `a` to `b`, i.e., $(a, b) \in A$; whereas, the set of nodes V is not explicitly represented, since the nodes appearing in the transitive closure are implicitly given by these facts.

The following program defines a predicate `reachable(X,Y)` containing all facts `reachable(a,b)` such that `b` is reachable from `a` through the edges of the input graph G:

$$\texttt{r}_1 : \texttt{reachable}(\texttt{X},\texttt{Y}) :-\ \texttt{edge}(\texttt{X},\texttt{Y}).$$
$$\texttt{r}_2 : \texttt{reachable}(\texttt{X},\texttt{Y}) :-\ \texttt{edge}(\texttt{X},\texttt{U}),\ \texttt{reachable}(\texttt{U},\texttt{Y}).$$

The first rule states that the node Y is reachable from the node X if there is an edge in the graph from X to Y, whereas the second rule represents the transitive closure by stating that node Y is reachable from node X if there is a node U such that U is directly reachable from X (there is an edge from X to U) and Y is reachable from U.

As an example, consider a graph represented by facts: `edge(1,2). edge(2,3).`

`edge(3,4)`. The answer set of the program together with the facts is

$$\{\texttt{reachable}(1,2),\ \texttt{reachable}(2,3),\ \texttt{reachable}(3,4),\ \texttt{reachable}(1,3),\ \texttt{reachable}(2,4), \texttt{reachable}(1,4), \texttt{arc}(1,2), \texttt{arc}(2,3), \texttt{arc}(3,4)\}$$

The atoms {`reachable(1,2), reachable(2,3), reachable(3,4)`} are inferred by exploiting the rule r_1, whereas the other atoms containing the predicate `reachable` are inferred by using rule r_2.

3.6.2 Hamiltonian Path

Given a finite directed graph $G = (V, A)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once?

This is a classical NP-complete problem in graph theory. Suppose that graph G is specified by using facts over predicates `node` (unary) and `edge` (binary), and the starting node a is specified by the predicate `start` (unary). Then, the following program $\mathcal{P}_{hp}$ solves the *Hamiltonian Path* problem:

```
% Guess arcs of the path
r1 : inPath(X, Y) v outPath(X, Y) :- edge(X, Y).
% Auxiliary rules
r2 : reached(X) :- start(X).
r3 : reached(X) :- reached(Y), inPath(Y, X).
% Checking part : specify constraints on solution.
% Each vertex in the path must have
% at most one incoming and one outgoing edge.
r4 : :- inPath(X, Y), inPath(X, Y1), Y <> Y1.
r5 : :- inPath(X, Y), inPath(X1, Y), X <> X1.
% All vertexes must be in the path.
r6 : :- node(X), not reached(X), not start(X).
```

The disjunctive rule (r_1) guesses a subset S of the edges to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary predicate `reached` is defined, which specifies the set of nodes which are reached from the starting node. Doing this is very similar to reachability, but the transitivity is defined over the guessed predicate `inPath` using rule r_3. Note that `reached` is completely determined by the guess for `inPath`, no further guessing is needed.

In the checking part, the first two constraints (namely, r_4 and r_5) ensure that the set of edges S selected by `inPath` meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two edges starting at the same node, and (ii) there must not be two edges ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. The mentioned constraints can be also written in a different way by introducing aggregate functions. In particular,

the constraints r_4 and r_5 can be written in the following way:

$$\mathtt{r}_4: \;:\!-\; \mathtt{node(X2)}, 2 <= \#\mathtt{count}\{\mathtt{X1}:\mathtt{inPath(X1,X2)}\}.$$
$$\mathtt{r}_5: \;:\!-\; \mathtt{node(X2)}, 2 <= \#\mathtt{count}\{\mathtt{X1}:\mathtt{inPath(X2,X1)}\}.$$

3.6.3 Maximal Clique

The maximum clique is a classical hard problem in graph theory requiring to find the largest clique (i.e., a complete subgraph of maximal size) in an undirected graph. Suppose that the graph G is specified by using facts over predicates `node` (unary) and `edge` (binary), then the following program solves the problem.

```
% Guess the clique
r1 : inClique(X1) v outClique(X1) :- node(X1).
% Order edges in order to reduce checks
r2 : uedge(X1, X2) :- edge(X1, X2), X1 < X2.
r3 : uedge(X2, X1) :- edge(X1, X2), X2 < X1.
% Ensure property.
r4 : :- inClique(X1), inClique(X2), not uedge(X1, X2), X1 < X2.
r5 : :~ outClique(X2).
```

The disjunctive rule ($\mathtt{r}_1$) guesses a subset S of the nodes to be in the clique, while the rest of the program checks whether S constitutes a clique, and the weak constraint ($\mathtt{r}_5$) maximizes the size of S (since it prefers interpretations in which the number of true `outClique` instances is minimized). An auxiliary predicate *uedge* exploits an ordering to reduce the time spent in checking.

3.6.4 Maze Generation

A maze is an M x N grid where two distinct cells are indicated as entrance and exit and each cell of the grid can be empty or a wall. The maze we are considering must satisfy the following properties:

- all the cells placed near the edges of the grid are walls, except entrance and exit that are empty;
- in a 2 x 2 square of the grid, cells can be neither all empty nor all walls;
- if two walls are placed on a diagonal of a 2 x 2 square, then not both of their common neighbors are empty;
- each wall cell cannot be completely surrounded by empty cells;
- there is always a path (a finite sequence of cells horizontally or vertically adjacent to the next cell in the sequence) from the entrance to every empty cell of the grid.

The Maze Generation problem consists in generating mazes with fixed dimensions of the grid. The problem has recently been proved to be *NP-Complete* [1].

The following program $\mathcal{P}_{mz}$ solves the *Maze Generation* problem:

```
% Startup
grid(X, Y) :- col(X), row(Y).
adjacent(X, Y, X, Y1) :- grid(X, Y), Y1 = Y + 1, row(Y1).
adjacent(X, Y, X, Y1) :- grid(X, Y), Y1 = Y - 1, row(Y1).
adjacent(X, Y, X1, Y) :- grid(X, Y), X1 = X + 1, col(X1).
adjacent(X, Y, X1, Y) :- grid(X, Y), X1 = X - 1, col(X1).
border(1, Y) :- row(Y).
border(X, 1) :- col(X).
border(X, Y) :- row(Y), maxCol(X).
border(X, Y) :- col(X), maxRow(Y).
% Input
empty(X, Y) :- input_empty(X, Y).
wall(X, Y) :- input_wall(X, Y).
% Condition 1 :
wall(X, Y) v empty(X, Y) :- grid(X, Y), not border(X, Y),
      not entrance(X, Y), not exit(X, Y).
% Condition 2 :
wall(X, Y) :- border(X, Y), not entrance(X, Y), not exit(X, Y).
empty(X, Y) :- entrance(X, Y).
empty(X, Y) :- exit(X, Y).
% Condition 3 :
 :- wall(X, Y), wall(X1, Y), wall(X, Y1), wall(X1, Y1),
      X1 = X + 1, Y1 = Y + 1.
 :- empty(X, Y), empty(X1, Y), empty(X, Y1), empty(X1, Y1),
      X1 = X + 1, Y1 = Y + 1.
% Condition 4 :
 :- wall(X, Y), wall(Xp1, Yp1), empty(Xp1, Y), empty(X, Yp1),
      Xp1 = X + 1, Yp1 = Y + 1.
 :- wall(Xp1, Y), wall(X, Yp1), empty(X, Y), empty(Xp1, Yp1),
      Xp1 = X + 1, Yp1 = Y + 1.
% Condition 5 :
 :- wall(X, Y), not border(X, Y), not wallWithAdjacentWall(X, Y).
wallWithAdjacentWall(X, Y) :- wall(X, Y), adjacent(X, Y, W, Z), wall(W, Z).
% Condition 6 :
reach(X, Y) :- entrance(X, Y).
reach(XX, YY) :- adjacent(X, Y, XX, YY), reach(X, Y), empty(XX, YY).
 :- empty(X, Y), not reach(X, Y).
```

The input of the problem is represented by a number of row and column facts, defining the rows and the columns of the grid. These facts are represented by the predicates `row(X)` and `col(X)` and are given as a range of consecutive, ascending integers, starting from 1. The maximum number of rows and columns are provided too and they are represented by the predicates `maxRow(X)` and `maxCol(X)`. Two facts are provided, indicating the entrance and the exit; they are represented by the predicates `entrance(X,Y)` and `exit(X,Y)` and are composed of the column and row index. The entrance and the exit are placed on the edges of the grid. Finally, one or more facts, indicating cells that are known to be empty or contain walls, can be given; they are represented by the predicates `wall(X, Y)` and `empty(X, Y)`.

The rules belonging to *Startup* part of the program initialize the grid by identifying cells, adjacent cells and borders. The predicates *maxCol* and *maxRow* are used to limitate the dimension of the grid. The *Input* part initializes some cells as empty or walls according to the elements contained in the predicates `input_empty` and `input_wall`. The rule of *Condition 1* guesses cells to be empty or walls. The rules of *Condition 2* state that each cell at an edge of the grid is a wall, except entrance and exit that are empty. The rules of *Condition 3* forbid the existance of a 2 x 2 square of empty cells or walls. The rules of *Condition 4* state that if two walls are on a diagonal of a 2 x 2 square, then not both of their common neighbors are empty. The rules of *Condition 5* state that no wall is completely surrounded by empty cells. Finally, the rules of *Condition 6* state that there is a path from the entrance to every empty cell.

Chapter 4

ASPIDE

This Chapter describes *ASPIDE* in its various aspects. A graphical interface overview is first delineated and, subsequently, all the features offered by the IDE are listed and described in detail. Moreover, use case examples are exploited that illustrate the single features. At the end of the Chapter, the system architecture and implementation of *ASPIDE* is depicted. *ASPIDE* can be freely downloaded from the system web site `http://www.mat.unical.it/ricca/aspide`.

4.1 *ASPIDE* Graphical Interface Overview

The system interface of *ASPIDE* is depicted in Figure 4.1, where the main components are outlined in different numbered zones.

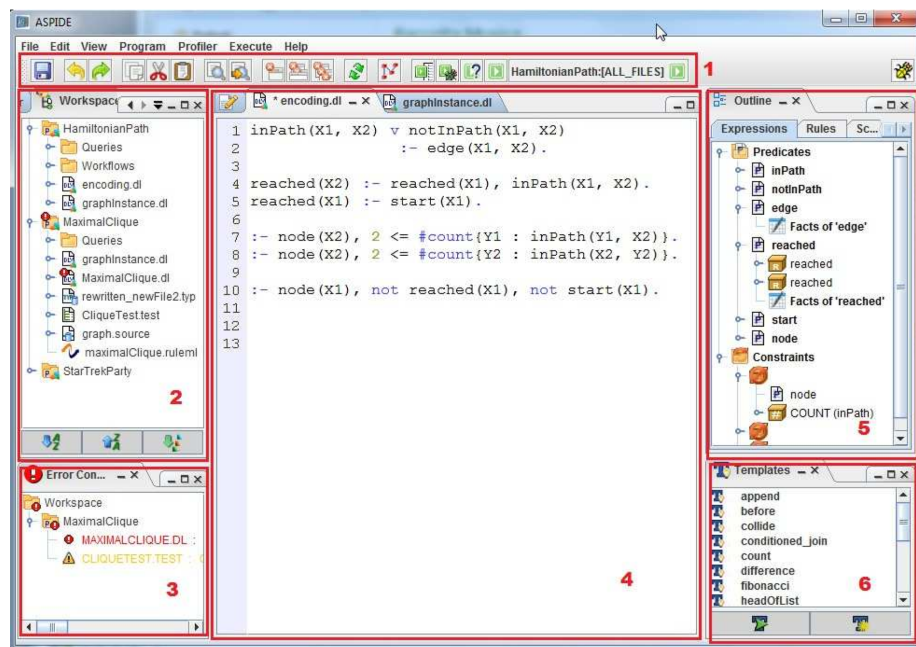


Figure 4.1: The *ASPIDE* graphical user interface.

In the upper part of the interface (zone 1) a toolbar allows the user to call the most common operations of the system (from left to right: save files, undo/redo, copy & paste, find & replace, switch between visual to text editor, run the solver/profiler/debugger). In the center of the interface there is the main editing area (zone 4), organized in a multi-tabbed panel possibly collecting several open files. The left part of the interface is dedicated to the explorer panel (zone 2), and to the error console (zone 3). The explorer panel lists projects and files included in the workspace, while the error console organizes errors and warnings according to the project and files where they are localized. On the right, there are the outline panel (zone 5) and the templates panel (zone 6). The first shows an outline of the currently edited file, while the latter reports a list of user-defined templates for exploiting common ASP patterns to be inserted in programs. The one shown in figure 4.1 is the standard appearance of the system, which can be however modified, since panels can be moved as the user likes.

4.2 System Features

In the following the features of *ASPIDE* are first listed and then described in details by exploiting examples of use. For the examples, we start with an empty workspace and we use some of the programs described in Section 3.6 of Chapter 3 to illustrate single features. By stressing and modifying the program encodings, many possible ways and shortcuts will be shown in order to create files, edit the program, fix problems and execute the program by exploiting the DLV solver.

The features of *ASPIDE* are listed below:

- *Workspace organization*: the system allows ASP programs to be organized in projects à la Eclipse, which are collected in a special directory (called workspace);
- *Advanced text editor*: the editing of ASP files is simplified by an advanced text editor, which provides several functionalities: from simple text coloring to auto-completion of predicates and variables names;
- *Code template*: *ASPIDE* provides support for assisted writing of rules (guessing patterns, aggregates, etc.), as well as automated writing of entire sub-programs (e.g., transitive closure rules) by exploding code templates;
- *Visual Editor*: the users can *draw* logic programs by exploiting a full graphical environment that offers a QBE-like tool for building logic rules. This feature is described in Chapter 5;
- *Annotation management for ASP programs*: *ASPIDE* exploits annotations for ASP programs for indicating, e.g., rule names, predicate schemas (name, arity, optional data-type), handle database connectivity and so on;
- *Outline navigation*: *ASPIDE* creates an outline view which graphically represents program elements;
- *Schema management and interaction with databases*: interaction with external databases is made easy by a fully graphical *import/export* tool for

specifying tables mappings in an assisted way. Advanced schema management and interaction with databases are described in Chapter 8;

- *Errors and Warnings management*: errors and warnings in *ASPIDE* can be easy managed by exploiting specific panels and actions;
- *Dynamic code checking and errors highlighting*: syntax errors and relevant conditions (like safety) are checked *while typing programs*: portions of code containing errors or warnings are immediately highlighted;
- *Quick fix*: the system suggests quick fixes to reported errors or warnings, and applies them (on request) by automatically changing the affected part of the code;
- *Dependency graph*: the system provides a graphical representation of the dependency graph of a program;
- *Configuration of the execution*: this feature allows configuration of input programs and execution options;
- *Presentation of results*: the output of the program (either answer sets, or query results) are visualized in a tabular representation, in a text-based console or in a custom way;
- *Unit Testing for ASP*: *ASPIDE* provides a unit testing framework in the style of JUnit and allows developers to compose units, inputs and assertions on expected outputs. This feature is described in Chapter 6;
- *Debugger and Profiler*: semantic errors detection as well as code optimization can be done by exploiting graphic tools.;
- *User-defined Plugins*: Developers can implement user-defined plugins for extending *ASPIDE* to deal with new input formats, program rewritings, and even customizing the format of solver results. This feature is described in Chapter 7.

In the following, the above mentioned functionalities are described in details.

4.2.1 Workspace organization

The system allows one to organize ASP programs in projects similar to Eclipse, which are collected in a special directory (called workspace). This facilitates the development of complex applications by organizing modules (or projects) in a space where either different parts of an encoding or several equivalent encodings solving the same problem are stored. When *ASPIDE* is open, the system asks for specifying a folder to be used as workspace; *ASPIDE* will show all the projects, files and folders that are contained in the workspace.

The workspace is organized in several kinds of file which *ASPIDE* manages according to their nature. In particular, *ASPIDE* allows one to manage:

- *DLV files*, defining ASP programs in both DLV syntax [63] and the syntax of the Third Answer-Set Programming System Competition `ASPCore` [19];

- *TYP files*, specifying a mapping between program predicates and database tables in the DLV^{DB} syntax [88];
- *TEST files*, defining unit tests;
- *TEXT files*, that are opened in a standard text editor;
- *PLUGIN files*, associated with any file format handled by a user defined plugin.

Predicates defined in both *DLV files* and *TYP files* are shown to the user so that he can use them for some purpose like drag/drop to some other file and for arity error checking with predicates defined in other files of the same project. *PLUGIN files*, that will be described in Chapter 7, can also be configured to contain predicates.

For the workspace elements, two kinds of view are associated (fig. 4.2):

- *Project Explorer* which shows, in a tree view, only projects and files that are associated with some nature;
- *Workspace Explorer* which shows, in a tree view, all the projects, folder and all kinds of file that are contained physically in the workspace.

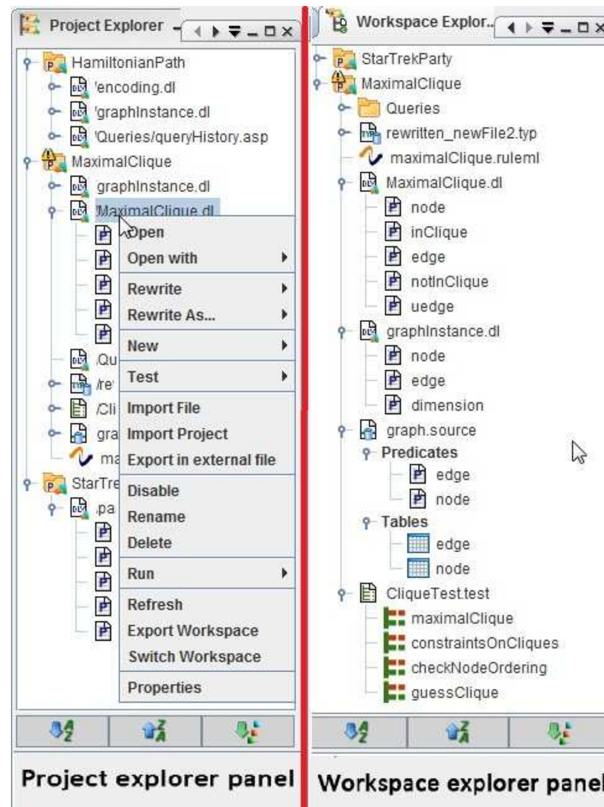


Figure 4.2: Project and Workspace Explorer panels.

Acting on these views, the user can do simple file operations (renaming, deleting, moving...) and complex file operations like changing the nature of a file according to the file extension, and disabling files, so that *ASPIDE* can ignore it, avoiding error checking operations. The operations can be done by exploiting both the *File* menu of *ASPIDE* and the popup menu. At the bottom of both panels there are also buttons useful for files ordering.

Example 4.2.1. For defining a new workspace we open *ASPIDE* and choose a folder as workspace (fig. 4.3). In the main window of *ASPIDE* we use the menu *File* to create a new project and we give *HamiltonianPath* as project name (fig. 4.4). We create a new *DLV File*, named *encoding.dl*, by right-clicking on the *Workspace Explorer* (fig. 4.5); the file will contain the program encoding of the Hamiltonian Path problem.

□

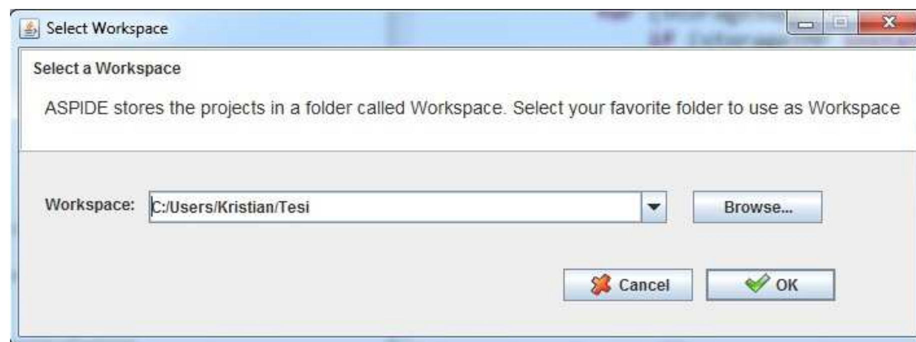


Figure 4.3: Select the workspace.

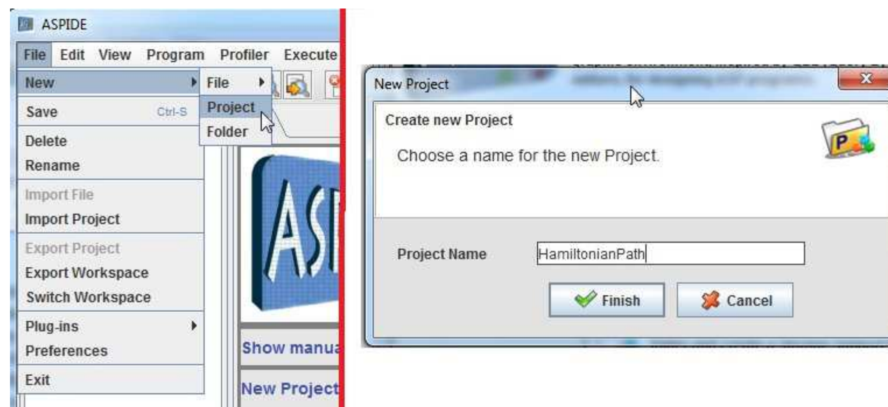


Figure 4.4: Create a new project.

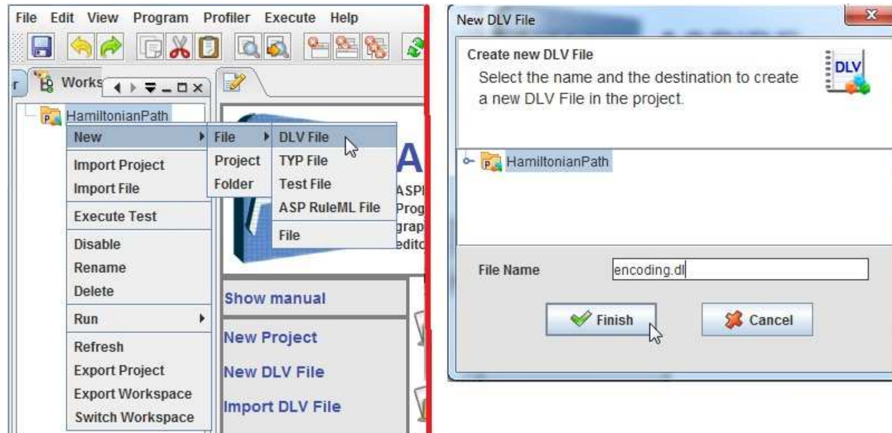


Figure 4.5: Create a new DLV File.

4.2.2 Advanced text editor

The presence of an editor that provides a set of advanced features is indispensable for a good development environment. In particular, besides the core functionality that basic text editors offer (like, code line numbering, find/replace, undo/redo, copy/paste, etc.), *ASPIDE* offers others advanced functionalities that are shown in the following.

The editor performs keyword outlining (such as " :− ") and dynamic highlighting of predicate names, variables, strings, and comments. Specific colors are also exploited for indicating predicates mapped to external databases or properly defined by annotations (more details about annotations are reported in the next).

The refactoring tool allows one to modify predicate names and variables, among others, in a guided way. For instance, variable renaming in a rule is done by considering bindings of variables, so that common side effects of find/replace are avoided by ensuring that variables/predicates/strings occurring in other expressions remain unchanged. Another interesting feature regarding refactoring consists in selecting rules and applying some rewriting (possibly implemented in a user-defined plug-in, more details follow in Chapter 7).

4.2.3 Automatic completion

The system is able to perform automatic completion of the editing phase on the text editor to complete predicate names, as well as variable names and to apply common ASP patterns. There are two different ways to perform automatic completion:

- *on line*, suggestions are made while writing, so that the user can choose to confirm what the editor suggests or continue writing normally;
- *on request*, by key-stroking CTRL+SPACE a popup window is opened and the user can choose possible suggestions.

For both *on line* and *on request* suggestions, predicate names are learned while writing, and extracted from the files belonging to the same project; variables are suggested by taking into account the rule we are currently writing; common ASP patterns are identified by actions and keywords we are currently writing on the editor. This helps while developing either an alternative encoding for the same problem (input/intermediate predicate names are ready to be suggested after the first file is completed) or when the same solution is divided in several files.

For the *on line* auto-completion, the possible suggestions are:

- *Disjunction completion*, when the user starts writing a disjunction, the editor suggests a possible way to complete the disjunction;
- *Atom completion*, when the user starts writing something on the editor, it suggests a possible atom to be written by considering predicates that start with the string that the user is currently writing; in case of n-ary predicates, the terms of the atom are filled with fresh variables. If the user types CTRL in this phase, other predicates starting with the same string are sequentially suggested;
- *Variable completion*, on the writing phase of variables in atoms, already used variables are suggested. This suggestion is useful because users often re-use the same variables introduced to the head, in the body of the rule they are writing.

Example 4.2.2 (Disjunction Completion). To insert the disjunctive rule r_1 of the Hamiltonian Path problem, we start writing `inPath(X1, X2) v` and the editor automatically suggests, as a possible way to complete the disjunction, `inPath(X1, X2) v notInPath(X1, X2)` (fig. 4.6). □

Figure 4.6: On-line auto-completion of a disjunction.

Example 4.2.3 (Atom completion). For the Hamiltonian Path problem, suppose that rule r_2, defining in which case a node is reached, was already added to the editor; in this way the new predicate `reached` with arity 1 was introduced. If we start writing rule r_3 by typing the first characters of `reached`, the editor suggests the atom `reached` filling the term with a new variable (fig. 4.7). □

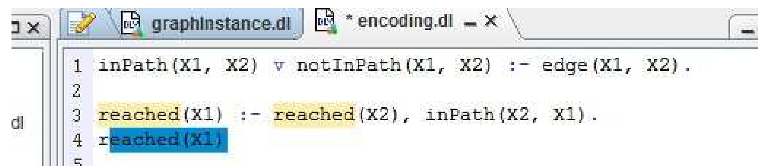


Figure 4.7: On-line auto-completion of an atom.

Example 4.2.4 (Variable completion). For the Hamiltonian Path problem, we will add rule r_3. When in the body of the rule we write `start`, the editor suggests variable `X1` that we have already used in the head of the rule (fig. 4.8). □

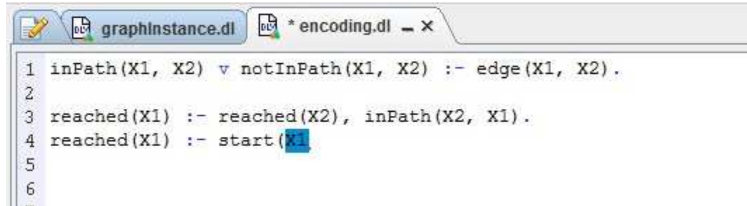


Figure 4.8: On-line auto-completion of a variable.

On request auto-completion is activated when the user key-strokes CTRL+ SPACE on the editing phase; in this case a popup window is opened showing possible (contextual) suggestions. The popup that appears is useful for completing, for example, predicate and variables names, in the same way as common IDEs for imperative programming language that suggest possible variables, functions and class names. Some IDEs offer also the possibility of using specific keywords for building quickly specific language constructs (e.g. if in Eclipse for Java we write *syso* and type CTRL+SPACE, the editor automatically substitutes `syso` with `System.out.println`). In *ASPIDE* were also introduced keywords that automatically build language constructs and common patterns.

The possible *on request* suggestions are:

- *Atom completion*, the user can complete the atom that he is currently writing; using CTRL+SPACE a popup window is opened showing the possible predicates, already used in the files of the projects, which can be used to complete the atom. In the same way the user can also add new atoms with predicates (which still do not exist) by specifying the arity of the predicate; in this way the atom is automatically built;

- *Variable completion*, variable in atoms can be suggested also using on-request auto-completion. Also in this case, already used variables in the rule will be suggested;

- *Common ASP patterns completion*, using specific keywords and typing CTRL+SPACE, the user can insert complex and common language construct fast. *ASPIDE* supports a set of code templates that helps the user to define the preferred construct (see next Paragraph for more details).

Example 4.2.5 (On Request Atom completion). Suppose we want to write again rule r_2 using the *on request* auto-completion. We write `reached` and type CTRL+SPACE; since the predicate `reached` still does not exist in the program, the editor gives to the user some feedback to build the new predicate. Using the suggestion the user can specify the arity of the predicate, so that the atom can be automatically built (fig. 4.9). □

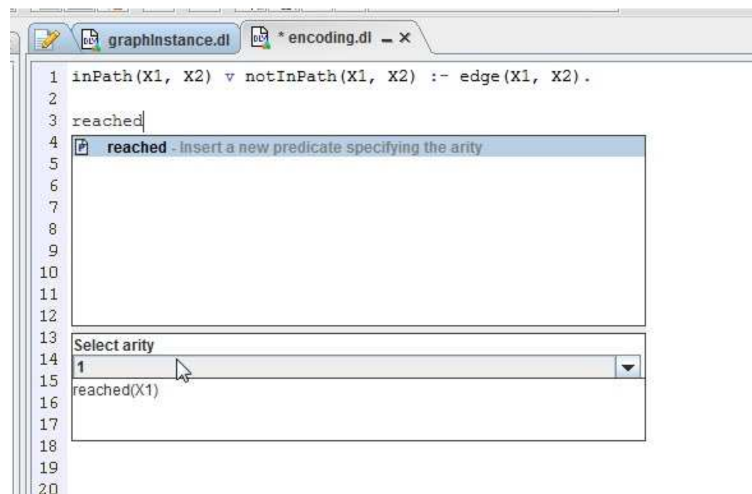


Figure 4.9: Atom auto-completion using a new predicate.

Example 4.2.6 (On Request Variable completion). For rule r_3 of the Hamiltonian Path problem, on writing the atom `start`, we use CTRL+SPACE and the editor suggests the variable `X1` that we have already used in the head of the rule (fig. 4.10). □

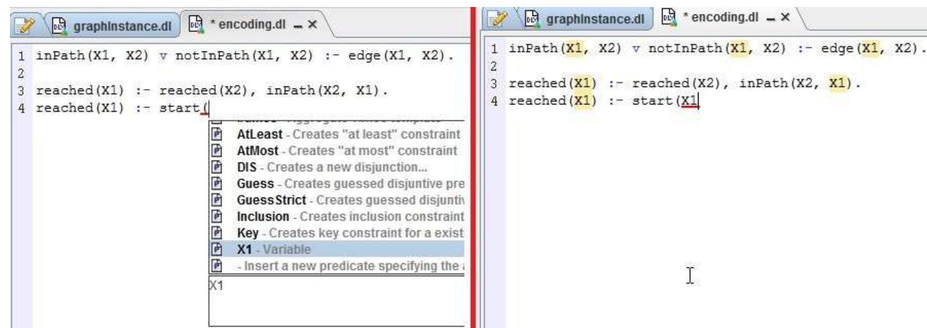


Figure 4.10: On-request auto-completion of a variable.

4.2.4 Code template

Common patterns for ASP like guessing patterns, aggregates, as well as subprograms like transitive closure rules, can be written in *ASPIDE* in an assisted way by exploiting *code templates*. Some code templates are available at the on-request auto-completion feature of the text editor; to this end the user must write the correct keyword and, by using CTRL+SPACE, the editor suggests code templates associated with the keyword. According to the chosen code template, the popup window provides a preview of the result and some information that the user can edit to make the code template as he prefers. *ASPIDE* allows users also to introduce new user-defined templates by exploiting a particular syntax.

In the following the code templates supported by *ASPIDE* and the user-defined code templates definition process are described. The descriptions are integrated by examples of use on the Hamiltonian Path problem.

Quick disjunction definition. The user can quickly define a new disjunction by using the keyword *DIS*,. When the user writes this word and types CTRL+SPACE, the editor asks the user, using the popup, how many new predicates (with default arity 1) should be added to the disjunction (fig. 4.11). Note that using this code template only one predicate can be specified.

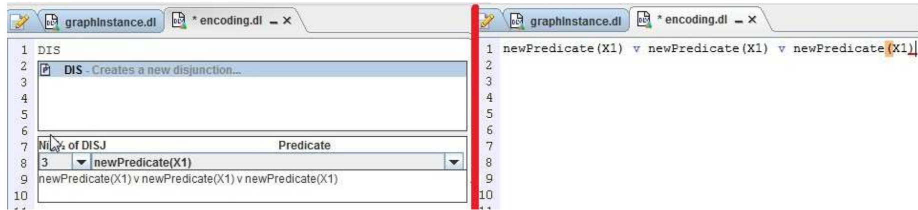


Figure 4.11: Auto-completion using the keywork DIS for writing a disjunction.

By expanding the same keyword, the editor can suggest disjunction with different number of atoms, a different arity for each atom and a different predicate name.

Example 4.2.7. If the user would like to write fast the disjunction `inPath(X1, X2) v inPath(X1, X2) v inPath(X1, X2)` he can write *DIS3inPath2* where the integer 3 is the number of atoms of the disjunction, the string *inPath* is the name of the predicate to appear in the disjunction and the integer 2 is the arity of the predicate (fig. 4.12). □

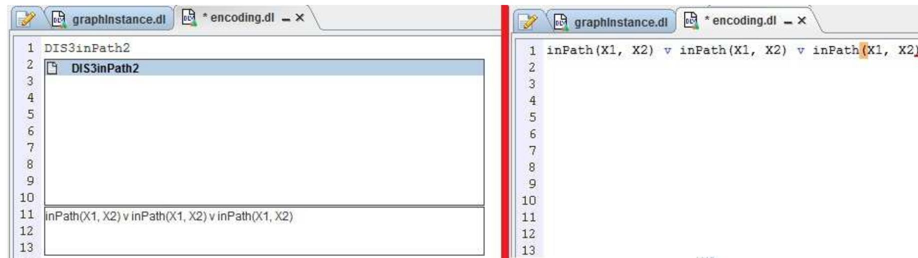


Figure 4.12: Code template using the keyword *DIS3inPath2* for building a disjunction.

A user can exploit the code template also in a faster way. Suppose the user would like to write a disjunction over a predicate, deciding whether corresponding atoms are either true or not, he can write a keyword in the form *DNOTpredname* where *predname* indicates the name of a predicate. For example the user can write the keyword *DNOTedge* to generate the disjunction `edge(X,Y) v notEdge(X,Y)`.

Guessing subsets in search space. As described in Chapter 3, guessing part of a program is generally composed of disjunctive rules performing the candidate solutions search. By using the keyword *Guess*, the user can write disjunctive rules that guess a subset of elements belonging to a set (e.g. in ASP, for guessing a subset of a set of elements represented by the predicate `element` a rule like `inSubset(X) v notInSubset(X) :- element(X).` can be used).

Example 4.2.8. For writing fast rule r_1 of the Hamiltonian Path problem we use the keyword *guess* and, by exploiting the popup window, we specify the predicate *path* with arity 2 (fig. 4.13). □

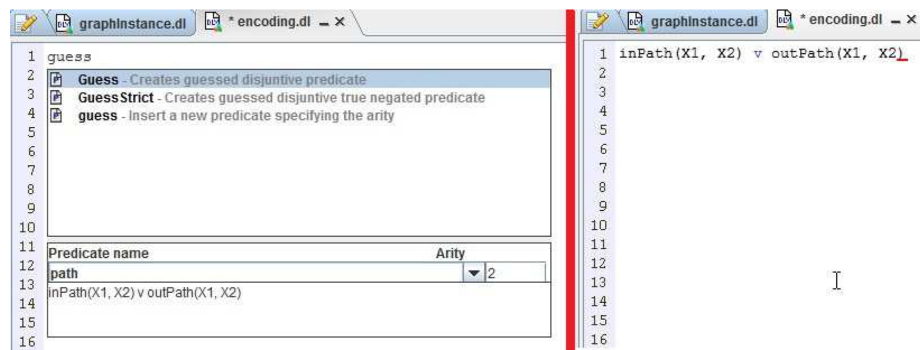


Figure 4.13: Code template using the keyword *guess* for writing a disjunction that guesses paths.

A user can exploit the code template also in a faster way. For example, if the user would like to guess values of a predicate, he can write a keyword in the form *GUESSpredname* where *predname* indicates the name of a predicate. For example, if the user would like to guess a subset of edges, he can write the keyword *GUESSedge* to generate the disjunction `inEdge(X,Y) v outEdge(X,Y) :- edge(X,Y).`

Another keyword similar to *Guess*, named *GuessStrict*, allows users to guess a subset of elements (in the same way) but changing the atoms involved in the head. In particular, the head will contain an atom and its true negated version (eg. `a` v ¬`a`).

Example 4.2.9. We can write rule r_1 of the Hamiltonian Path problem in a different way, using *guessStrict* and specifying, in the popup window, the predicate `inPath` with arity 2; the editor will write the disjunction `inPath(X1)` v ¬`inPath(X1)` (fig. 4.14). □

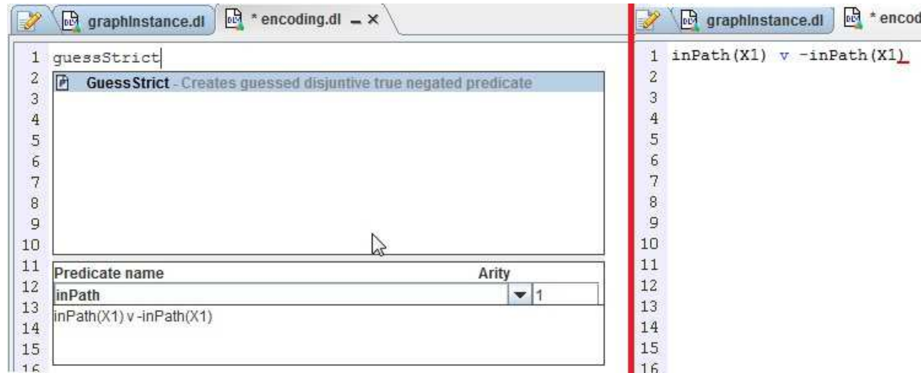


Figure 4.14: Code template using the keyword *guessStrict* for writing a disjunction containing a normal atom and its true negated version.

Aggregates definition. The keywords involved in the code template for definining aggregates functions are `#count`, `#min`, `#max`, `#sum`, `#times`. Writing one of these keywords, the popup window allows one to set values of the aggregate atom, like lower and upper guards, compare operators and an atom to be included in the aggregate.

Example 4.2.10. We want to write the constraint r_4 of the Hamiltonian Path problem; once we have written `node(X2)`, we write the character `#` and type CTRL+SPACE. After selecting *#count* from the popup window, we fill the template with the value *2* as lower guard, select $<=$ as lower operator and select `inPath(X1, X2)` as atom of the aggregate. At the end of the operation, the aggregate is written to the editor (fig. 4.15). □

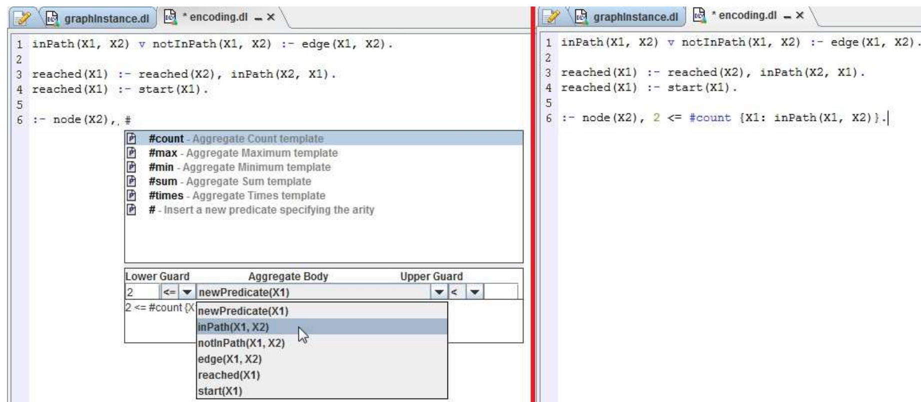


Figure 4.15: Code template for writing the aggregate `#count`.

Key constraints definition. The keyword *Key* allows one to create *key constraints* for predicates. Given a set of attributes S of a predicate, the purpose of a key constraint is to forbid, in the same Answer Set, the existence of two or more atoms with the same values for the attributes S and different values for the remaining attributes.

Example 4.2.11. For the Hamiltonian Path problem, the constraint r_5 says that, given a node, it must not have two or more outgoing edges. In other words the first attribute of **`inPath`** must be unique in any answer set; consequently the first attribute can be seen as a (primary) key. We now write the constraint r_5 in a different way: we use the keyword *key* and we exploit the code template to set the predicate **`inPath`** and specify the first attribute as key attribute; the constraint :– **`inPath(X1,X2), inPath(X1,Y2), X2 <> Y2.`** will be written to the editor (fig. 4.16). □

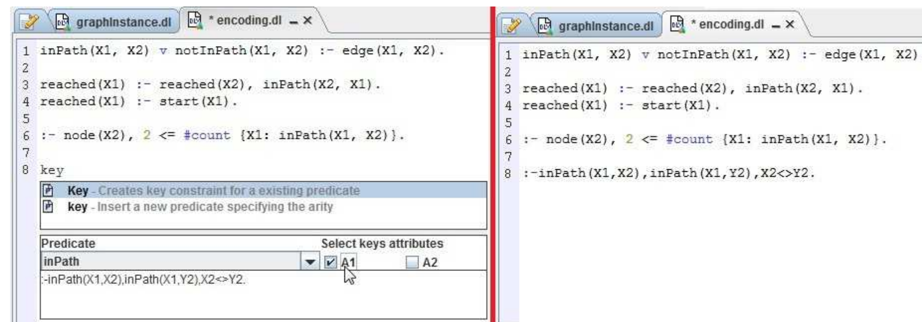


Figure 4.16: Code template for writing a key constraint.

A user can exploit the code template also in a faster way. For example, if the user would like to generate a key for a predicate, he can write a keyword in the form *KPredAttrpos*, where *Pred* indicates the predicate name, while *Attrpos* is an integer representing the attribute position where the user wants to applicate the key constraint. For example, the previous constraint :– **`inPath(X1,X2), inPath(X1,Y2), X2 <> Y2.`** can be written using the keyword *KinPath1*.

Inclusion constraints definition. The keyword *Inclusion* allows the user to define an inclusion constraint saying that elements (also obtained after some attributes projection) of a predicate must be a subset of elements of some other predicate. The code template allows one to specify the predicates involved and the attributes of the predicate to be considered.

Example 4.2.12. Despite the Hamiltonian Path problem of the example being complete, we can add more supporting rules ensuring, for example, that a property is really satisfied. For the inclusion constraint we add a constraint ensuring that all the elements of **`inPath`** must be a subset of **`edge`**; to this end we write *Inclusion*, type CTRL+SPACE and use the popup to specify the predicate **`inPath`** as subset of **`edge`** and selecting all the attributes of both predicates; the constraint :– **`inPath(Y1,Y2), not edge(Y1,Y2).`** will be written to the editor (fig. 4.17). □

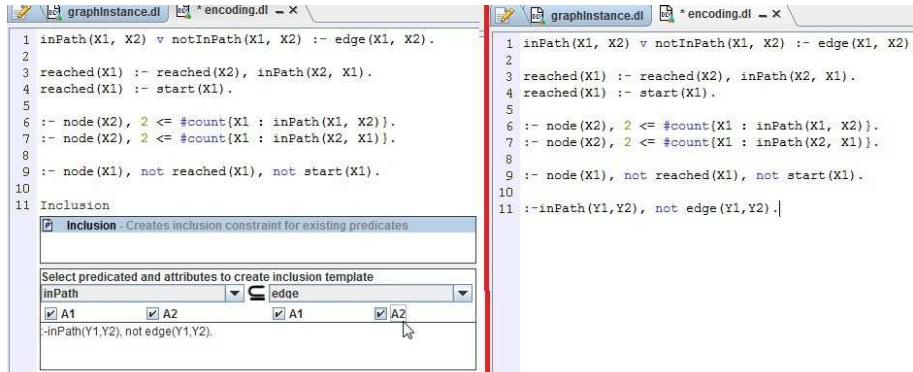


Figure 4.17: Code template for writing an inclusion constraint.

Numbers of elements constraints. The keywords *AtLeast* and *AtMost* allow one to set bounds on the number of elements contained in a predicate (projected to exactly one attribute). Using this code template the user can generate a constraint containing the aggregate `#count` with a predicate, and set an integer, representing the bound, as lower guard if the code template is *AtLeast*, and as upper guard if the code template is *AtMost*.

Example 4.2.13. Other supporting rules that we can add to the Hamiltonian Path encoding can be constraints that check whether the number of reached nodes is exactly the number of nodes of the graph (i.e. all the nodes of the graph must be effectively reached). On the program we suppose that the input graph is composed of exactly 5 nodes, so we add two constraints ensuring that the number of elements of the predicate `reached` must be neither lower than 5 nor greater than 5. For the upper case we use the keyword *AtMost* and we insert the predicate `reached` on the popup window of the code template specifying the value 5 for the attribute of `reached` (fig. 4.18). □

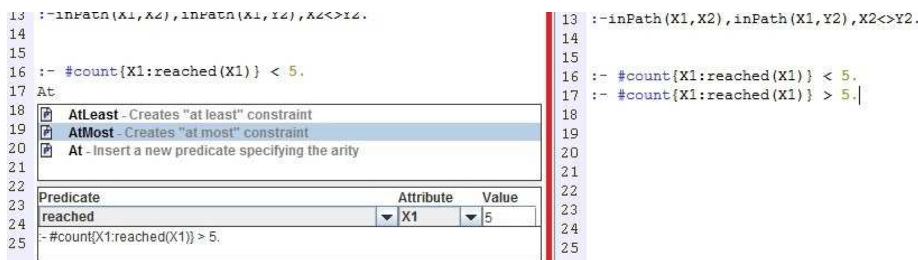


Figure 4.18: Code template for writing an *AtMost* constraint.

Transitive Closure. This code template, activable by exploiting the keywords *TC* and *TCC* is used for defining transitive closures on binary predicates. *TC* defines a classical transitive closure by exploiting one rule only that completes a predicate including other elements representing the transitive closure of the same predicate; for example, the rule `edge(X,Y) :- edge(X,Z),edge(Z,Y).` defines a classical transitive closure for the predicate `edge`. *TCC* defines a closed

transitive closure, represented by two rules that populate a new predicate with the transitive closure of the selected predicate; for example, rules define a closed transitive closure for the predicate `edge`:

```
edge_closed(X, Y) :- edge(X, Y).
edge_closed(X, Y) :- edge(X, Z), edge_closed(Z, Y).
```

To activate this code template (suppose the closed one) the user must write *TCCpredname* where *predname* indicates a binary predicate in which the transitive closure has to be applied.

Example 4.2.14. We want to use the code template in our Hamiltonian Path program. Suppose we want to verify the effective reachability of all the nodes of the graph. To this end we use the keyword *TCCedge*, to activate the code template defining the transitive closure for the predicate `edge`. We confirm and the rules of the transitive closure are inserted (fig. 4.19). □

Figure 4.19: Code template for defining a transitive closure.

User-defined code templates

In the main window of *ASPIDE* there is a panel named *Template* that shows a list of other available code templates; they allow one to build other commonly used ASP patterns like *union*, *intersection*, *subset*, *permutation* and so on. Using the panel the user can both use them in the current file where he is currenctly working and create a new user-defined code template that works how he prefers.

The creation process of a new code template consists in writing a DLT [17] file. DLT introduces the concept of *template predicate* that can be seen as a way to define intensional predicates by means of a sub-program.

The following example:

```
#template path{p(1), h(2)}(2){
  path(X, Y) :- p(X), p(Y), h(X, Y).
  path(X, Y) :- path(X, Z), h(Z, Y).
}
```

introduces a generic template program, defining the predicate `path`, intended to compute the transitive closure of a generic binary predicate `h` where the domain of `h` is composed by the elements of the generic unary predicate `p`. The

template can be seen as a graph reachability program where the predicate `h` represents edges, the predicate `p` represents nodes and `path` contains a pair of nodes that are reachable between each other. The predicates `p` and `h` contained in the header of the DLT template are signatures of input predicate that can be passed to the template for the transitive closure computation.

The mentioned code templates can be used in *ASPIDE* by exploiting a window that guides the user to specify input predicates. When the user uses this kind of template in a program, DLT is called with a particular option and the result of DLT will be a sub-program obtained by substituting the predicates of the input predicate signatures with the predicates passed as input; the resulting sub-program will be inserted in the editor.

Example 4.2.15. We want to use the code template `path` in our Hamiltonian Path program. Suppose we want to verify (using this different method of the one proposed before) the effective reachability of all the nodes of the graph; in this case the input predicate of `p` is represented by the unary predicate `node` and the input predicate of `h` is represented by the binary predicate `edge`. We select the template `path` from the templates panel and click on the *Insert Template* button (fig. 4.20). A dialog window is opened where the user can specify the input predicates of the program and ask for a preview of the sub-program obtained from DLT. We click on *OK*, and the obtained sub-program is inserted in the editor (fig. 4.21). □

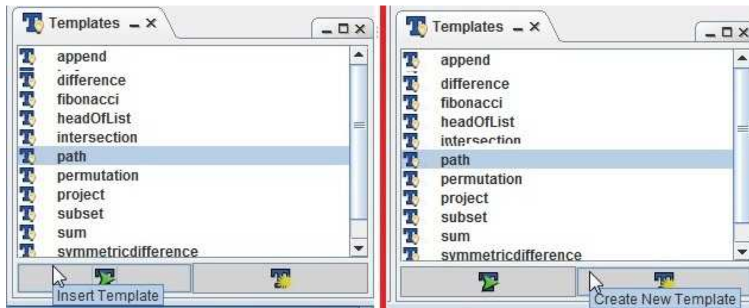


Figure 4.20: Template creation and definition.

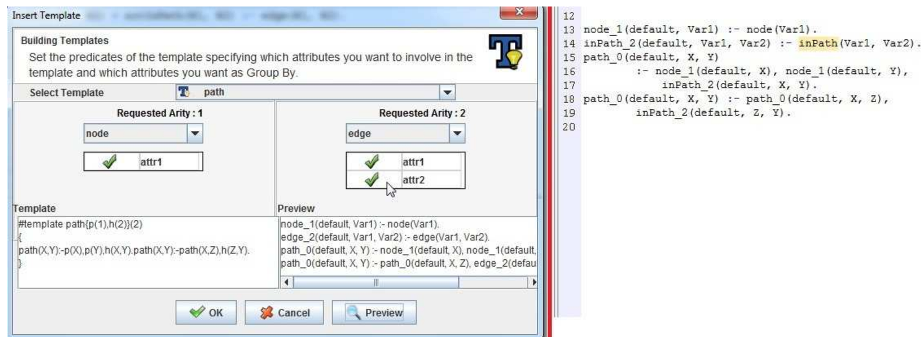


Figure 4.21: Specify the input predicates of the template `path`.

New user-defined templates can be created in *ASPIDE*. To create a new template we click on the *Create New Template* button of the templates panel (fig. 4.20); a dialog will be opened where the user can write the new template in accordance with the DLT files syntax (fig. 4.22). The new template will be saved and inserted in the templates panel.

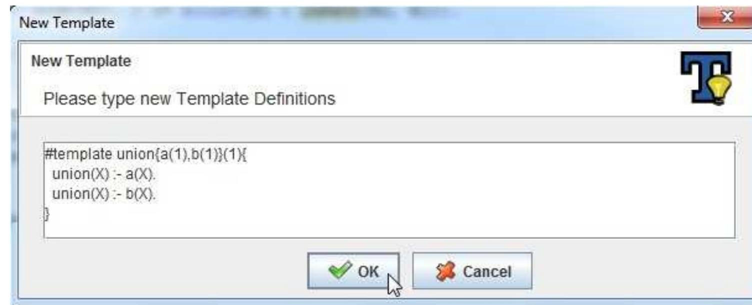


Figure 4.22: Create a new template.

4.2.5 Annotation management for ASP programs

Meta-information is used in many programming languages to give more expressiveness to them for both programmers and compilers. For example, Java uses annotations to add meta-information to classes, methods, functions and so on. *ASPIDE* offers also annotations for ASP programs for indicating rule names, specify predicate schemas (name, arity, optional data-type), handling database connectivity and so on. Each annotation in *ASPIDE* is included within a comment (so that the solvers can ignore it) and starts by "@". Meta-information given through annotations is exploited by the IDE to enrich the information displayed in the outlines and to provide the user with smarter editing facilities (such as auto-completion, test case composition, etc.).

In the following, the list of annotations introduced in *ASPIDE* are described.

Simple ASP annotaions

Simple ASP annotations, described in the following, allow simple meta information to be specified like rule names and comments:

- `@name = ruleName`: by writing this kind of annotation before a rule, the name *ruleName* is assigned to the rule;
- `@start-program-comment` ... `@end-program-comment`: all comments that are written between these two keywords represent comments to the entire program;
- `@start-comment` ... `@end-comment`: all comments that are written between these two keywords refer to rules that are also written between the keywords. This is a way to introduce explicitly comments to rules.

Example 4.2.16 (Rule name annotation). We can set the name *guess* to

the disjunctive rule of the Hamiltonian Path program in the following way:

```
%@name = guess
inPath(X1, X2) v notInPath(X1, X2) :- edge(X1, X2).
```

□

Example 4.2.17 (Program Comment annotation). We can comment the Hamiltonian Path program in this way:

```
%@start - program - comment
% The following program resolves the Hamiltonian Path problem
% that is a well known NP - complete problem
% in graph theory.
%@end - program - comment
```

□

Example 4.2.18 (Rule Comments annotation). To introduce comments to rules r_4 and r_5 of the Hamiltonian Path program, by explicitly telling *ASPIDE* that those comments refer to those rules we write:

```
%@start - comment
% Checking part : specify constraints on solution.
% Each vertex in the path must have
% at most one incoming and one outgoing edge.
 :- node(X2), 2 <= #count{X1 : inPath(X1, X2)}.
 :- node(X2), 2 <= #count{X1 : inPath(X2, X1)}.
%@end - comment
```

□

Note that exploiting *comments* annotations, the comments used for programs can be separated by the ones used for rules and the ones that are not explicitly associated with anything. Program comments information can be also exploited by the *Visual Editor* for *DLV files* (see Chapter 5).

Schema and Database oriented annotations

ASPIDE introduces also annotations for specifying predicates schema and *TYP file* directives [88] on *DLV files*. The most relevant annotation is `@schema` which allows to specify schemas on predicates like attributes names and datatypes. This annotation, together with annotations for *TYP file* directives, are useful for advanced schema management and interaction with databases, and are described in details in Chapter 8.

4.2.6 Outline navigation

ASPIDE creates a graphic outline of programs, *TYP files* and *TEST files*, and it represents language statements. Outlines can be introduced also in *PLUGIN*

files; in this case it is managed by the *ASPIDE* plugin associated with the files (see Chapter 7 for more details).

Regarding *TYP files* and *TEST files* the outlines allow one to see a tree representation of elements contained in files. The figure 4.23 shows *TYP file* and *TEST file* outline examples; the *TEST file* outline shows a list of test cases contained in the file (see Chapter 6).

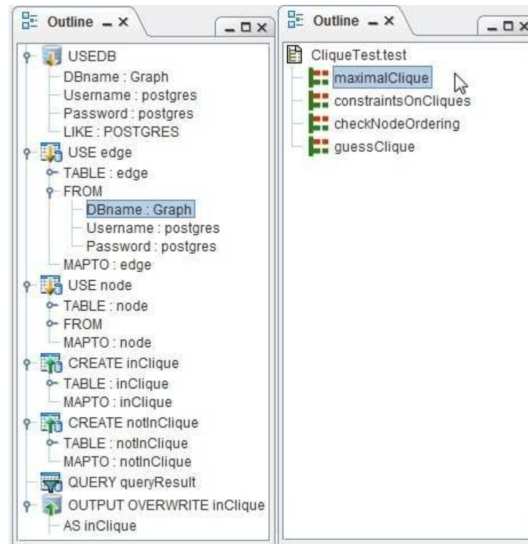


Figure 4.23: Outline for a *TYP file* on the left and for a *TEST file* on the right.

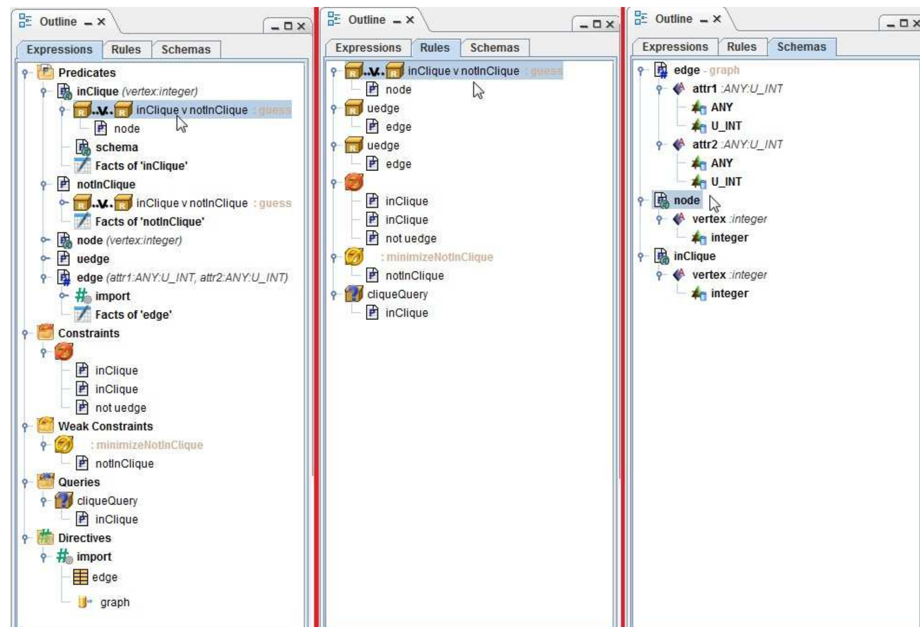


Figure 4.24: Different views for the outline representing a program.

Regarding programs, there are three kinds of outline (fig. 4.24) described in the following:

- *Rules*: it represents all the rules and constraints, contained in the program, using a tree view where in the first level of the tree there are head atoms, while the second level corresponds to bodies;

- *Expressions*: it classifies the elements of the program in groups. The first group is named *Predicates* and consists in organizing rules by predicates, showing all the rules that "define" a specific predicate (i.e. all the rules containing the predicate in the head) as children of the predicate. The groups *Constraints* and *Weak Constraints* contain, respectively, all the constraints and weak constraints of the program. The group *Queries* contains the queries of the program and, finally, the group *Directives* contains all the DLV directives (eg. `#import`, `#maxint`) defined to the program. Rules contained in groups are represented in the same way as the "Rules" based outline;

- *Schemas*: it shows, using a tree representation, all the schemas defined in the program in case the user has explicitly defined a schema or a `#import` directive. The outline shows attributes and datatypes and indicates whether a schema comes from the `#import` directive or not.

Each item in the outline can be used to access quickly the corresponding line of code (a very useful feature when dealing with long files), and also provides a graphical support for building rules in the graphical editor (see Chapter 5).

Example 4.2.19. Figure 4.25 shows the *Maximal Clique* program including an `#import` directive for the predicate `edge`, and schemas for the predicates `node` and `inClique`. The outline shows all the elements of the program based on the "Expressions" visualization. By exploiting icons, the user can easily identify predicates and:

- the rules that define them;

- more information indicating attributes and datatypes if the predicates have an associated schema or are associated with an external source like a database;

- names associated with rules and queries.

To access the corresponding line of the disjunctive rule quickly, we double-click on rule *guess* in the outline and the editor highlights the corresponding line (fig. 4.25). □

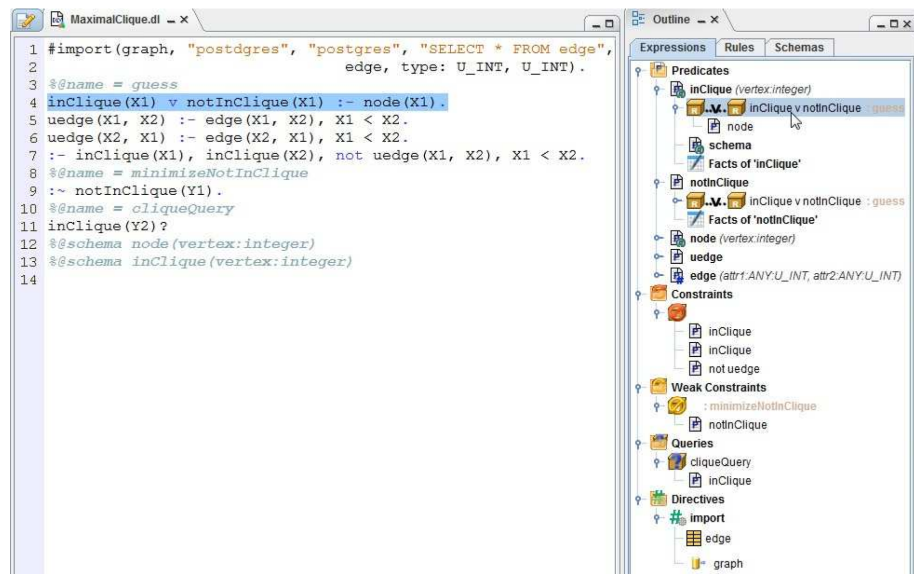


Figure 4.25: Outline for the Maximal Clique program and accessing to a line of code.

4.2.7 Errors and Warnings management

ASPIDE can detect errors and warnings in the files of the current workspace. When a workspace is initially loaded, files are parsed and syntax errors/warnings are shown in the console window of *ASPIDE*. Errors and warnings checking is done also when files are saved using the editors of *ASPIDE*. In the case where a file contains errors or warnings, the *Project* and *Workspace Explorer* panels highlight this file to signal the presence of errors to the user. Errors that *ASPIDE* can detect are described in the next Paragraphs.

Syntax errors. Syntax errors are detected when a file does not respect the relative syntax. In this case the *Error Console* indicates a description of the error and the position of it in the file. A syntax error is detected also in the case where a file is well-formed from the syntax-grammar point of view, but contains some construct that does not respect a property (e.g. safety).

Possible syntax errors are:

- *Safety Error*, detected when a rule is not safe;
- *Arity Error*, detected when a predicate defined in the program has a different arity compared to the same predicate previously defined in the same program. Arity errors can also be detected when a rule contains a predicate that has a different arity compared to the *schema* that was defined in the same file.

Warnings. Warnings represent possible errors a user might have made which are not considered, in general, as real errors from the point of view of the solver and do not need to be highlighted in *ASPIDE* as errors (they are highlighted in

a different way). For example, if a file contains a predicate that has a different arity compared to the same predicate used in another file of the same project, it is a warning; it is in reality considered an error if the user chooses to execute the two files together in the same solver calling. The possible warnings for *DLV files* are:

- *No definition for predicates used in body*, detected when a predicate is used in the body of a rule, but it was never defined in the files of the same project, neither by specifying it in the head of another rule, nor as predicate imported from an external source;
- *Arity Error in different files*, detected when a predicate used in a file has a different arity compared to the same predicate used in another file of the same project.

By exploiting the error console, an error (warning) can be opened by a double-click on the *Error Console*; in this case *ASPIDE* opens the default editor (if an editor of the file is not already open) and moves the cursor to the position of the error (warning).

Example 4.2.20. Suppose that on the *Hamiltonian Path* problem encoding we have the following rule:

$$:- \texttt{node}(\texttt{X1}), \texttt{not reached}(\texttt{X1}), \texttt{not start}(\texttt{X}).$$

It has a safety error on the variable `X` of the atom `start(X)`; the error console signals the error. By a double-click on the error, *ASPIDE* opens the editor evidencing the line containing the error (fig. 4.26). □

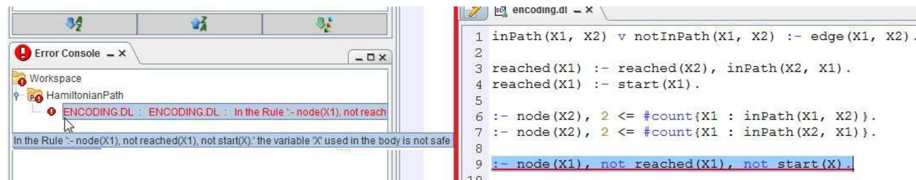


Figure 4.26: Select the rule with the safety error.

Other errors and warnings concern schemas definitions. For example, if a used predicate has a different arity compared to its respective schema, the error is signaled. Moreover, a user might also specify a schema by the `@schema` annotation and a *USE* directive of a *TYP file* that refers to the same predicate. In this case, if the two schema definitions are different (different attributes names or different arity) errors and warnings are signaled.

4.2.8 Dynamic code checking and errors highlighting

Programs are parsed while writing, and arity and safety errors or possible warnings are immediately outlined without the need to save files. In particular, syntax errors as well as mismatching predicate arities and safety conditions are checked. Note that, the checker considers the entire project for errors and warnings, indicating e.g., that atoms with the same predicate name have different

arity in several files. This condition is usually revealed only when programs divided into multiple files are run together. It is applied to any error condition, and a specific support was introduced also for checking some annotations.

Example 4.2.21. The figure 4.27 shows the unsafe rule of Example 4.2.20 that is highlighted in red, and for the disjunctive rule, we are supposing that the user has wrongly written the predicate `edge` as `edg` that is never defined to any file; consequently *ASPIDE* has highlighted the rule (in yellow). □

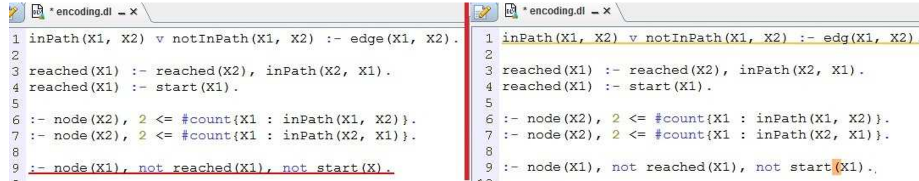


Figure 4.27: Error and Warning highlighting.

4.2.9 Quick fix

The system suggests quick fixes to reported errors or warnings, and applies them (on request) by automatically changing the affected part of the code. This can be activated by clicking on the line of code which contains an error/warning and choosing the desired fix among several suggestions from a popup window, e.g., safety problems can be fixed by correcting variable names or by projecting "unsafe" variables through an auxiliary rule (which will be automatically added).

Example 4.2.22. Suppose we have the rule

:– `node(X1), not reached(X1), not start(X).`

it is clearly unsafe. To apply the quick fix that transforms the rule into

:– `node(X1), not reached(X1), not start(X1).`

we double-click on the portion of code containing the error and we apply the suggested quick fix; the wrong rule is substituted with the correct rule (fig. 4.28). □

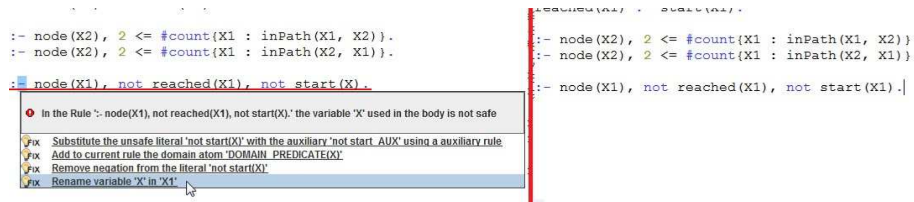


Figure 4.28: Applying a quick fix for a safety error.

For all errors and warning described in the previous Paragraph, there are a collection of quick fixes that can be applied; quick fixes can depend also on the

context and on the available predicates and variables. In the following, possible quick fixes for each error are described.

Fix Safety Errors. For safety errors, the following quick fixes can be applied:

- *Rename unsafe variable*: the system suggests changing the unsafe variable with another, also suggested, variable appearing in a positive literal of the body of the rule;
- *Remove negation from the unsafe literal*: in the case where the safety error is in a negated literal, the system suggests removing the keyword *not*;
- *Substitute the unsafe atom with an auxiliary atom*: in the case where the safety error is in a negated literal, the system suggests substituting the atom of the literal with an auxiliary atom where the unsafe attribute is removed. The old unsafe atom is inserted in the body of a new rule that defines the auxiliary atom;
- *Add to the unsafe rule a domain literal*: unsafe variables are not bound to any domain; the quick fix suggests adding a new positive domain literal that resolves the error by binding the unsafe variable to the new literal. The user can rename the domain literal;
- *Remove unsafe guard in aggregate atom*: in the case where the safety error is on a guard of an aggregate atom, the system suggests removing the guard.

Example 4.2.23 (Rename unsafe variable). The unsafe rule

```
:- node(X1), not reached(X1), not start(X).
```

is substituted by

```
:- node(X1), not reached(X1), not start(X1).
```

□

Example 4.2.24 (Remove negation from the unsafe literal). The unsafe rule

```
:- node(X1), not reached(X1), not start(X).
```

is substituted by

```
:- node(X1), not reached(X1), start(X).
```

□

Example 4.2.25 (Substitute the unsafe atom with an auxiliary atom). The unsafe rule

```
:- node(X1), not reached(X1), not start(X).
```

is substituted by the rules

$$:-\ \texttt{node(X1)},\ \texttt{not reached(X1)},\ \texttt{not start_AUX}.$$
$$\texttt{start_AUX} :-\ \texttt{start(X)}.$$

□

Example 4.2.26 (Add to the unsafe rule a domain literal). The unsafe rule

$$:-\ \texttt{node(X1)},\ \texttt{not reached(X1)}, \texttt{not start(X)}.$$

is substituted by the rule

$$:-\ \texttt{node(X1)},\ \texttt{not reached(X1)}, \texttt{not start(X)},\ \texttt{DOMAIN_PREDICATE(X)}.$$

The user can easily rename DOMAIN_PREDICATE with a significant name. □

Example 4.2.27 (Remove unsafe guard in aggregate atom). The unsafe rule

$$:-\ \texttt{node(X2)},\ 2 <= \#\texttt{count}\{\texttt{X1} : \texttt{inPath(X1,X2)}\} < \texttt{X}.$$

is substituted by the rule

$$:-\ \texttt{node(X2)},\ 2 <= \#\texttt{count}\{\texttt{X1} : \texttt{inPath(X1,X2)}\}.$$

□

Fix Arity Errors and Warnings. For arity errors and arity warnings, the following quick fixes can be applied:

- *Remove terms*: this quick fix is suggested in the case where a literal has more attributes than the original predicate. It allows one to remove excessive attributes;
- *Add a used variable*: this quick fix is suggested in the case where a literal has fewer attributes than the original predicate. It allows one to fill the literal with new variables, already used in the rule but never used at the same literal, until the arity error is fixed or no used variables are available;
- *Add underscores for other attributes*: this quick fix is suggested in the case where a literal has fewer attributes than the original predicate. It allows one to fill the literal with underscores until the arity error is fixed;
- *Add new variables*: this quick fix is suggested in the case where a literal has fewer attributes than the original predicate. It allows one to fill the literal with new never used variables until the arity error is fixed.

Example 4.2.28 (Remove terms). The rule

$$\texttt{reached(X1)} :-\ \texttt{reached(X2,1)},\ \texttt{inPath(X2,X1)}.$$

can be substituted by the rule

```
reached(X1) :- reached(X2), inPath(X2, X1).
```

or by the rule

```
reached(X1) :- reached(1), inPath(X2, X1).
```

□

Example 4.2.29 (Add a used variable). The rule

```
reached(X1) :- reached(X2), inPath(X2).
```

has a problem because `inPath` has arity 2 but in this rule it has arity 1; it is substituted by the rule

```
reached(X1) :- reached(X2), inPath(X2, X1).
```

□

Example 4.2.30 (Add underscores for other attributes). The rule

```
reached(X1) :- reached(X2), inPath(X2).
```

is substituted by the rule

```
reached(X1) :- reached(X2), inPath(X2, _).
```

□

Example 4.2.31 (Add new variables). The rule

```
reached(X1) :- reached(X2), inPath(X2).
```

is substituted by the rule

```
reached(X1) :- reached(X2), inPath(X2, Var_2).
```

□

Another possible arity warning arises in the case where a predicate used in a program has different arity compared to a schema defined for this predicate.

Fix undefined predicate warning. In the case where a predicate used in the body of some rule is not defined, the only quick fix proposed consists in renaming the predicate with another predicate already defined in the project.

Example 4.2.32. The rule

```
inPath(X1, X2) v notInPath(X1, X2) :- edg(X1, X2).
```

has an warning because the user has wrongly written `edg` and not `edge`; the quick fix can suggest a renaming with a defined predicate (in our case `edge`)

and the resulting rule will be

$$\texttt{inPath}(\texttt{X1},\texttt{X2})\ \texttt{v}\ \texttt{notInPath}(\texttt{X1},\texttt{X2}) :- \texttt{edge}(\texttt{X1},\texttt{X2}).$$

□

4.2.10 Dependency graph

The system provides a graphical representation of several variants of the (non-ground) dependency graphs associated with the project depending on whether both positive and negative dependencies are considered.

Also the graph of strongly connected components (playing an important role in the instantiation of the program) can be displayed.

Thus, *ASPIDE* offers three different visualizations of the dependency graph: *Complete*, *Positive* and *Strongly Connected Components* (fig. 4.29).

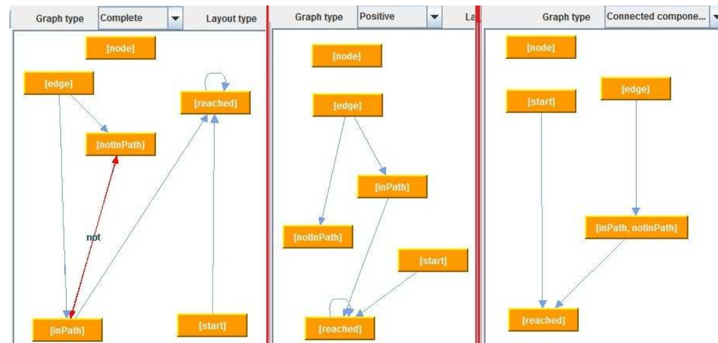


Figure 4.29: Complete, Positive and Connected Components Dependency Graphs of the Hamiltonian Path program.

4.2.11 Configuration of the execution

Files contained in the current workspace can be executed by exploiting a solver/system. *ASPIDE* provides a form for configuring and managing execution and allows one to set the solver/system executable, setup invocation options and input files, ask to perform a rewriting procedure before the execution and choose one of the possible views proposed for results visualization. The chosen options compose a *Run Configuration* that *ASPIDE* exploits on the execution phase; Run Configurations can be used by the user for configuring execution options of other features of *ASPIDE* like unit testing execution, debugging, querying and so on. Figure 4.30 shows the Run Configuration Dialog for executing the Hamiltonian Path program files.

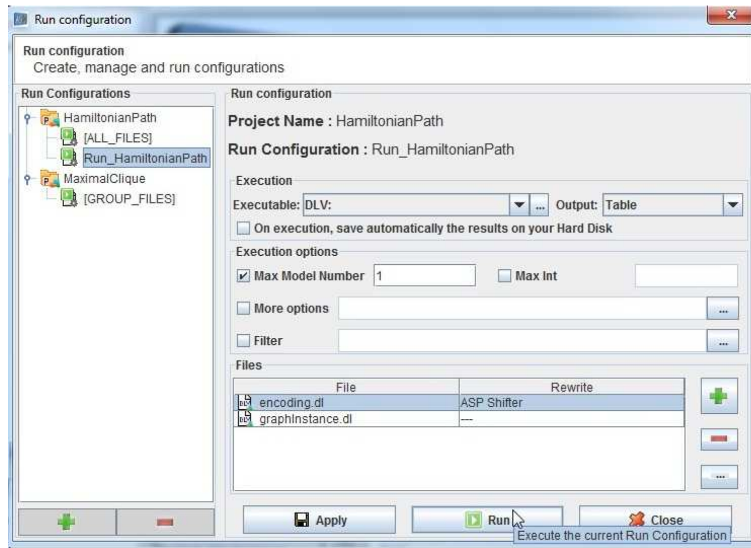


Figure 4.30: Run Configuration Dialog.

On the left, the dialog shows a tree view that organizes Run Configurations in projects and it can be used to create/rename/delete Run Configurations. By a double-click on a Run Configuration of the tree, the panel placed on the right shows all the properties of the Run Configuration. The user can choose the executable solver by setting a default one[1] or by selecting one from the file system. Regarding the *Output* property of the Run Configuration, the user can choose to visualize the execution results among a table representation, a query specific window, the tool *IDPDraw* [92] for graphical visualization of the answer sets, and a classical console window; moreover, the user can specify a user-defined plugin of *ASPIDE* for managing the results in a different way (see Chapter 7). Other properties that can be set on a Run Configuration, are solver specific options like "*Max Int*" and "*Max Model Number*". The last part of the Run Configuration dialog allows one to select the files to be executed; the user can select both project files, external files and, for each file, some rewriting procedure to be applied on single files before the execution (see Chapter 7). If no files are specified on the Run Configuration dialog, all the enabled files of the current project will be executed. By clicking on the button *Run*, the execution will start.

ASPIDE exploits also the concept of *Default Run Configuration*, which is a ready Run Configuration where the executable file is set to the default DLV solver and all the files of the project are selected by default. In this way a user currently working on a project, before creating any Run Configuration, can immediately run the project by clicking directly on the execution button (fig. 4.31).

[1] Default solvers are DLV, DLV +ODBC and DLV^{DB} and can be set using a properties window of *ASPIDE*.

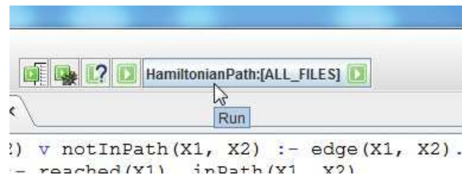


Figure 4.31: Run Button.

Note that the execution button shows the last Run Configuration that was executed for the current project; the Default Run Configuration is set to the button as long as the user does not execute a different one for the same project.

A Default Run Configuration can be exploited for quickly running single files without the need to create new Run Configurations; in particular the user can select, from the workspace explorer panel, a set of files and ask *ASPIDE* to execute them directly; in this case *ASPIDE* proposes, on a popup menu, possible ways for executing the files, like normal execution, querying or debugging (fig. 4.32).

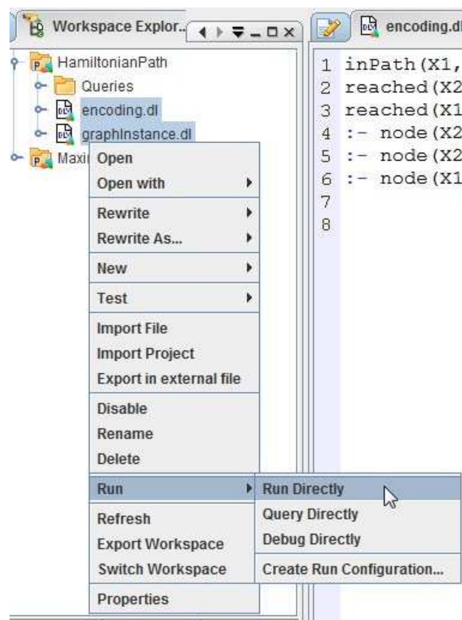


Figure 4.32: Quick Run of files.

This feature is particularly useful in the case where a user would like to test single files in the same project by avoiding the creation of new Run Configurations. Using the same procedure, the user can quicly create a new Run Configuration starting from the selected files.

ASPIDE introduces also a prototypical, workflow based way, to configure execution.

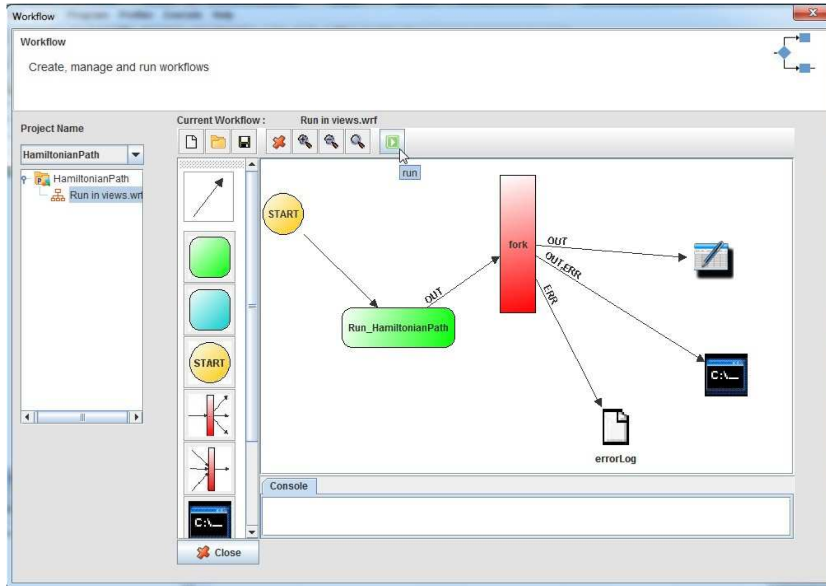


Figure 4.33: Workflow execution Editor.

In particular a user can exploit a graphical interface for building an execution process (fig. 4.33) consisting in combining several solver/system calls or piping results of a solver to:

- other solver(s) configured using different Run Configurations;
- an external executable system(s);
- a text file(s);
- the console or on the tabular results window of *ASPIDE*.

The user can easily decide combinations or piping and solvers calls; the entire workflow is transparently compiled in *perl* scripts for the execution. Figure 4.33 shows a workflow example on which the output obtained with the execution of the Run Configuration *Run_HamiltonianPath* is "forked" in three parallel flows:

- the first flow shows the output of the execution in *ASPIDE* using a tabular view;
- the second flow sends both standard output and possible errors of the execution to the console placed on the bottom of the panel;
- the third flow saves possible errors of the execution in a log file.

Using the Workflow Editor, the user can exploit also the tool *join* (opposed to *fork*) and the tool for setting a generic executable object in which the user can specify the path of the executable and execution options.

4.2.12 Presentation of the results

Execution results are presented to the user in different modalities:

1. in a comfortable view combining tabular representation of predicates and a tree-like representation of answer sets;
2. in a console window where the output of the solver is printed;
3. by exploiting the *IDPDraw* tool [92] for graphical visualization of results;
4. in a customized way by exploiting an *Output Plugin*;
5. in the Query Dialog in the case where the user wants to execute a query.

Regarding modality 1, *ASPIDE* wraps the results of the DLV solver and builds a tree representation of the answer sets. Figure 4.34 shows the results of the *Hamiltonian Path* program; on the left of the window there is a tree representing the list of generated answer sets and, for each one, a list of computed predicates are shown as children.

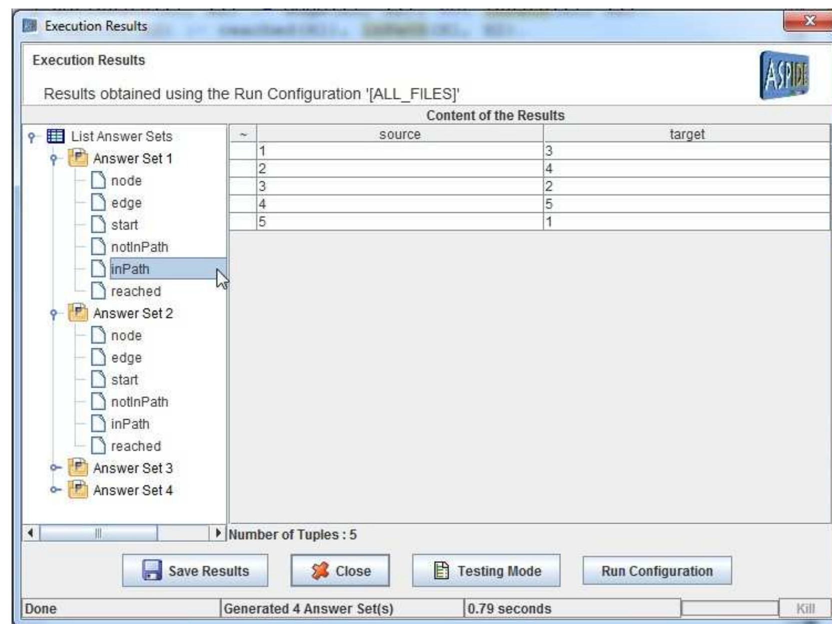


Figure 4.34: Tabular results for the *Hamiltonian Path* program.

By clicking on a predicate (e.g. `inPath`), on the right of the window a table is shown containing the set of computed tuples of the predicate. In the case where the program has associated a schema with the predicate, the results shows attributes names specified to the schema (in this case *source* and *target* for the predicate `inPath`). By exploiting the window, the user can choose to save the results in an external file, open the current Run Configuration directly, and expand the window in testing modality (see Chapter 6).

For modality 2, the results of the solver are simply printed to the results console of *ASPIDE* (fig. 4.35).

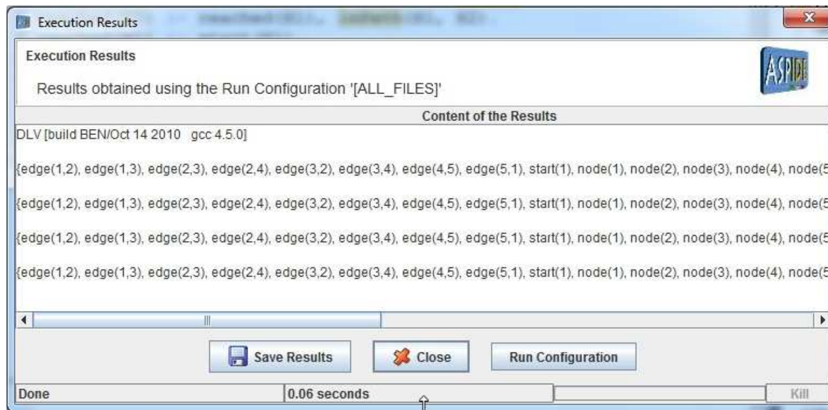


Figure 4.35: *Hamiltonian Path* program solver output printed to the Console.

Also in this case the user can save the results to an external file and can open the current Run Configuration directly.

Modality 3 allows the user to exploit the *IDPDraw* tool [92]; in this case a configuration program file that allows one to draw, in *IDPDraw*, the correct solution customized to the problem, must also be included to the Run Configuration under execution. In the execution phase, *ASPIDE* calls *IDPDraw* externally by passing to it the results of the execution, augumented with the special drawing atoms of the configuration program file. Figure 4.36 shows a result of the *Maze Generation* program described in Chapter 3; in this case the configuration program is customized for drawing mazes.

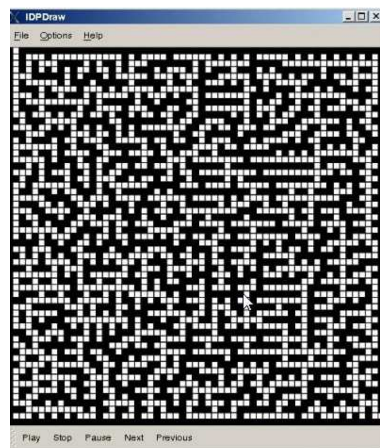


Figure 4.36: Graphical visualization of a Maze Generation result.

Modality 4 allows the user to exploit different ways to manage the results depending on an *Output Plugin* that he chooses for the execution; the user can also implement new user defined *Output Plugins*. For a detailed explanation of *Output Plugins* see Chapter 7.

Modality 5 allows the user to write queries in a confortable way by exploiting a Query Window. The Query Window can be open by exploiting the dedicated

button of the toolbar (fig. 4.37) and it works on a Run Configuration choosen by the user.

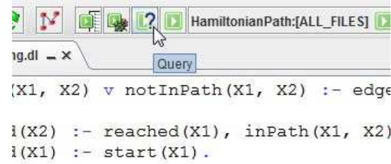


Figure 4.37: Open the Query Window.

For a quick query composition and execution, the window (fig. 4.38) offers a text field for writing a query, a button *Execute* for running the query, a button *Query Visual* for creating a query by exploiting the *Visual Editor* (see Chapter 5) and reasoning options (*brave* or *cautious*). By clicking on the button *Execute* the query is executed, the results are shown into the window and the written query is saved in a *Queries History* (fig. 4.38).

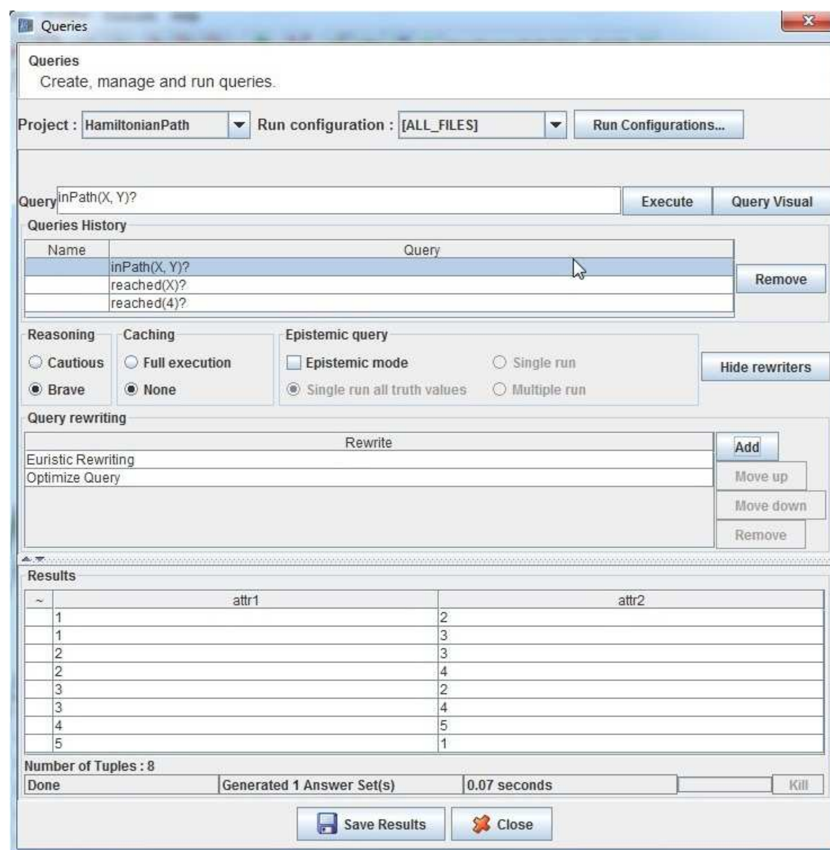


Figure 4.38: The Query Window.

The window offers also other options enabling the *Epistemic Mode* query execution proposed in [48] and revised with a new version in [49]. The idea of query execution in epistemic mode consists in affirming that an atom, which is neither *true* nor *strong negated* in an answer set, is considered *unknown* for the answer set.

Example 4.2.33. Consider the program

$$\texttt{p(a).}$$
$$\neg\texttt{p(b).}$$
$$\texttt{q(c).}$$

The answer set of the program is $\{\texttt{p(a)}, \neg\texttt{p(b)}, \texttt{q(c)}\}$. If we execute, in epistemic mode the query `p(a)?`, the answer will be *true*, for the query `p(b)?` the answer will be *false* and for the query `p(c)?` the answer will be *unknown*.

□

In case of multiple answer sets, the statement is valid by considering the union of all the answer sets for *brave* reasoning and the intersection of all the answer sets for *cautious* reasoning.

Example 4.2.34. Suppose now we have the following program:

```
a v ¬a.
c.
¬ b.
```

The answer sets of the program are $\{\texttt{a}, \texttt{c}, \neg\texttt{b}\}$ and $\{\neg\texttt{a}, \texttt{c}, \neg\texttt{b}\}$. The atom `a` is bravely *true* (there is an answer set in which it occurs), ¬`a` is bravely *true* (there is an answer set in which it occurs) and both `a` and ¬`a` are cautiously *false* (they do not occur in the intersection of all answer sets). If we execute the query `a?` in epistemic mode using the *cautious* reasoning option the results will be *unknown* since the intersection of the answer sets is $\{\texttt{c}, \neg\texttt{b}\}$.

□

The Query Window allows the user the choose different execution modalities for Epistemic Mode. In particular the modality *Single run* and *Single run all truth values* for unary queries allows the epistemic execution with one run of the solver by rewriting a query `q(X)?` in:

$$\texttt{query(X,true)} :- \texttt{q(X)}.$$
$$\texttt{query(X,false)} :- \neg\texttt{q(X)}.$$
$$\texttt{query(X,Y)}?$$

Regarding non-ground conjunctive queries, the modality *Single run* gets *true* results only, while the modality *Single run all truth values* gets *true* and *false* results. The modality *Multiple run* for a query `q(X)?` allows the epistemic execution by calling the solver twice; the first time with `q(X)?` and the second time with $\neg\texttt{q(X)}$? to get both *true* and *false* answers. The user will not notice any significant difference in use with the methods mentioned above, but in hard cases execution time may change.

The *caching* option allows one to save the results (oversimplifying the union/ intersection of the answer sets) in way that subsequent executions will not require an expensive evaluation. By running several queries on the same program, this option is expected to improve performance in some cases.

The window offers also a possibility of exploiting one or more query rewriters before the execution; in this way the user can, for example, apply an optimization before the execution. The user can create and/or exploit a *Query Rewriter Plugin* (see Chapter 7 for more details) that rewrites the query also as a function of the program files contained in the current Run Configuration. The window also allows the user to exploit, sequentially, more than one rewriter. By using the *Query rewriting* part of the window, query rewriters can be specified and the ordering of the rewriters can be easily changed. For example, in the Query Window shown in Figure 4.38, two query rewriters, namely *Euristic Rewriting* and *Optimize Query* were applied before the execution. The query `inPath(X, Y)?` was passed to the rewriter *Euristic Rewriting* and the resulting rewritten query, together with some supporting program also generated by the rewriter, were passed to the rewriter *Optimize Query*; finally the new resulting query was executed.

4.2.13 Debugger and Profiler

This Paragraph describes the integration/embedding of two external tools, *spock* and the DLV *Profiler* for debugging and profiling purposes. In ASP there are no standard methodologies for debugging programs, and tracing a solver execution can be a practical, but solver implementation dependent, way for debugging purposes.

Debugging ASP programs in *ASPIDE* with *spock*

The debugging interface of *ASPIDE* exploits the debugging methodology for ground programs proposed in [14], and implemented in the debugging tool *spock* [15]. The interface calls externally *spock* and wraps the results reported by the tool for presenting, in *ASPIDE*, debugging information about the program the user is currectly debugging.

The purpose of a debugger consists in explaining *why*, rather than *how*, a program is wrong. The basic idea of [14] is to show, given an interpretation, the rules that were *applicable* (i.e. the rules that are *true* for the given interpretation) and *blocked* (i.e. the rules that are *false* for the given interpretation). The solution proposed by the paper is ASP-based and was implemented in *spock* to detect applicable and blocked rules and to do more sophisticated analysis on any abnormal behavior.

Example 4.2.35. Consider the problem of inviting guests to a party and people would appear only if certain others do or do not attend the festivity. By considering that, the following program that tries to resolve the problem can be

carried out by knowing people preferences:

$$\mathtt{r}_1 : \mathtt{jim} :- \mathtt{uhura}.$$
$$\mathtt{r}_2 : \mathtt{jim} :- \mathtt{not\ chekov}.$$
$$\mathtt{r}_3 : \mathtt{uhura} :- \mathtt{chekov}, \mathtt{not\ scotty}.$$
$$\mathtt{r}_4 : \mathtt{chekov} :- \mathtt{not\ bones}.$$
$$\mathtt{r}_5 : \mathtt{bones} :- \mathtt{jim}.$$
$$\mathtt{r}_6 : \mathtt{scotty} :- \mathtt{not\ uhura}.$$

The program has the two answer sets {`chekov, scotty`} and {`bones, jim, scotty`}. Supposing that the user was expecting different results, the program is probably buggy. By exploiting the debugger tool we can see which rules have been applicable or blocked for the two answer sets. To this end we open the debugger by clicking on the *Debug* button of the toolbar (fig. 4.39). The debugging window is open by executing the program and showing, for each answer set, which rules were applicable or blocked; the Figure 4.40 shows applicable and blocked rules of the first answer set. □

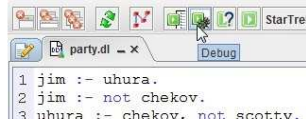


Figure 4.39: Debug the current program.

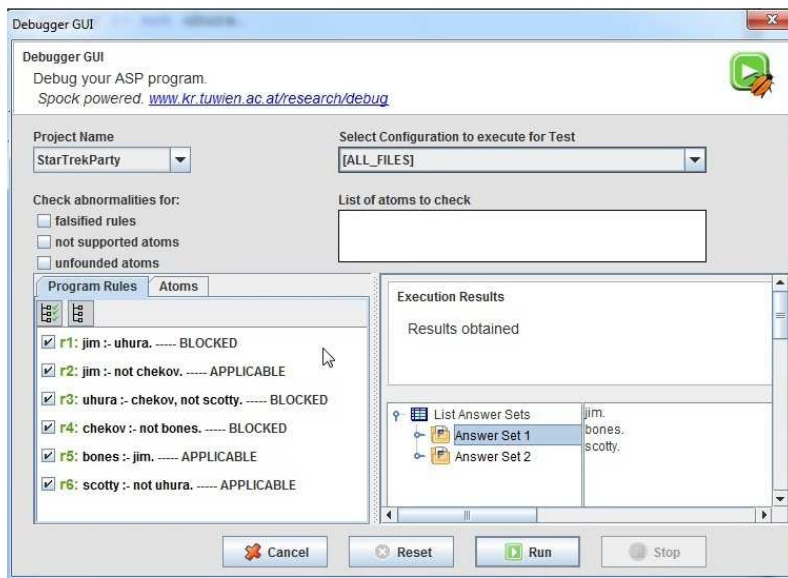


Figure 4.40: Applicable and blocked rules on the debugging process.

The basic idea just described analizes the program as a function of the real generated answer sets. However, the user would try to do more analysis by

considering incoherent situations like: "what happens to applicability of rules in case a different interpretation is considered" or "what happens in case an atom is considered true but it has no support in the program". To reply on these questions, we can exploit the user interface to check abnormalities related to:

- falsified rules;
- not supported atoms;
- unfounded atoms.

Those three abnormalities checking will be explained by presenting three examples on the *party* program of previous examples.

Example 4.2.36 (falsified rules). Suppose we would like to check abnormalities for rules defining `jim` by considering different possible intepretation scenarios for the two rules (i.e. what happens if `jim` is *true* and what happens if `jim` is *false*); we select rules r_1 and r_2 only and ask to the interface to check abormalities for falsified rules. *Spock* carries out that in the case where the interpretation {`chekov, uhura`} is considered, it would falsify, in the program, rule r_1 because it is applicable but `jim` is false (fig. 4.41). □

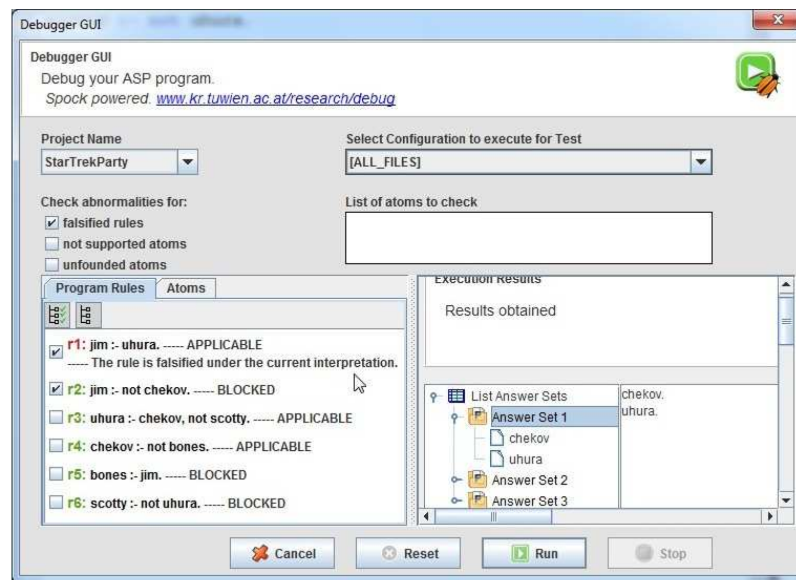


Figure 4.41: Check rule abnormalities for rules.

Example 4.2.37 (not supported atoms). Suppose now we would like to check supporting atoms abnormalities considering possible cases in which both `jim` and `chekov` are in the same answer set or not. We check the option *not supported atoms*, write the two atoms on the text area labelled *List of atoms to check* and select them in the *Atoms* tabbed pane. *Spock* carries out that considering the interpretation {`jim, chekov, bones, uhura`} where `jim` and `chekov` are together, `chekov` has no supporting rule in the program (fig. 4.42). □

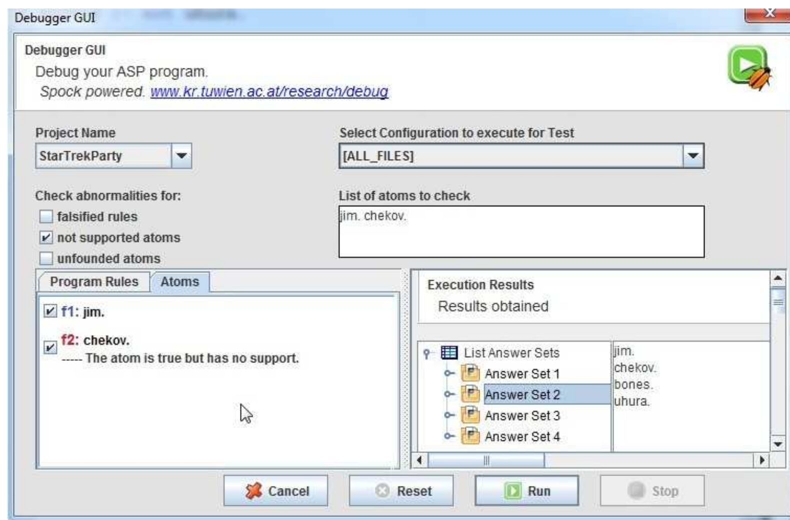


Figure 4.42: Check supporting atoms abnormalities.

Example 4.2.38 (unfounded atoms). Suppose now we would like to check unfounded atoms abnormalities considering possible cases in which both `uhura` and `scotty` are in the same answer set or not. We check the option *unfounded atoms* and write the two atoms in the same way as the previous example. *Spock* carries out that considering the interpretation {`jim, bones, uhura, scotty`} where `uhura` and `scotty` are together, both atoms are considered unfounded in the program (fig. 4.43). □

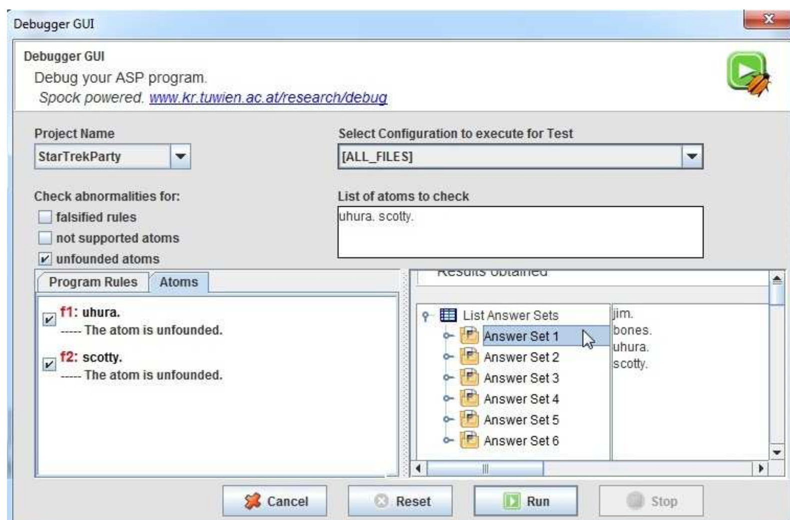


Figure 4.43: Check unfounded atoms abnormalities.

Profiling through the Visual Tracer for DLV

In software engineering, tracing is a specialized way for recording information about the execution of a program for debugging purposes or (depending on the type and detail of information provided by the tracing system) by experienced system developers to diagnose problems or optimize implementations.

In paper [20], a suitable solution to the problem of tracing the execution of an ASP system is presented by offering also an implementation into the ASP system DLV that features a graphical user interface and an on-line tracing method that sets it on the way between tracing and debugging. The graphical interface was fully embedded in *ASPIDE* by enabling the possibility of profiling the DLV solver inside the environment. The embedded GUI allows the user to exploit the full power of DLV features for tracing in a simple and intuitive way.

For using the GUI the user chooses to profile a program contained in a Run Configuration. To open the profiler, the user must click on the *Profiler* button of the toolbar (fig. 4.44) and the graphical interface is open (fig. 4.45); the system starts and the session is initialized.

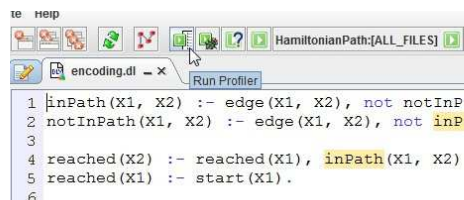


Figure 4.44: Run the *DLV Profiler* for the current Run Configuration.

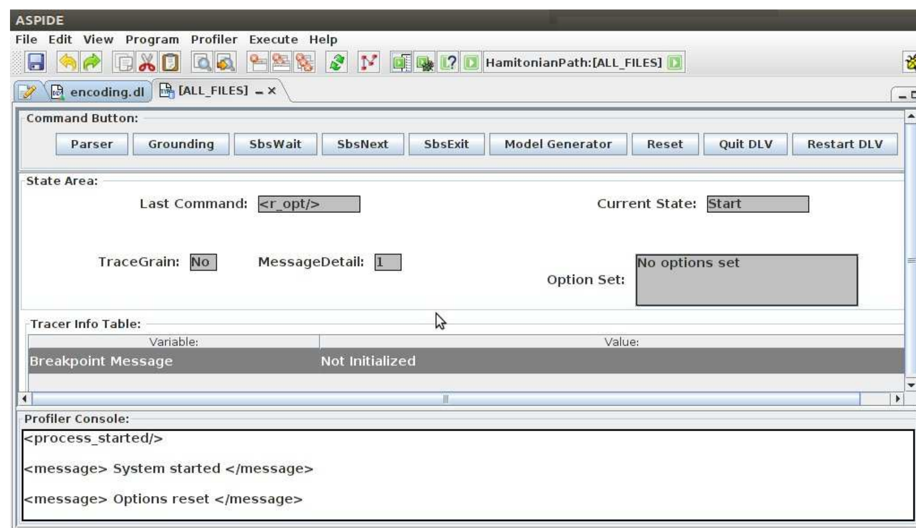


Figure 4.45: *DLV Profiler* GUI in *ASPIDE*.

The central area of the main window is divided into two parts: the main available commands and a tracing management area are placed in the upper side, while the lower consists of a *Console*. The tracing management area features

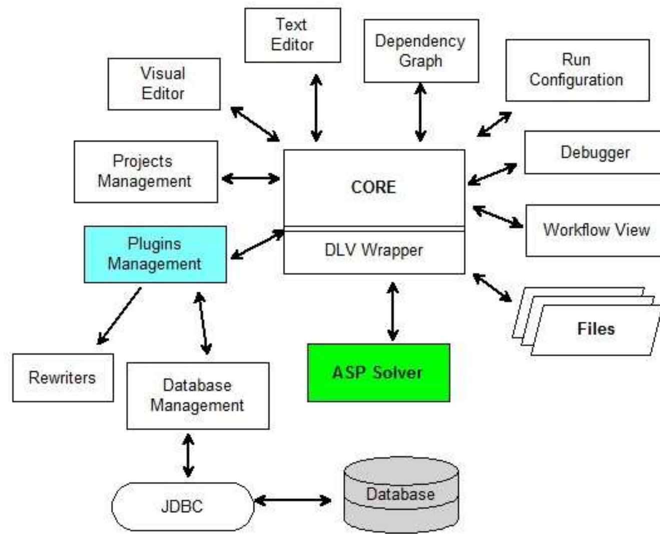


Figure 4.46: System architecture of *ASPIDE*.

some fields that remember the overall status of the application. Fields are in charge of showing: the current status of DLV, the last command given by the user, DLV options that are currently enabled, and the grain and detail levels set during the last tracing analysis. Under these fields a table, initially empty, is placed which show all the information printed by DLV during the session. The information printed at each breakpoint changes according to the detail level; however, the user can also customize the current configuration by means of appropriate buttons. Finally, a *Console* is shown in the lowest area of the window. This text-area shows the output of the DLV*Controller* [20] as it is released; thus, when a command is invoked, the user can view the results.

4.3 System Architecture and Implementation

ASPIDE is written in Java by following the Model View Controller (MVC) pattern. Figure 4.46 shows the general architecture of *ASPIDE*. In particular a *core* module manages, by means of suitable data structures, projects, files content, system status (e.g., error lists), and external components management (e.g., interaction with solver/debugger/profiler). For the execution, the user can exploit a Workflow module that combines several system/solver calls. The Text Editor module allows the user to write program in a textual way, while the Visual Editor module allows him to draw ASP programs by means of a set of graphical tools. A plugins manager module allows the system to interact with user-defined plugins that offer the possibility of introducing rewriters and to manage different input/output formats. The database management of *ASPIDE* is assigned to a user-defined plugin that can interact with the external databases by exploiting the *JDBC* library. Any update to the information managed by *ASPIDE* is obtained by invoking methods of the core, while, *view* modules (graphically implemented by interface panels) are notified by proper events in

case of changes in the system status. *ASPIDE* exploits:

- the JGraph[2] library in the *Visual Editor*, in the dependency graph and in the workflow modules;
- the DLV Wrapper [79] for interacting with DLV;
- JDBC libraries for database connectivity.

Debugging and profiling are implemented by wrapping the tool *spock* [15], and the DLV *profiler* [20], respectively. In the integration phase of the tool *spock*, a work on the adaptation of the tool for dealing with the syntax of the DLV system has been done.

[2] `http://www.jgraph.com/`

Chapter 5

Visual Editor for drawing logic programs

In general the main task of designing a logic program consists of writing text files (more or less computer-assisted). However, although the basic syntax of ASP is not particularly difficult, writing ASP programs might be uncomfortable for novices and error-prone users; moreover, programmers are often required to know the details of a specific ASP input dialect. This means that writing ASP programs is mainly an activity for ASP-experts (or, even worse, for experts in a specific ASP-system).

This Chapter presents a contribution for enabling users to write logic programs by exploiting a graphical way. In particular the Chapter describes the *Visual Editor* for writing ASP programs by exploiting a full graphical environment. It is integrated in *ASPIDE* and accessible by opening a *DLV File* in graphical modality; moreover, every time a user needs, a text/visual (and vice-versa) switching is possible thanks to a reverse-rengineering mechanism from text to graphical format. The *Visual Editor* allows the *drawing* of an ASP-program on the screen and the user is not required to edit files textually or know the details of a specific ASP dialect, since the fully graphic environment is inspired by QBE editors. As a consequence, the system should reduce the difficulty of producing ASP programs for both novice and inexperienced programmers, ease the encoding tasks for experts who prefer graphic tools, and reduce the probability of making syntax mistakes. Currently, the *Visual Editor* supports all the main advanced language features (i.e. disjunction, aggregates and constraints), and, since the core of the system is modular, it can be easily extended to support other language features and/or other input formats (e.g. *lparse/gringo* syntax [70, 86, 22]).

In the following, the *Visual Editor*, and the relative drawing methodology, are described with a short overview and a use case example. Note that, the system supports many different ways of creating modifying rules and constraints. The example reports only one of the possibile combination of commands and shortcuts that can be exploited for designing a program to solve the considered problem. Anyway, the described example is quite complete and helps in understanding how the *Visual Editor* effectively works.

5.1 Visual Editor overview

Figure 5.1 shows an overview of the *Visual Editor*.

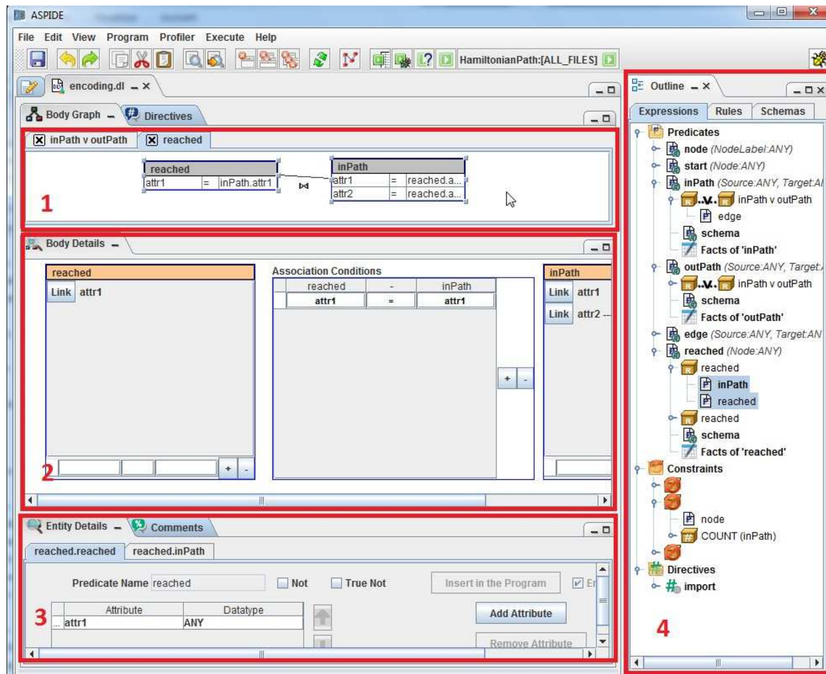


Figure 5.1: Overview of the *Visual Editor*.

In the Figure, zone 1 represents the *Body Graph* panel where the user can have a general view of the body of a rule. In particular, it is a graph representation of the literals inserted in the body of a rule and shows associations (join, union, intersection, not) between the body literals. Each literal represents an *entity* shown in the graph as a rectangular shape. Inside each entity there are also the join conditions of the respective attributes among:

- the attributes of the same entity;
- the attributes of this entity with other entities of the body;
- the attributes of this entity with atoms of the head.

By selecting the entities of the *Body Graph*, zone 2 shows their details; in particular we can see all the attributes belonging to the selected entities. We can also do operations like *Joins* between attributes of the body and binding (unbinding) of the attributes with one or more attributes of the head. We can set also join conditions like equality/inequality of attributes with other attributes or constants.

Zone 3 shows the details of single entities that we have selected. For each entity we can add, edit or delete attributes and set the datatypes. An entity can also be a rule, so we can use the same section to insert *static constant values* to the head of the rule, give a name to the rule, disable the rule, and so on. If the

selected entity is a literal of the body, the zone can show which attributes (if any) are not safe; we can also negate the literal of the rule by using the zone.

Zone 4 shows the outline of the program that we are building. We can view the program by *Predicates*, *Rules* or *Schema*[1]. In the views containing rules, we can do a mouse double-click on the rules to open them in the *Body Graph* for editing. To insert the facts to a predicate we have to use the *Predicates* view, and select the node *Facts of 'predicateName'* where *predicateName* is the name of the corresponding predicate. After the selection, the *Entity Details* panel shows a table where we can insert, edit and delete facts.

5.2 The Visual Editor by a Use Case Example

In this Section the *Visual Editor* is exploited for drawing the Hamiltonian Path program presented in Chapter 3. For the program, the following facts are considered:

```
node(1). node(2). node(3). node(4). node(5).
edge(1,2). edge(2,3). edge(3,4). edge(4,5).
edge(3,2). edge(1,3). edge(2,4). edge(5,1).
start(1).
```

Creating Predicates and Facts

Supposing we have already created an empty *DLV File* and opened it with the *Visual Editor*, we start by adding the input predicates `node`, `edge` and `start`. To add a predicate to the program, we click on the menu *Program* and select *New Predicate* (fig. 5.2).

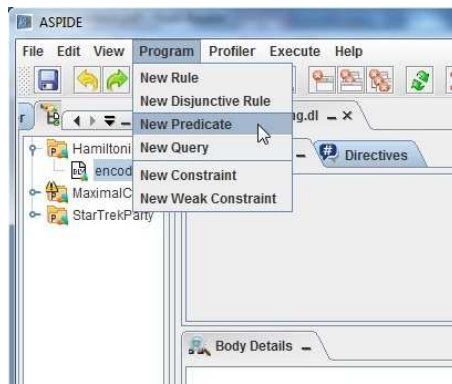


Figure 5.2: Creating a new Predicate.

A dialog will ask for the name of the predicate to be inserted, and after specifying the required information and confirming the command, a new predicate icon appears on the *Outline* panel (fig. 5.3).

[1]See Chapter 4.

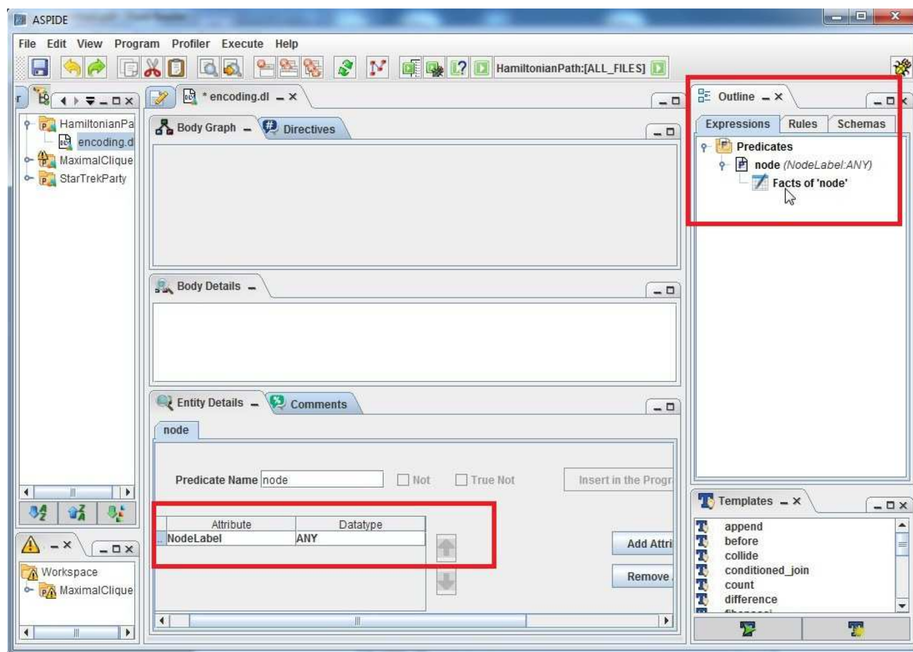


Figure 5.3: The predicate `node` in the *Outline* and the attribute *NodeLabel* inserted in `node`.

In order to simplify the program specification, the system allows one to specify a name for each attribute, and a datatype. This additional information is very useful during the editing phase, since it allows for rapidly identifying and joining attributes. To insert attributes, the user can exploit the panel placed in the bottom-center, labelled *Entity Details* (fig. 5.3), which shows the details of current predicate selected in the *Outline*. Note that in the *Outline*, near the new predicate, inserted attribute names and datatypes are shown (this visualization is done only in case the user has specified attribute names or datatypes). In the textual way it corresponds to specifying a `@schema` annotation[2] for that predicate. Similar steps have to be performed for adding the other input predicates `start` and `edge`.

In order to insert the facts to a predicate, e.g. `edge`, we select *Facts of 'edge'* on the *Outline* panel; the *Entity Details* panel will show a table where we can insert the facts (fig. 5.4).

[2]See Chapters 4 and 8.

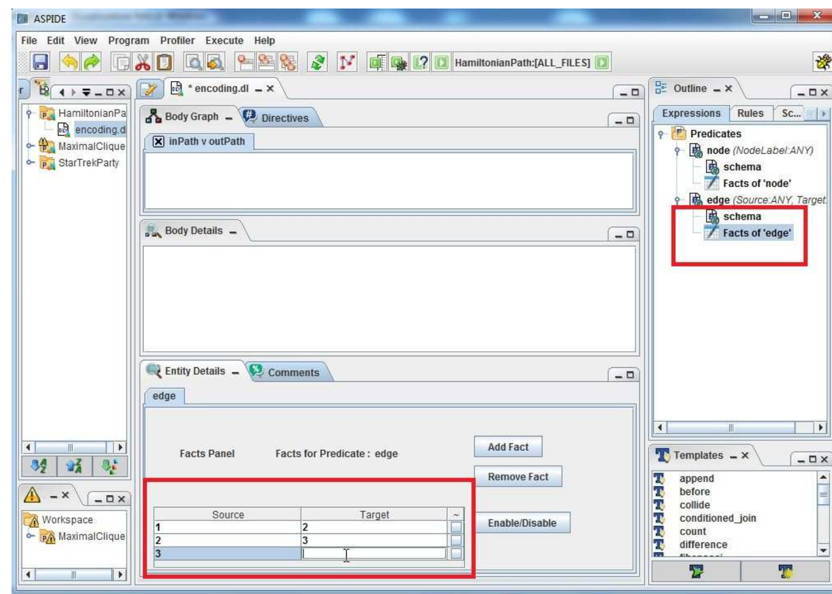


Figure 5.4: Insert facts in `edge`.

5.2.1 Drawing Rule r_1

We now insert the disjunctive rule r_1 by selecting *New Disjunctive Rule* on the menu *Program*. The system opens a dialog window used to specify the head of the rule (fig. 5.5).

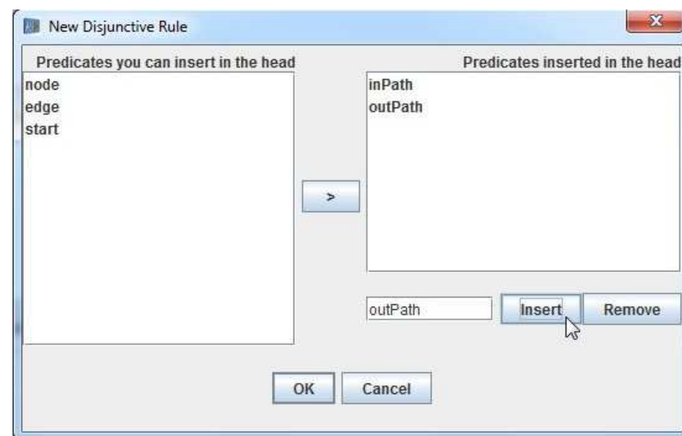


Figure 5.5: Dialog window for setting the head of the new disjunctive rule.

For inserting atoms in the head, the window allows the user to choose existing predicates and/or insert new predicates. We specify the new predicates `inPath` and `outPath` and click on the OK button; the new disjunctive rule will be inserted in the program and visualized at the *Visual Editor* both in the *Outline* panel and in the *Body Graph* panel situated at the top-center (fig. 5.6). Note

that in the *Outline*, showing a visualization by predicates, the disjunctive rule is inserted twice under both the predicates `inPath` and `outPath` (the rule defines both predicates). Then, one can specify the body of the disjunctive rule by dragging predicates from the *Outline* panel to the *Body Graph* panel. In our example we drag the predicate `edge` in the *Body Graph* of the disjunctive rule (fig. 5.6).

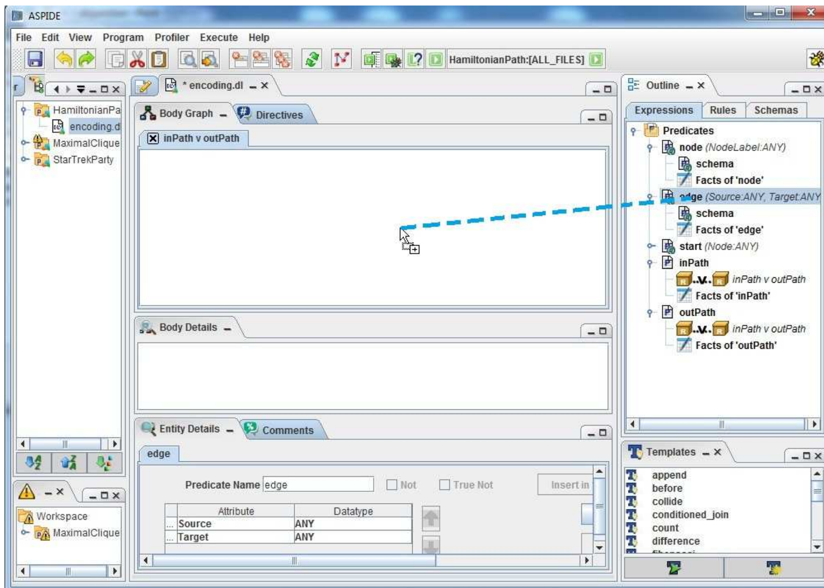


Figure 5.6: Dragging of the predicate `edge` from the *Outline* to the *Body Graph*.

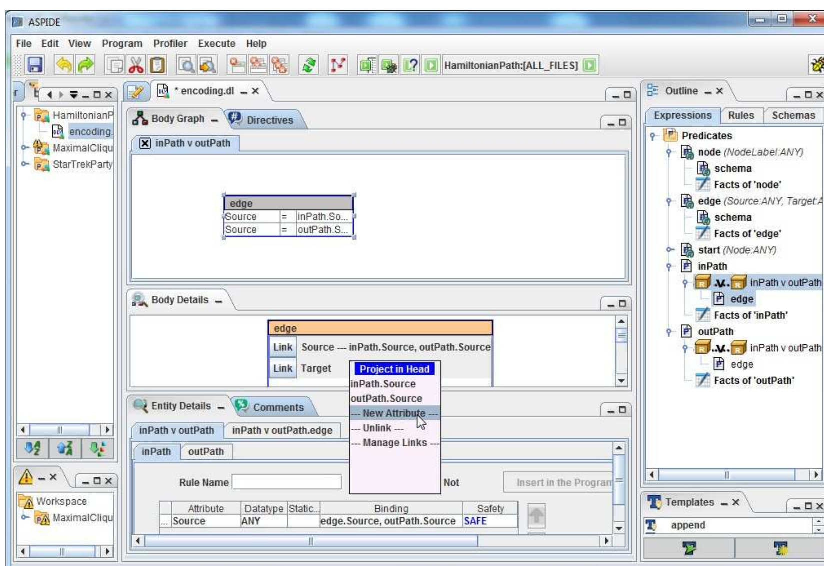


Figure 5.7: Projecting the attribute *Target* of the predicate `edge`.

A box representing the just added body predicate appears in the *Body Graph* (fig. 5.7). By selecting the box, the *Body Details* panel shows details of the predicate like the list of attributes and joins with other attributes (if defined). By clicking on the button *Link* placed near a particular attribute of the predicate `edge`, a pop-up menu allows for rapidly projecting the attribute with an attribute of the head (fig. 5.7). In this case the predicates `inPath` and `outPath` have no attributes, so we will select *New Attribute* on the pop-up menu and the system will create a new attribute both in `inPath` and `outPath`, simultaneously making projections. We do the same action also for the second attribute of edge (fig. 5.7). Finally, the *Body Graph*, the *Body Details* and *Entity Details* panels show information regarding the linking (projection).

5.2.2 Drawing Rules r_2 and r_3

The creation of rule r_2 is a bit more involved, since its body features two literals sharing variables. We insert a new rule by selecting *New Rule* on the menu *Program* and we name its head atom `reached`. Then we select the predicate `reached` and, using the *Entity Details* panel, we add a new attribute to the predicate naming it *Node*. Now we drag the predicates `inPath` and `reached` in the body panel of the new rule; note that this definition is recursive. Then, we join the two body literals by selecting both of them (this is done by clicking on the corresponding boxes in the body panel while pressing the shift key) and clicking on the join icon that appears near the selected boxes (fig. 5.8).

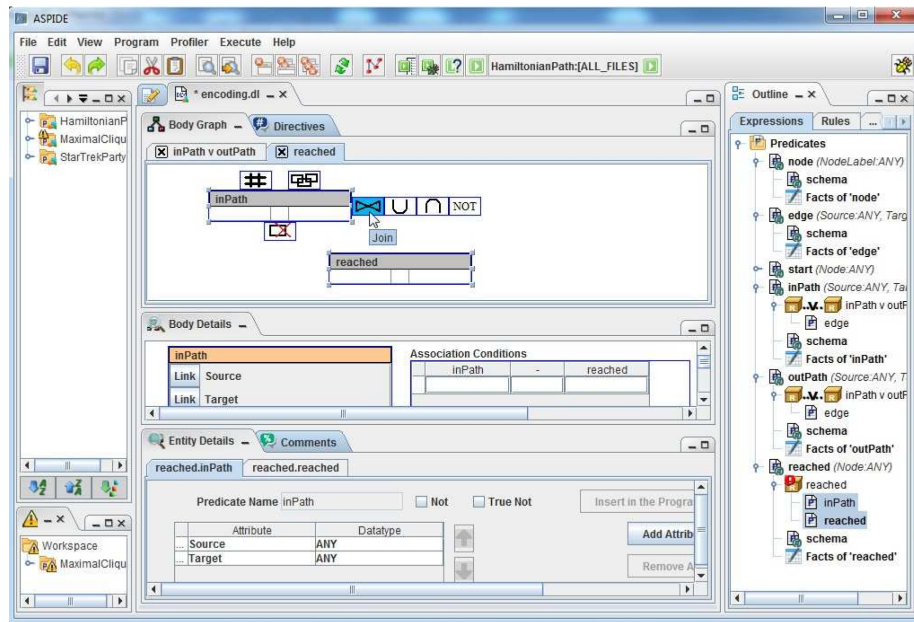


Figure 5.8: Join the predicates `inPath` e `reached`.

The system will show a dialog window where one can setup the join (fig. 5.9). The details of the join are reported in the *Body Details* panel (fig. 5.10), where the joined attributes can be further modified. As before, we properly link head

and body attributes, in this case by acting on the button *Link* placed near the second attribute of `inPath`.

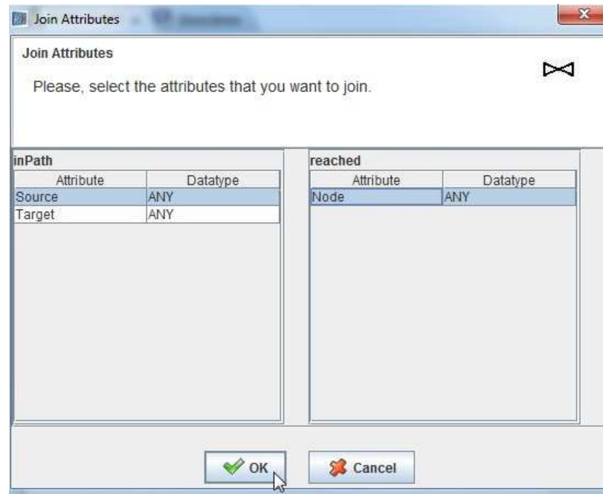


Figure 5.9: Join the attribute *Source* of the predicate `inPath` and *Node* of the predicate `reached`.

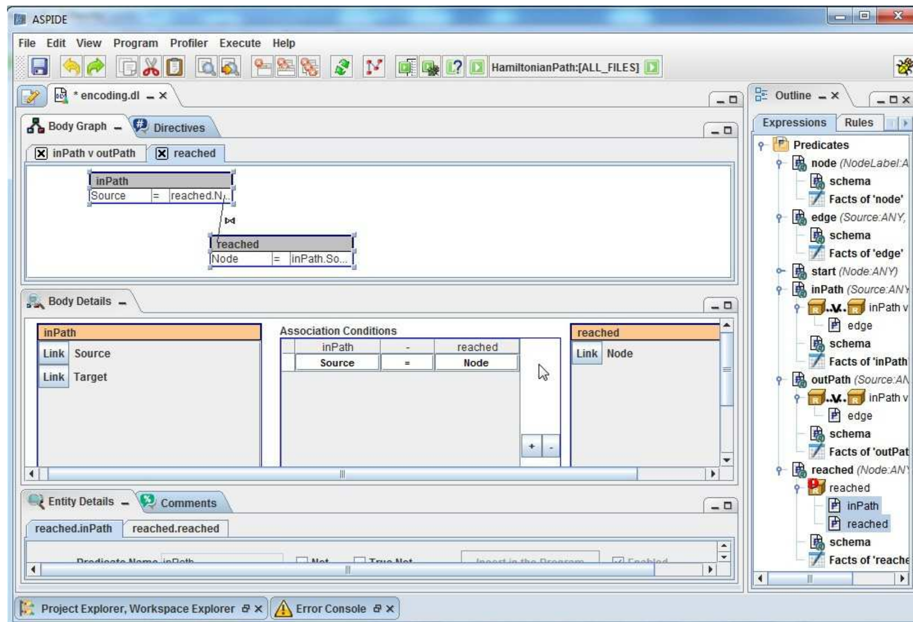


Figure 5.10: Details of the join.

Regarding rule r_3 we insert a new rule by selecting *New Rule* on the menu *Program* and drag the predicate `start` in the *Body Graph* (fig. 5.11).

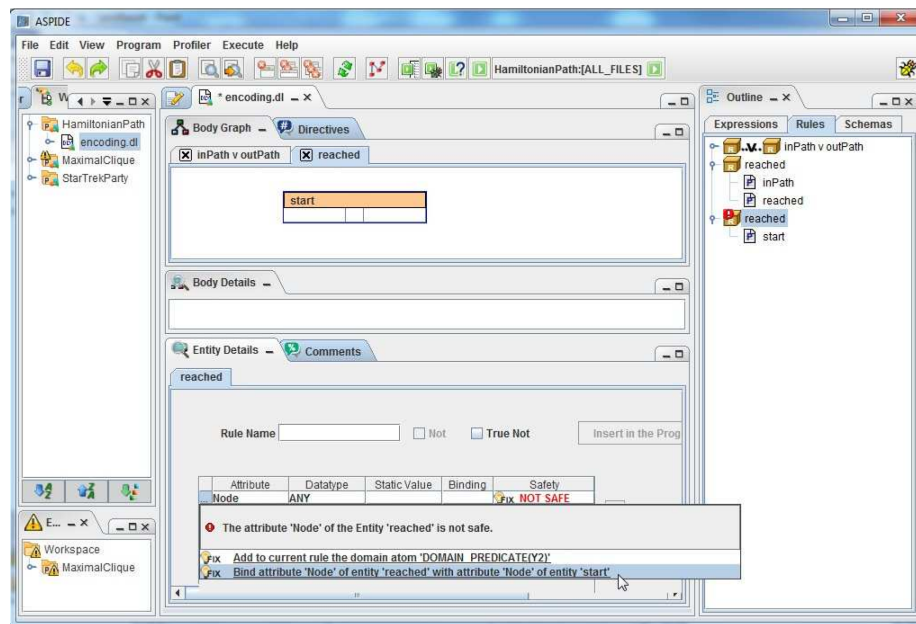


Figure 5.11: Applying a quick fix for a Safety Error.

Note that there is a *Safety Error* because the attribute *Node* of the predicate `reached` is still not bound. To resolve the problem we can apply a quick fix by double-clicking on the "lamp" on the *Entity Details* panel (fig. 5.11) and choosing to bind the attribute *Node* of `reached` with the one of `start`.

5.2.3 Drawing Constraint r_4

To create the constraint defined by rule r_4 we select *New Constraint* from the menu *Program*. The constraint, having a "forbidden" icon, appears on the *Outline* panel. We can specify a name to the constraint, in this case *OneOutgoingEdge*, by clicking on it and acting on the *Entity Details* panel (fig. 5.12). The body of constraints can be specified in the same way as before by dragging predicates from the *Outline* panel. In our case we drag `node` and `reached` and make a join between the two predicates (fig. 5.13).

To negate the literal `reached` of the body, we select that predicate in the *Body Graph* panel and, on the *Entity Details* panel, we tick the checkbox "Not"; in the *Body Graph* panel, the join arc between the two predicates will become an "arrow" from the literal `node` to the negated literal `reached`, the colour of the negated literal will become different and, in the *Outline* panel, the literal `reached` of the rule will have the keyword "not" (fig. 5.14).

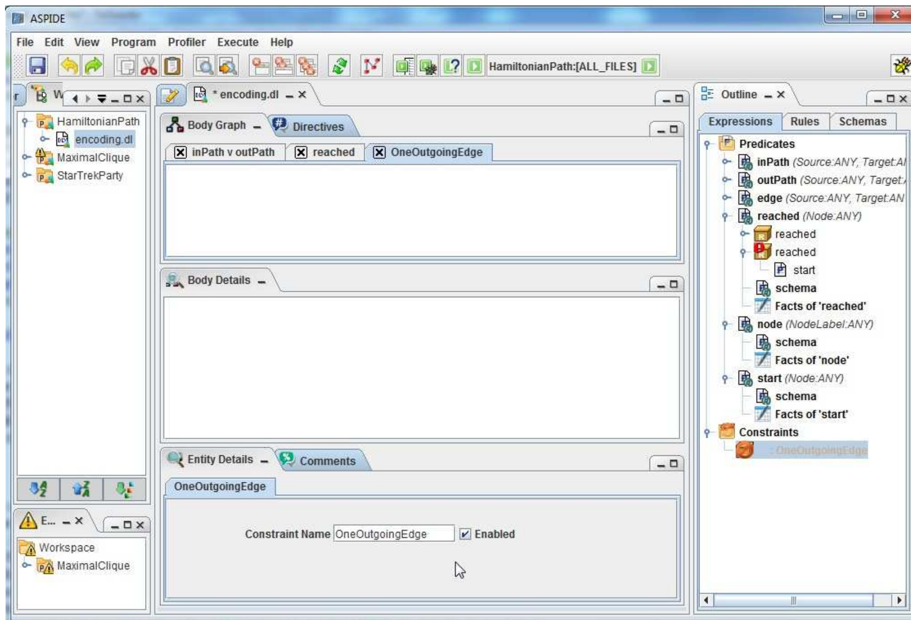


Figure 5.12: Specifying a name to the constraint.

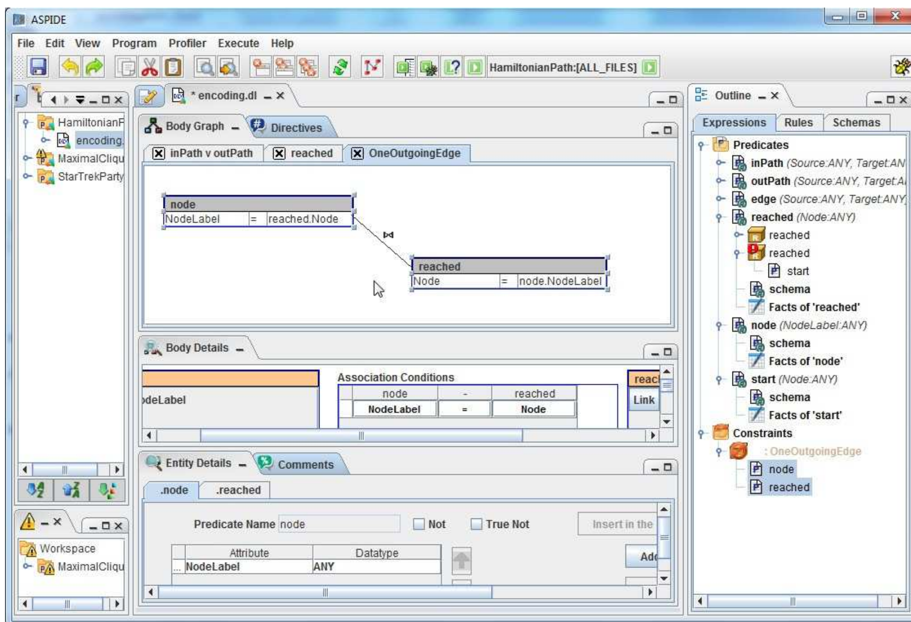


Figure 5.13: Created a new constraint and made a join between the predicates `node` and `reached` inserted in the body.

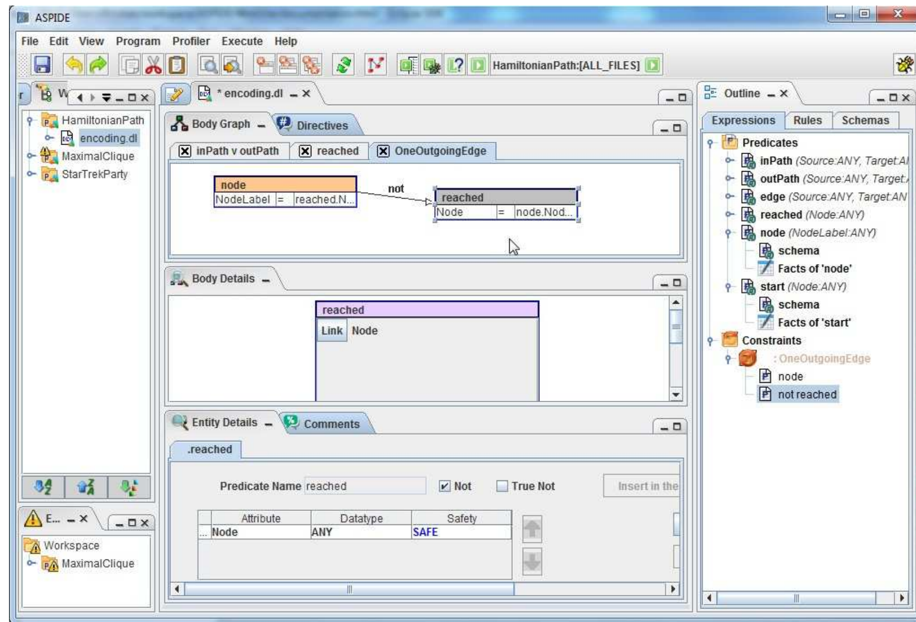


Figure 5.14: Negation of the predicate reached.

To also add the predicate `start` in the body, a quick way can be exploited allowing the user to bind a literal quickly with an other new negated literal. To this end we select the predicate `node` on the *Body Graph* panel and choose *NOT* from the popup window (fig. 5.15a); at this point we select `start` as target predicate of `node` (fig. 5.15b) and a "negated" binding will be quickly performed (fig. 5.15c).

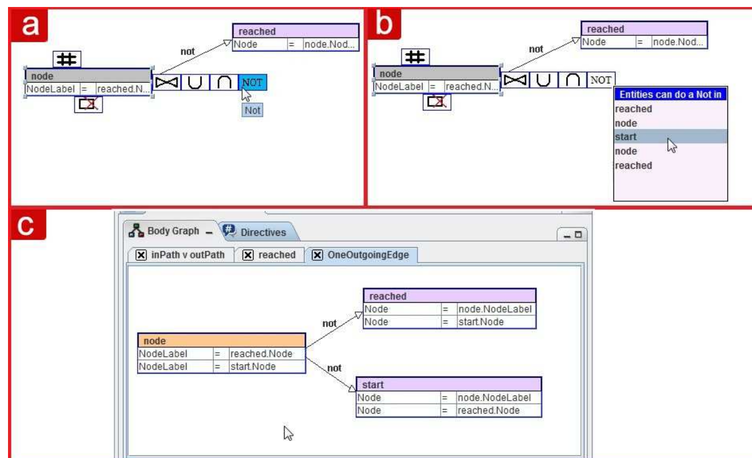


Figure 5.15: Creating a quick negation for the predicate `start`.

5.2.4 Drawing Constraint r_5 and r_6

For defining constraints r_5 and r_6 the procedure is the same as for previous rules. The only difference consists in the presence of built-ins atoms (see `Y <> Y1` and `X <> X1`). Consider rule r_5, the inequality is between the second attribute (*Target*) of the first literal `inPath` and the second one (*Target*) of the second literal `inPath`. To introduce the inequality we select the two literals and, by acting on the *Body Details* panel, we exploit a small panel placed between the two literals and insert the inequality between the attributes *Target* of the literals (fig. 5.16). In the same way we build rule r_6.

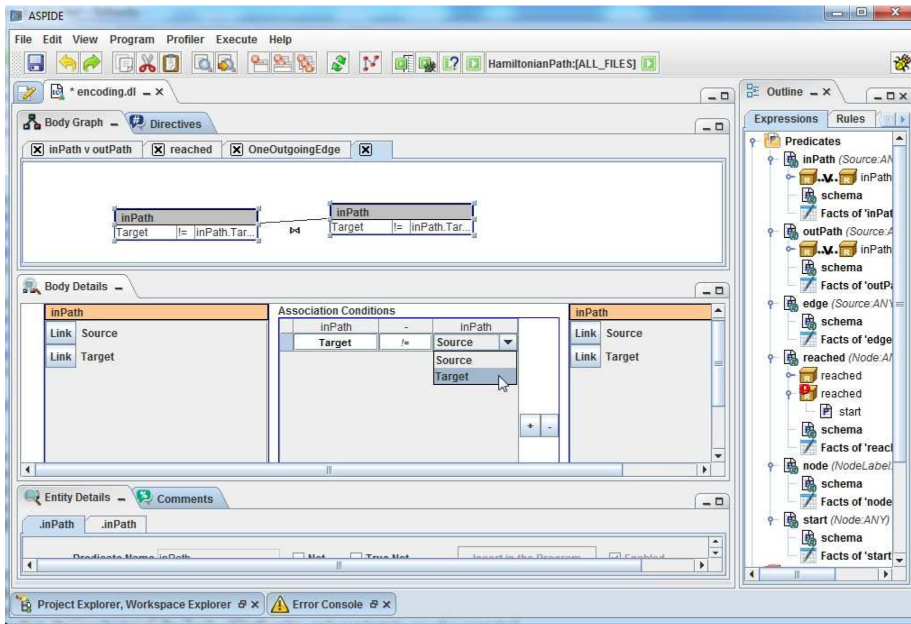


Figure 5.16: Defining an inequality for attributes.

At this point the entire graphical representation of our program solving the Hamiltonian Path problem was drawn. In order to show the specification of aggregates in the system, we modify the problem encoding by replacing constraints r_5 and r_6 with the following equivalent ones that use aggregates:

$$\mathtt{r_{5a}} : \ :- \mathtt{node}(\mathtt{X2}), 2 <= \#\mathtt{count}\{\mathtt{X1} : \mathtt{inPath}(\mathtt{X1}, \mathtt{X2})\}.$$
$$\mathtt{r_{6a}} : \ :- \mathtt{node}(\mathtt{X2}), 2 <= \#\mathtt{count}\{\mathtt{X1} : \mathtt{inPath}(\mathtt{X2}, \mathtt{X1})\}.$$

In order to remove r_5 and r_6 from the program, we right-click on their specification and select *Remove*. Then, we specify r_{5a} and r_{6a}. To aggregate the information contained in the predicate `inPath` we select the corresponding literal in the body of the constraint and, by a right-click, we click on *Aggregate* and select *Count* from a drop-down menu (fig. 5.17).

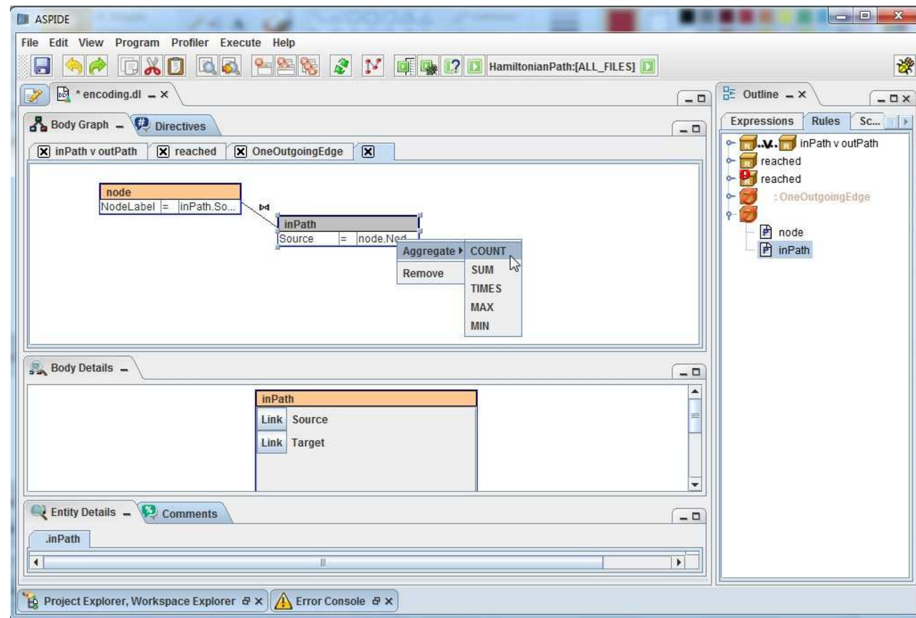


Figure 5.17: Aggregation of the predicate *inPath*.

In the *Outline* panel, the aggregate will be added, and in the *Body Graph* panel, a new entity with the aggregate will be created (fig. 5.18).

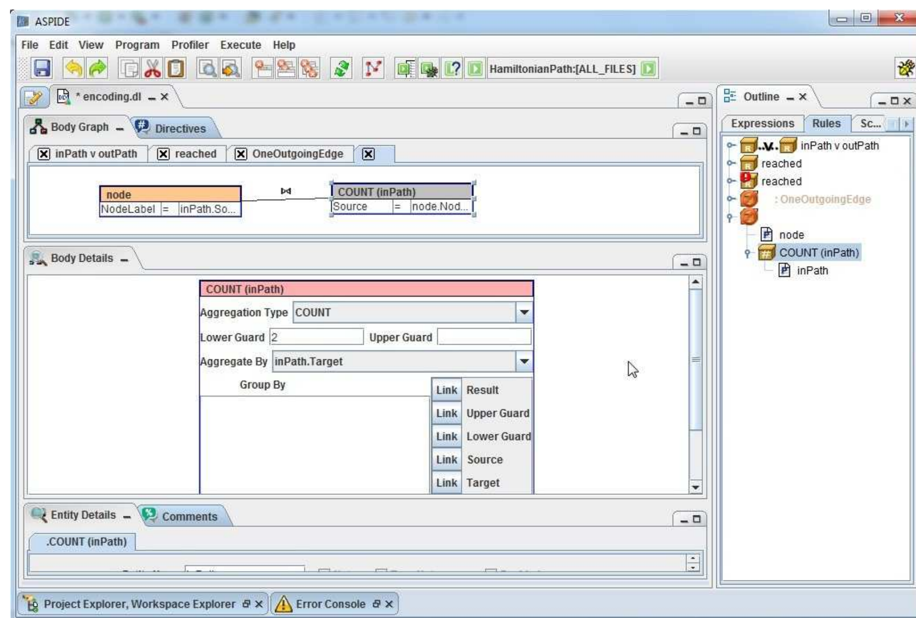


Figure 5.18: Result of the aggregation of the predicate `inPath`.

Note that, in the *Body Details* panel, an aggregate-specific box allows for setting guards, local variables, and so on. Finally, in a similar way we can add

constraint r_{6a}.

5.2.5 Switching from Visual Mode to Textual Mode

The program that we have just built graphically can be switched from visual mode to textual mode by clicking of the *Switch* icon available in the toolbar; in this way *ASPIDE* will show the program built in textual mode. (fig. 5.19).

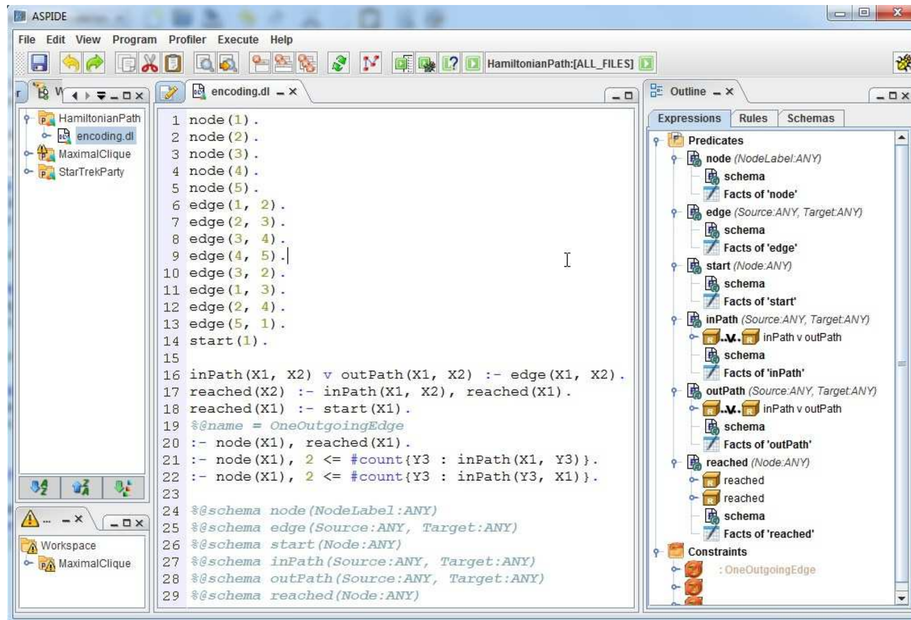


Figure 5.19: Result of the aggregation of the predicate `inPath`.

Note that `@schema` annotations are written to the text editor because we have specified attribute names on the predicates in the *Visual Editor*. This feature is very useful since we can exploit the *Visual Editor* for specifying predicate schemas in an intuitive way and, moreover, we can have an easy way to manage database tables mapped in predicates (see Chapter 8 for more details). At this point, if we switch again to the visual mode, schema annotations will be interpreted and (visual) predicates will be filled with the attributes specified in the annotations. Note that also the constraint name *OneOngoingEdge* of :– `node(X1), reached(X1).` was added as annotation.

5.2.6 Specifying DLV Directives and comments

DLV offers the possibility of exploiting directives for different purposes like setting the *maxint* value or mapping tables of external database. Suppose that now, in our example, instead of using facts for the predicate `edge` we want to import values contained in some external database table named `edge`, we do a right-click on the *Outline* panel and select *Import* from the menu *New Directive*. A new *Import* directive will be shown in the *Outline* (fig. 5.20a). By clicking on the tab *Directives* placed near the tab *Body Graph*, a panel is proposed where we can specify our directive (fig. 5.20b).

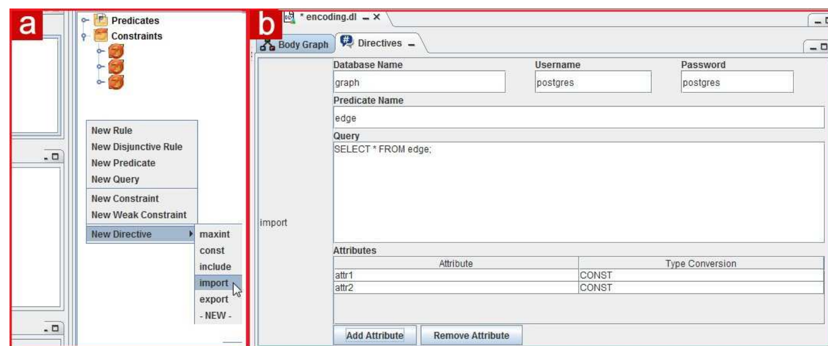


Figure 5.20: Specifying an *Import* Directive.

Chapter 4 describes the possibility of adding comments to programs by exploiting annotations. In particular, two annotations allow one to specify comments to the program and comments to some rules; the annotations are used by the *Visual Editor* to show graphically which are the comments of the program and which are comments of single rules. By clicking on the tab *Comments* placed near the tab *Entity Details*, program comments can be written (fig. 5.21a).

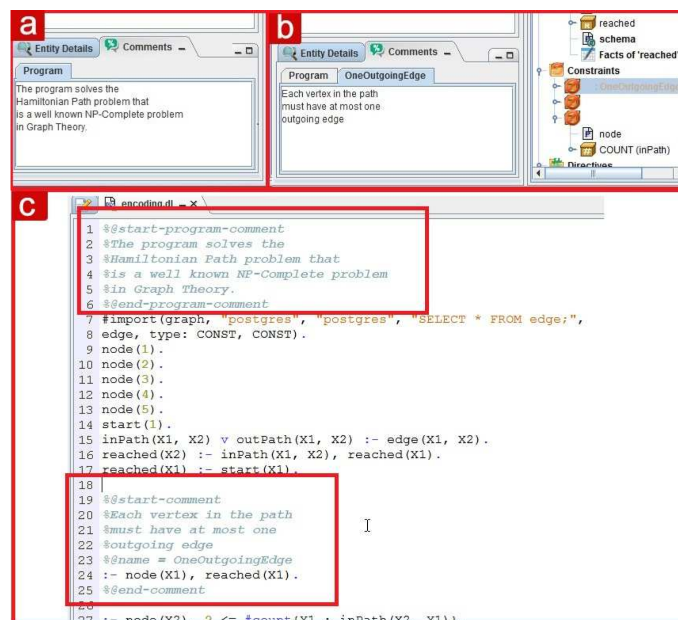


Figure 5.21: Specifying a comment to the *Visual Editor*.

Moreover, to add a comment to a rule, the same panel can be exploited; we select the constraint that we have named *OneOutgoingEdge*, and we write the comment in the opened panel that refers to the constraint (fig. 5.21b). If we switch to the text editor, annotations will be inserted containing the comments (fig. 5.21c).

5.2.7 Other features of the *Visual Editor*

The *Visual Editor* allows also to specify built-in predicates. For example the rule

$$\mathtt{a(X)} :- \mathtt{X} = \mathtt{Y}, \mathtt{Y} = \mathtt{Z} + \mathtt{3}, \mathtt{b(Z)}.$$

is represented in the *Visual Editor* by using special entities. In particular, `X = Y` is an entity and `Y = Z + 3` is an other entity; the two entities share the variable `Y` so a join between the entities in defined as we have previous seen. Figure 5.22 shows the graphical representation of the rule.

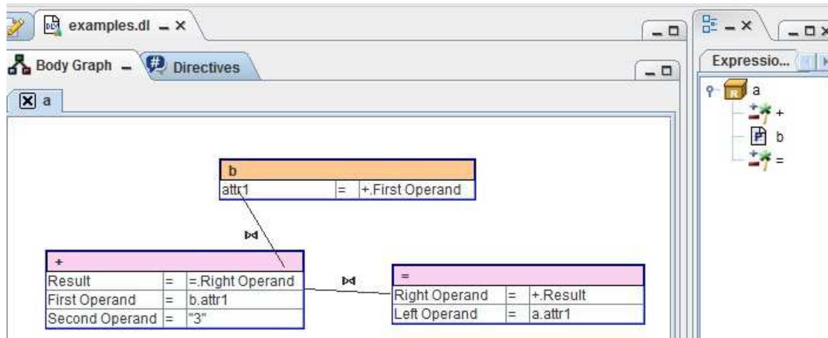


Figure 5.22: A rule with built-in literals.

Another feature consists in the possibility of "*collapsing*" two or more entities by creating a new sub-rule. For example, given the rule

$$:- \mathtt{node(X1)},\ \mathtt{not\ start(X1)},\ \mathtt{not\ reached(X1)}.$$

we can compact the body of the rule by collapsing, for example, the literals `node(X1)` and `not start(X1)`, so that we can see more emphasis on the reachability of nodes that are not the start node of the graph. In particular the rule is transformed to:

$$:- \mathtt{nodeNotStart(X1)},\ \mathtt{not\ reached(X1)}.$$
$$\mathtt{nodeNotStart(X1)} :- \mathtt{node(X1)},\ \mathtt{not\ start(X1)}.$$

To collapse the two literals we select them in the *Body Graph* and click on the *Collapse* icon appearing as popup (fig. 5.23a). The result is depicted in figure 5.23b showing the collapsed entity as a sub-rule (see the *Outline*).

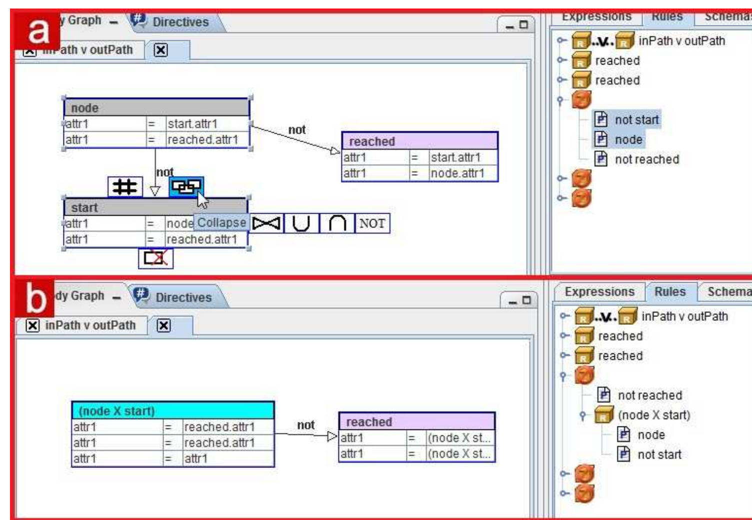


Figure 5.23: Collapsing two body literals.

The last feature is the *Weak Constraint* definition. A *Weak Constraint* is specified in the same way as a constraint and, moreover, *cost* and *level* values can be specified using the *Entity Details* panel.

Chapter 6

Unit Testing in ASPIDE

6.1 Context and Motivation

In software engineering the task of testing and validating programs is a crucial part of the life cycle of the software development process. Software testing [84] is an activity aimed at evaluating the behavior of a program by verifying whether it produces the required output for a particular input. The goal of testing is not to provide a means for establishing whether the program is totally correct. Conversely testing is a pragmatic and cheap way of finding errors. There are two kinds of testing methods:

- **black-box testing**, conceived for verifying some functionalities of an application without knowing the code internals. The programmer, in fact, is only interested in verifying whether a software functionality works correctly by considering input and output data only, without worrying about the behaviour of the internal code. For exploiting this method, a good choice of input data (possibly extreme and critical) is important. For *black-box testing*, the concept of *test coverage* for input data is defined, indicating possible ranges of input data to be considered. A set of tests are considered good if they can, together, cover large input data ranges.
- **white-box testing**, conceived for verifying the behavior of a specific part of a program. White box testing is an activity usually carried out by developers and it is a key component of agile software development [84]. The programmer is interested, in particular, in the behaviour of the internal code (eg. testing whether a function works correctly, whether a *while* cycle never loops and so on). For *white-box testing*, the concept of *test coverage* for code is introduced; the goal is to cover a large part of the lines of the code, and the tester tries to force the execution of specific parts of the code (*if* blocks, function calls, ...) to verify that single parts work correctly.

One of the most diffused white-box testing techniques is *unit testing*. The idea of unit testing is to assess an entire software by testing its subparts called *units*, which correspond to small testable parts of a program; in imperative object-oriented languages, unit testing corresponds to assessing separately portions of the code like class methods. Unit testing methodology is demonstrated to be valid in practice in programming languages (where unit testing has been

introduced), especially when development is driven by testing. Indeed, if some class is modified/removed then unit tests have to be rewritten. Unit tests may help the programmer to fix bugs and implement regression testing, but also:

- they help to understand/correct requirements, and
- they help to detect designs mistakes (e.g., the impossibility of fulfilling a condition may lead to changing the current implementation of a class/library interface).

In these cases (requirement understanding/refactoring) both test cases and program may change.

The crucial task of *testing* ASP programs received less attention [56, 57, 72, 90], and it is an Achilles's heel of the available programming environments. Indeed, the majority of available graphic programming environments for ASP does not provide the user with a testing tool (see Chapter 9). Testing ASP programs was approached for the first time in [56, 57] where the notion of structural testing for ground normal ASP programs is defined and methods for automatically generating tests is introduced. The paper [90] describes the tool supporting the language *Lana* that allows one to define and execute test cases in ASP by exploiting annotations. Regarding imperative programming languages, a high proportion of errors can be found by testing a program for all test inputs within some small scope (Small-Scope Hypothesis); the paper [72] presents some experiments proving that the Small-Scope Hypothesis is valid also for testing ASP programs by using small instances as test input.

Moreover, it is worth noting that testing approaches developed for other logic languages, like PROLOG [55, 91, 21], cannot be straightforwardly ported to ASP because of the differences between the languages[1].

6.2 Contribution

In *ASPIDE*, a pragmatic solution for testing ASP programs was introduced through the implementation of a framework for Unit Testing. In particular, the framework exploits the notion of *unit test* for an ASP program and a new language inspired by the JUnit [60] framework for specifying and running unit tests on ASP programs. The testing language allows the developer to specify the rules composing one or several units, specify one or more inputs and assert a number of conditions on both expected outputs and the expected behavior of sub-programs. The obtained test case specification can be run by exploiting an ASP solver, and the assertions are automatically verified by analyzing the output of the execution. Notably, the test case specification language herein presented is general and applicable to any variant/dialect of ASP. The framework also provides the user with some graphic tools that make the development of test cases simpler.

Concerning the papers [56, 57, 90], the presented results are, somehow, orthogonal to the *ASPIDE* contribution. Only paper [90] proposes a language/

[1]As an example, note that the semantics of a Prolog program changes if we modify the rule order in the program, but this is not true in ASP where rule order is immaterial. Thus, meaningful unit tests in ASP can be obtained by collecting rules without taking care of their "position" in the program, the same does not hold for Prolog.

implementation for specifying/running the produced test cases, while there is no implementation for test case generation. However, the language presented for *ASPIDE* can be used for encoding the output of a test case generator based on the methods proposed in [56].

6.3 Unit Testing in ASP

The methodology of the testing framework of *ASPIDE* is inspired by the JUnit [60] framework; given an ASP program the developer can select the program unit, specify one or more inputs, and assert a number of conditions on the expected output. The obtained test case specification can be run, and the assertions are automatically verified by calling an ASP solver and checking its output. An introduction of the notion of a unit test for a given ASP program is given, and a new language for specifying and running unit tests is presented.

Definition 6.3.1. (Unit Test). *Given a program $\mathcal{P}$, a* unit test *$\mathcal{T}$ for $\mathcal{P}$ is a triple $\mathcal{T} = \langle \mathcal{U}, \mathcal{I}, \mathcal{A} \rangle$ where $\mathcal{U} \subseteq \mathcal{P}$ is the program unit to be tested, $\mathcal{I}$ the input program, and $\mathcal{A}$ is a set of assertions modeling properties that have to be verified by $ANS(\mathcal{U} \cup \mathcal{I})$. A test case $\mathcal{T}$* passes *if all the assertions in $\mathcal{A}$ are satisfied, and* fails *otherwise.*[2] *A* test suite *for a program $\mathcal{P}$ is a set of test cases.*

Basically, a unit test $\mathcal{T}$ focuses on a portion of the ASP program to be tested denoted by $\mathcal{U}$ and called program unit. The inputs for $\mathcal{U}$ are specified by an additional program $\mathcal{I}$, which, in the simplest scenario, can be made of input facts. Since $\mathcal{I}$ can be specified by means of an ASP program, ASP itself can be exploited for modeling different inputs in the same unit test. In order to specify a significant test case both $\mathcal{U}$ and $\mathcal{I}$ have to be specified with care. In particular, the program unit to be tested should "act as a module" so that its behavior can be effectively tested outside the original program. Moreover, $\mathcal{I}$ should only be exploited for specifying the test inputs for $\mathcal{U}$, and should not interfere with the usual "behavior" of $\mathcal{U}$. To this end, the framework provides a way for exploiting the notion of *splitting set* [67] both for specifying the unit program and checking that $\mathcal{I}$ only provides an input for $\mathcal{U}$ and not "interfering" with its execution. In addition, the framework also gives the possibility to the programmer of enforcing checking whether a test case satisfies the more fine grained notion of *DLP-function* [59], so that a more precise modularity of units can be exploited if needed[3].

For example, the testing language of *ASPIDE* that is described in the following allows one to specify the program unit as the bottom program [67] of a given splitting set.

The output of the test case are the answer sets of $\mathcal{U} \cup \mathcal{I}$, and $\mathcal{A}$ contains the specification of a number of properties that have to be satisfied to pass the test. For example, one might require that a given ground atom a is contained in all answer sets $\mathcal{U} \cup \mathcal{I}$, or that the unit program is expected to have a given number of answer sets.

[2]Note that this definition of test case is more general than the one of [56]. Indeed, all test cases in [56] are such that: $\mathcal{U} = \mathcal{P}$, $\mathcal{I}$ is the set of ground inputs, and $\mathcal{A}$ contains the assertions stating that the set of expected outputs is in $ANS(\mathcal{U} \cup \mathcal{I})$.

[3]Note that the notion of DLP-function does not allow the program to be split in a unique deterministic way, thus we decided pragmatically to support this notion only for checking units.

6.3.1 Testing Language

Test cases are specified in the framework by means of text files. A test file can be written according to the following grammar:[4]

```
1 :  invocation("invocationName" [ ,"solverPath", "options" ]?);
2 :  [ [ input("program"); ] | [ inputFile("file"); ] ]*
3 :  [
4 :  testCaseName([ SELECTED_RULES | SPLIT_PROGRAM | PROGRAM ])
6 :  {
7 :  [newOptions("options");]?
8 :  [ [ input("program"); ] | [ inputFile("file"); ] ]*
9 :  [ [ excludeInput("program"); ]
10 :  | [ excludeInputFile("file"); ] ]*
11 :  [
12 :  [ filter | pfilter | nfilter ]
13 :  [ [ (predicateName [,predicateName ]* ) ]
14 :  | [SELECTED_RULES] ] ;
15 :  ]?
16 :  [checkModularity( [ SPLITTING_SET | DLP_FUNCTION ] [,"atoms" ]?  ); ]*
17 :  [ [ selectRule("ruleName"); ]
18 :  | [ selectRulesWithPredicateInHead("predicateName"); ]
19 :  | [ selectRulesWithPredicateInBody("predicateName"); ]
20 :  | [ selectRulesWithPredicateInPositiveBody("predicateName"); ]
21 :  | [ selectRulesWithPredicateInNegativeBody("predicateName"); ]
22 :  | [ selectRulesWithPredicateInAggregates("predicateName"); ]
23 :  ]*
24 :  [ [ assertName( [ intnumber, ]?  [ [ "atoms" ] | [ "constraint" ] ]?); ]
25 :  | [ assertBestModelCost(intcost [, intlevel ]?  ); ] ]*
26 :  }
27 :  ]*
28 :  [ [ assertName( [ intnumber, ]?  [ [ "atoms" ] | [ "constraint" ] ]?  ); ]
29 :  | [ assertBestModelCost(intcost [, intlevel ]?  ); ] ]*
```

A test file might contain a single test or a test suite (a set of tests) including several test cases for the same program to be tested. Each test case includes one or more assertions on the results.

The *invocation* statement (line 1) sets the global invocation settings, which apply to all tests specified in the same file (name, solver, and execution options). The invocation name might correspond to an *ASPIDE* run configuration (see Chapter 4). In this latter case, both the solver path and invocation options are automatically imported from the corresponding run configuration.

The user can specify the program to be tested by writing one or more *input* and *inputFile* statements (line 2). The first kind of statement allows one to write the program to be tested; the second statement indicates a file that contains an input program in ASP format.

A unit test declaration (line 4 and 5) is composed of a name and an optional parameter that allows one to choose whether the unit program corresponds to the entire program (option PROGRAM), or is made of exactly the selected rules (option SELECTED_RULES), or whether the unit program to consider corresponds to the splitting set containing the atoms occurring on the selected rules (option SPLIT_PROGRAM). In the latter case, the "interface" between two splitting sets can be tested (e.g., one can assert some expected properties on the candidates produced by the guessing part of a program by excluding the effect of some constraints in the checking part). Note that, this feature of the language allows specification in a declarative way where parts of the tested program have to be considered in a unit test.

[4]Non-terminals are in bold face, token specifications are omitted for simplicity.

The user can specify particular solver options (line 7), as well as certain inputs (line 8) which are valid in a given unit test. Moreover, global inputs of the test suite can be excluded by exploiting *excludeInput* and *excludeInputFile* statements (lines 9 and 10). The optional statements *filter*, *pfilter* and *nfilter* (lines 12, 13, and 14) are used to filter out output predicates from the test results (i.e., specified predicate names are filtered out from the results when the assertion is executed)[5].

The statement *checkModularity* (line 16) can be added to a unit test to require verification that the selected rules compose either a correct DLP-function [59] or correspond to a splitting set for the tested program. The list of atoms, which can be specified by the user, represents either the input signature if the rules define a DLP-function (which can be joined with the remaining part of the program for a given choice of input/output) or the splitting set if the rules define a split of the program[6].

The statement *selectRule* (line 17) allows one to select rules among the ones composing the tested program. A rule r to be selected must be identified by a name, which is expected to be specified in the input program in a comment appearing in the row immediately preceding r. Actually, in the implementation rule names are added automatically as comments. The selection of the rules can be made also by using predicate names; in particular the statements (lines 18/22) allow one to select rules where a given predicate appears in the head, in the body, in the positive body, in the negative body and in an aggregate atom. Note that, this feature is very useful for selecting the rules composing the definition of a predicate in an easy way.

The expected output of a test case is expressed in terms of assertion statements (lines 24/29). The assertions supported by the language are:

- *assertTrue("atomList")/assertCautiouslyTrue("atomList")*: asserts that all atoms of the atom list must be true in any answer sets;
- *assertBravelyTrue("atomList")*: asserts that all atoms of the atom list must be true in at least one answer set;
- *assertTrueIn(number, "atomList")*: asserts that all atoms of the atom list must be true in exactly *number* answer sets;
- *assertTrueInAtLeast(number, "atomList")*: asserts that all atoms of the atom list must be true in at least *number* answer sets;
- *assertTrueInAtMost(number, "atomList")*: asserts that all atoms of the atom list must be true in at most *number* answer sets;
- *assertConstraint(":-constraint.")*: asserts that all answer sets must satisfy the specified constraint;
- *assertConstraintIn(number, ":-constraint.")*: asserts that exactly *number* answer sets must satisfy the specified constraint;
- *assertConstraintInAtLeast(number, ":-constraint.")*: asserts that at least *number* answer sets must satisfy the specified constraint;

[5] *pfilter* excludes the strongly negated ones, while *nfilter* has opposite behavior.

[6] DLP-Functions offer a fine way of decomposing a program in modules that can be joined together to construct $\mathcal{P}$. The interested reader is referred to [59] for a formal definition.

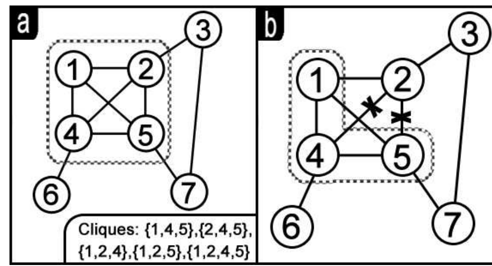


Figure 6.1: Input graphs of the Maximal Clique program.

- *assertConstraintInAtMost(number, ":-constraint.")*: asserts that at most *number* answer sets must satisfy the specified constraint;
- *assertBestModelCost(intcost)* and *assertBestModelCost(intcost, intlevel)*: in case of execution of programs with weak constraints, they assert that the cost of the best model with level *intlevel* must be *intcost*;
- *assertAnswerSetsNumber(number)*: asserts that the tested program is expected to generate exactly *number* answer sets.
- *assertNoAnswerSet*: asserts that the tests is expected to have no answer set.

together with the corresponding negative assertions: *assertFalse*, *assertCautiouslyFalse*, *assertBravelyFalse*, *assertFalseIn*, *assertFalseInAtLeast*, *assertFalseInAtMost*. The *atomList* specifies a list of atoms that can be ground or non-ground; in the case of non-ground atoms the assertion is true if some ground instance matches in some/all answer sets. Assertions can be global (line 20-21) or local to a single test (line 16-17). Note that, the set of supported assertions is redundant; actually, this is on purpose, indeed, having different possibilities to assert the same property eases the task of test case specification for the programmer, who can specify its requirements in the way he prefers.

In the following we report a test case example.

6.3.2 Test case example

We consider, as example, the *Maximal Clique* program introduced in Chapter 3 by considering that graph G is specified by using facts over predicates `node` (unary) and `edge` (binary). Suppose that the encoding is stored in a file named *clique.dl* and suppose also that the graph instance, composed of facts :

```
node(1).node(2).node(3).node(4).node(5).node(6).node(7).
edge(1, 2).edge(2, 3).edge(2, 4).edge(1, 4).edge(1, 5).edge(4, 5).
edge(2, 5).edge(4, 6).edge(5, 7).edge(3, 7).
```

is stored in the file named *graphInstance.dl* (the corresponding graph is depicted in Figure 6.1a).

The following is a simple test suite specification for the above-reported ASP program:

```
invocation("MaximalClique", "/usr/bin/dlv", "");
inputFile("clique.dl");
inputFile("graphInstance.dl");
maximalClique() {
  assertBestModelCost(3);
}
constraintsOnCliques() {
  excludeInput(":~ outClique(X2).");
  assertConstraintInAtLeast(1,":- inClique(1),
         inClique(4).");
  assertConstraintIn(5,":- #count{ X1: inClique(X1) } < 3.");
}
checkNodeOrdering(SELECTED_RULES) {
  inputFile("graphInstance.dl");
  selectRule("r2");
  selectRule("r3");
  assertTrue("uedge(1,2).");
  assertFalse("uedge(2,1).");
}
guessClique(SPLIT_PROGRAM) {
  selectRule("r1");
  assertFalseInAtMost(1,"inClique(X).");
  assertBravelyTrue("inClique(X).");
}
```

Here, we first set the invocation parameters by indicating DLV as solver, then we specify the file to be tested *clique.dl* and the input file *graphInstance.dl*, by exploiting a global input statement. Then, we add the test case *maximalClique*, in which we assert that the best model is expected to have a cost (i.e., the number of weak constraint violations corresponding to the vertices out of the clique) of 3 (*assertBestModelCost(3)*).

In the second test case, named *constraintsOnCliques*, we require that (*i*) vertices 1 and 4 do not belong to at least one clique, and (*ii*) for exactly five answer sets the size of the corresponding clique is greater than 2. (The weak constraint is removed to ensure the computation of all cliques by DLV).

In the third test case, named *checkNodeOrdering*, we select rules r_2 and r_3, and we require to test selected rules in isolation, discarding all the other statements of the input. We are still interested in considering ground facts that are included locally (i.e., we include the file *graphInstance.dl*). In this case we assert that *uedge(1,2)* is true and that *uedge(2,1)* is false, since edges should be ordered by rules r_2 and r_3.

Test case *guessClique* is run in *SPLIT_PROGRAM* modality, which requires to test the sub-program containing all the rules belonging to the splitting set corresponding to the selection (i.e., {*inClique, outClique, node*}). In this test case the sub-program that we are testing is composed of the disjunctive rule and by the facts of predicate *node* only. Here we require that there is at most one answer set modeling the empty clique, and there is at least one answer set modeling a non-empty clique.

6.3.3 Modularity aspects

The notion of DLP-function allows one to decompose ASP programs in modules only in case the program is ground. *DLP_FUNCTION* differs from *SPLITTING_SET* mainly because the first one would not allow to unambiguously select rules for test cases; whereas it is unambiguous how to collect rules belonging to the same "splitting set", the same does not hold in the case of DLP-functions. For this reason, in the specification language, *SPLITTING_SET* can be used for both selecting automatically rules (by setting the *SPLITTING_SET* option) and checking (via *checkModularity* option) whether the selected rules satisfy the condition.

To see why both features cannot be used in the same way for *DLP_FUNCTION*, consider the following example:

```
a v c :- not b.
b :- not e, a.
d :- not a.
```

Suppose we select the disjunctive rule, in the case of the splitting set, the program can be split into:

```
%Module 1
a v c :- not b.
b :- not e, a.
%Module 2
d :- not a.
```

By the definition of splitting set, one has "two" splits (upper and lower component), thus given a set of rules there is an "intuitive" way to complete the test case by adding rules until we split the program in two parts (that can be tested in isolation) in the spirit of the definition.

Now we try to do the same thing with DLP_FUNCTION. The first obstacle arises through input/output/hidden set atoms, which can be arranged in several different ways. This obstacle makes it already not possible in general to generate automatically (in a deterministic way) a test case corresponding to a DLP-function that can be composed with the remaining rules.

Now, we go further. Even if we could define a default way of selecting inputs and outputs, automatic DLP-function generation would still not be unique. Suppose we select the disjunctive rule in our example, and suppose that our "choice" of input/output is *Input*=$\{b\}$, *Output*=$\{a\}$, we still have at least two valid options for splitting the program:

Option 1	**Option 2**
%Module 1	*%Module 1*
`a v c :- not b.`	`a v c :- not b.`
%Module 2	`d :- not a.`
`b :- not e, a.`	*%Module 2*
`d :- not a.`	`b :- not e, a.`

it is, thus, impossible to determine/define which module is the "intended one" in a straight way simply because there are too many degrees of freedom.

Since there is no unique way of exploiting the composition of DLP-functions *ASPIDE* supports this notion of modularity only in checking modality (*checkModularity* option). In this case, we check whether the selected set of rules M (selected by the programmer) can be combined with the remaining part R of the program if we can find an input/output setting where the DLP-functions corresponding to R and M can be joined according to the original definition.

6.3.4 Testing methodology

In an expressive and concise language like ASP also testing small modules (and even one single rule) in isolation may help the programmer to detect bugs. Unit test cases may help the programmer to fix bugs as well as to understand/correct requirements. It might happen that both test cases and programs change if the requirements (i.e., the problem specification) are not well understood, but, in most cases, buggy rules (referenced by name) can be updated without the need to update the corresponding unit tests.

As an example, suppose that the first time we wrote the ASP program of our example we introduced a bug, and in particular a typo on the guessing rule:

```
%@name = r1
inClique(X1) v outClique(X1) :- nod(X1).
```

Note that rule *r1* has a bug since, in the body, `node` is written without the final `e`. When the test case *guessClique* is run it fails since the extension of `nod` (without `e`) is empty.

After fixing the rule *r1* the test case can be now re-run and will pass. In the above example, selecting one rule and testing it in isolation helped to fix the program without the need to update the test case. Note that the possibility of testing parts (i.e., units) of the original program (up to one single rule at time) helps detection by exploiting simple tests to find which specific part of the program does not behave as expected.

The test file described in this Section can be created and executed in *ASPIDE* as described in the next Section. The results can be inspected using the results window that marks the atoms that have participated in the passing/failure of test cases. Actually, in the IDE, test cases can be also created graphically, i.e., without the need to write (textual) test case specifications by hand. This

can be done, starting from the results of an execution, by selecting and marking wanted and unwanted results in an intuitive and "by example" test case creation interface. This is another distinguishing feature of the approach of *ASPIDE* to unit testing, which is described in the following.

6.4 Implementation in *ASPIDE*

In this Section, after overviewing the *ASPIDE* [41] development environment, the graphic tools conceived for developing and running test cases are described.

6.4.1 Unit testing in *ASPIDE*

ASPIDE allows the user to create and execute test suites specified in the language presented in Section 6.3. The user can both manually edit test case files, and he can create test cases by exploiting a number of visual tools. In order to provide a description that immediately gives an idea about the capabilities of the visual testing interface of *ASPIDE*, a step by step guide on how to implement the example illustrated in the previous Section is described. Suppose that we have created a project named *MaxClique* in *ASPIDE*, which contains the files *clique.dl* and *graphInstance.dl* storing the encoding of the maximal clique problem and a graph instance, respectively. Moreover we assume that both input files are included in a run configuration named *MaximalClique*, and we assume that the DLV system is the solver of choice in this run configuration.

Since the file that we want to test in our example is *clique.dl*, we select it in the *Workspace Explorer*, then we click the right button of the mouse and select *New Test* from the popup menu (fig. 6.2a). The system shows the test creation dialog (fig. 6.2b), which allows both the setting of the name of the test file and the selection of a previously-defined run configuration (storing execution options and input files). By clicking on the *Finish* button, the new test file is created (see fig. 6.2c) where a statement regarding the selected run configuration is added automatically. We add the first unit test (called *maximalClique*) by exploiting the text editor (fig. 6.2d), whereas we build the remaining ones (working on some selected rules) by exploiting the logic program editor. After opening the *clique.dl* file, we select rules r_2 and r_3 inside the text editor, we right-click on them and we select *Add selected rules in test case* from the menu item *Test* of the popup menu (fig. 6.2e). The system opens a dialog window where we indicate the test file in which we want to add the new test case (fig. 6.2f). We click on the *Create test case* button. The system will ask for the name of the new test case and we write *guessClique*. After that, in the window, we select the option *execute selected rules* and click on the *Finish* button. The system will add the test case *guessClique* filled with the *selectRule* statements indicating the selected rules. To add project files as input of the test case, we select them from the *Workspace Explorer* and click on *Use file as input* in the menu item *Test* (fig. 6.2g). We complete the test case specification by adding the assertion, thus the test created up to now is shown in Figure 6.2h.

Following an analogous procedure we create the remaining test cases (fig. 6.3a). To execute our tests, we right-click on the test file and select *Execute Test*. The *Test Execution Dialog* appears and the results are shown to the programmer (fig. 6.3b). Failing tests are indicated by a red icon, while green icons indicate

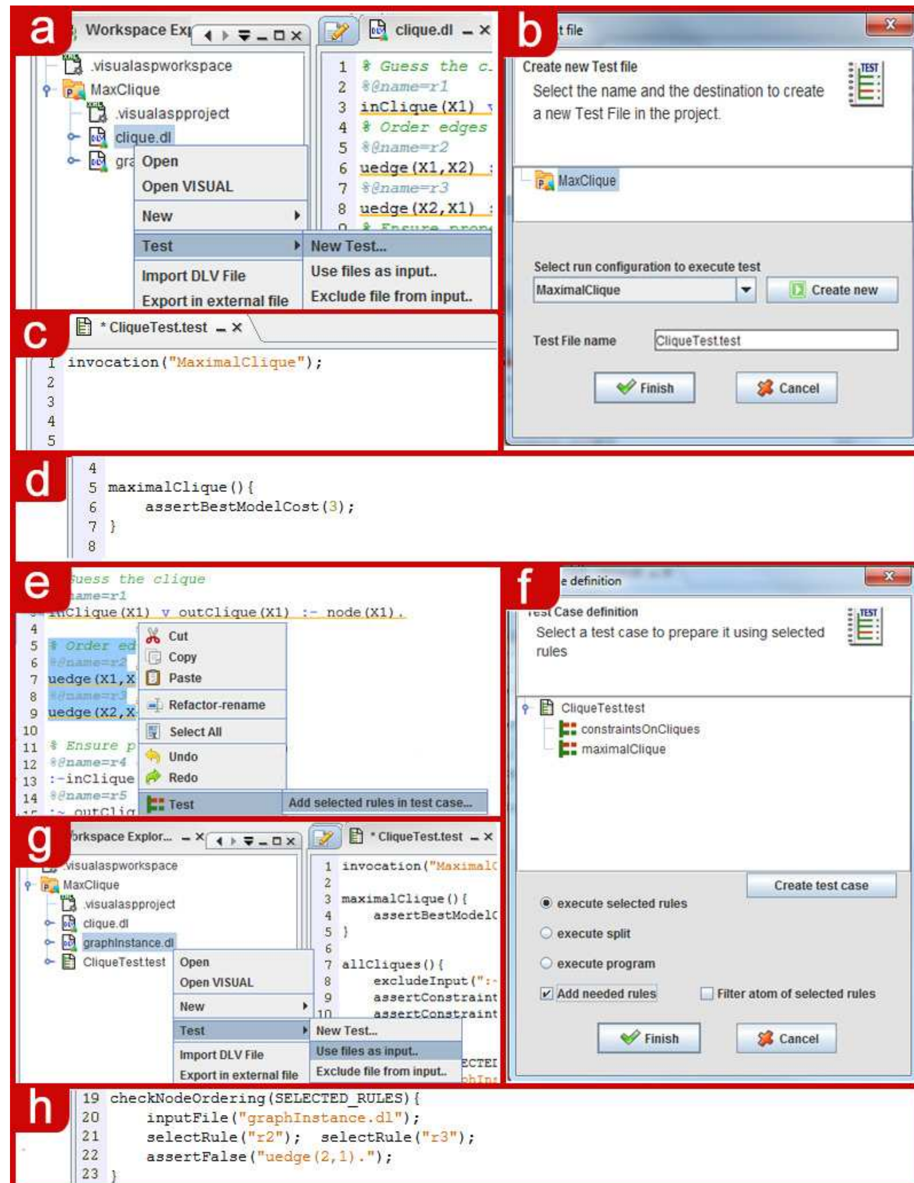


Figure 6.2: Test case creation.

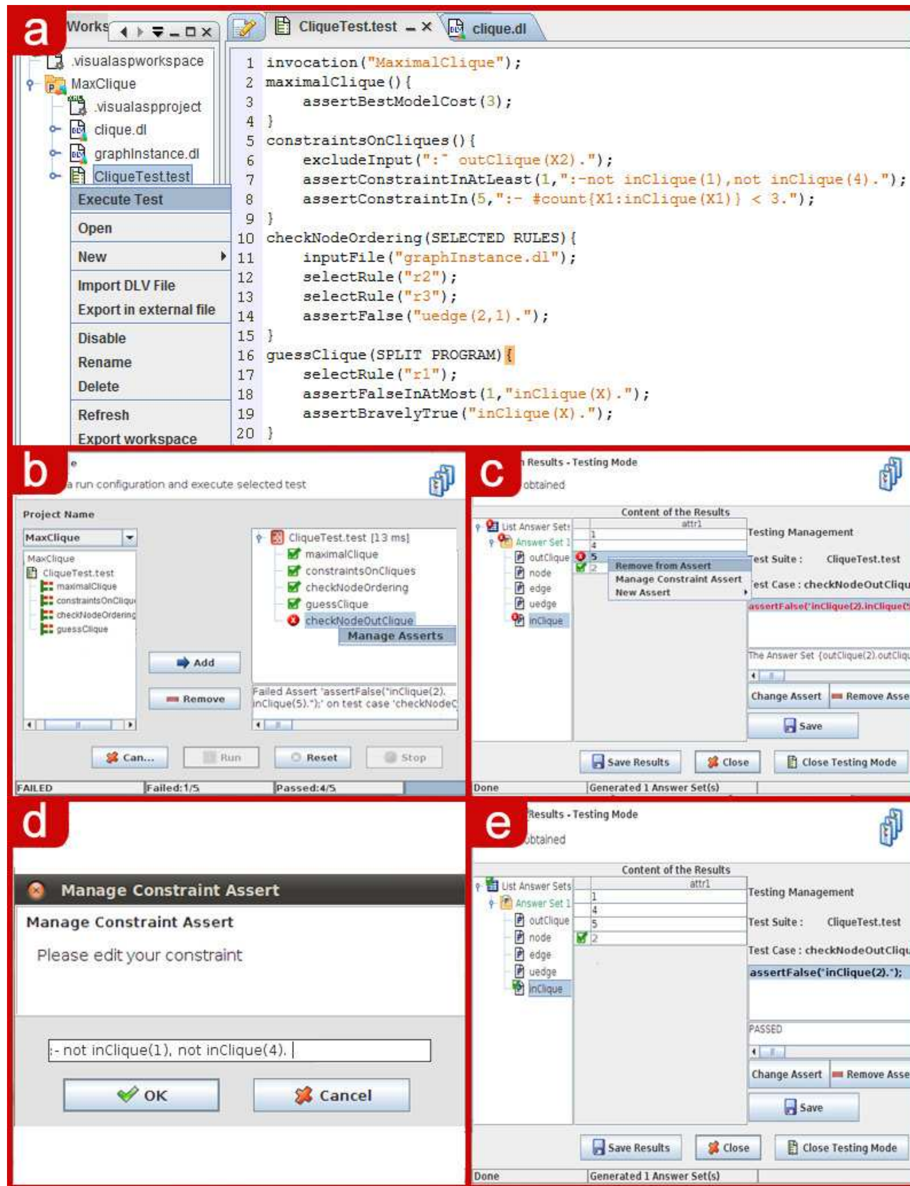


Figure 6.3: Test case execution and assertion management.

passing tests. At this point we add the following additional test:

```
checkNodeOutClique() {
  excludeInput("edge(2,4).edge(2,5).");
  assertFalse("inClique(2).  inClique(5).");
}
```

This additional test (purposely) fails, this can be easily seen by looking at Figure 6.1b. The reason for this failure is indicated (fig. 6.3b) in the test execution dialog. In order to know which literals of the solution do not satisfy the assertion, we right-click on the failed test and select *Manage Asserts* from the menu. A dialog showing the outputs of the test appears where, in particular, predicates and literals correctly matching the assertions are marked in green, whereas the ones violating the assertion are marked in red (gray icons may appear to indicate missing literals which are expected to be in the solution). In our example, the assertion is *assertFalse("inClique(2). inClique(5).")*; however, in our instance, node *5* is contained in the maximal clique composed of nodes *1, 4, 5*; this is the reason for the failing test. Assertions can be modified graphically, and, in this case, we act directly on the result window (Fig. 6.3c). We remove node 5 from the assertion by selecting it. Moreover, we right-click on the instance of *inClique* that specifies node *5* and we select *Remove from Assert*. The atom *node(5)* will be removed from the assertion and the window will be refreshed marking the test case as passed (fig. 6.3e). The same window can be used to manage constraint assertions. In particular, by clicking on *Manage Constraint Assert* of the popup menu, a window appears that allows the user to set/edit constraints (fig. 6.3d).

6.4.2 Visual Editor for building Test Suites

A *TEST file* can be created also by exploiting the *Visual Editor* for Test Suite composition (fig. 6.4). The Visual Editor offers a set of buttons and graphical tools for creating and editing single Test Cases by introducing input files, easy selecting of rules in a program and allowing one to launch the execution for creating/managing assert conditions by exploiting the results.
At the top of the Visual Editor (fig. 6.4) a user can set invocation options by either selecting one available Run Configuration or defining directly solver and execution options. The Visual Editor also offers tools for specifying global inputs, for creating test cases in which the user can set all the components on which a test case can be composed, and for specifying global assert conditions.

Suppose we want to create, by exploiting the Visual Editor, the test case *checkNodeOrdering* shown on the previous Subsection. We open the Visual Editor and we click on the button *New test Case* (fig. 6.5a). A popup window is open where we set the name of the test case and select the option *SELECTED_RULES* (fig. 6.5b); the new test case is created (fig. 6.5c). Now we click on the button *Input file* and select *graphInstance.dl* as input file (fig. 6.5d). For selecting the rules, we click on the button *Select rule* (fig. 6.5e) and set the rules $r2$ and $r3$ (fig. 6.5f). For inserting the assertion `assertTrue("uedge(1,2).");` we click on the button *Manage asserts*; *ASPIDE* starts the execution, gets the results in accordance with the test case specification and shows the results window in testing modality (fig. 6.5g). We select the predicate `edge` from the results window and, by right-click on the tuple representing `edge(1,2)`, we select *New*

Assert, *Assert True* on the popup window (fig. 6.5g) and the assertion will be inserted to the test case.

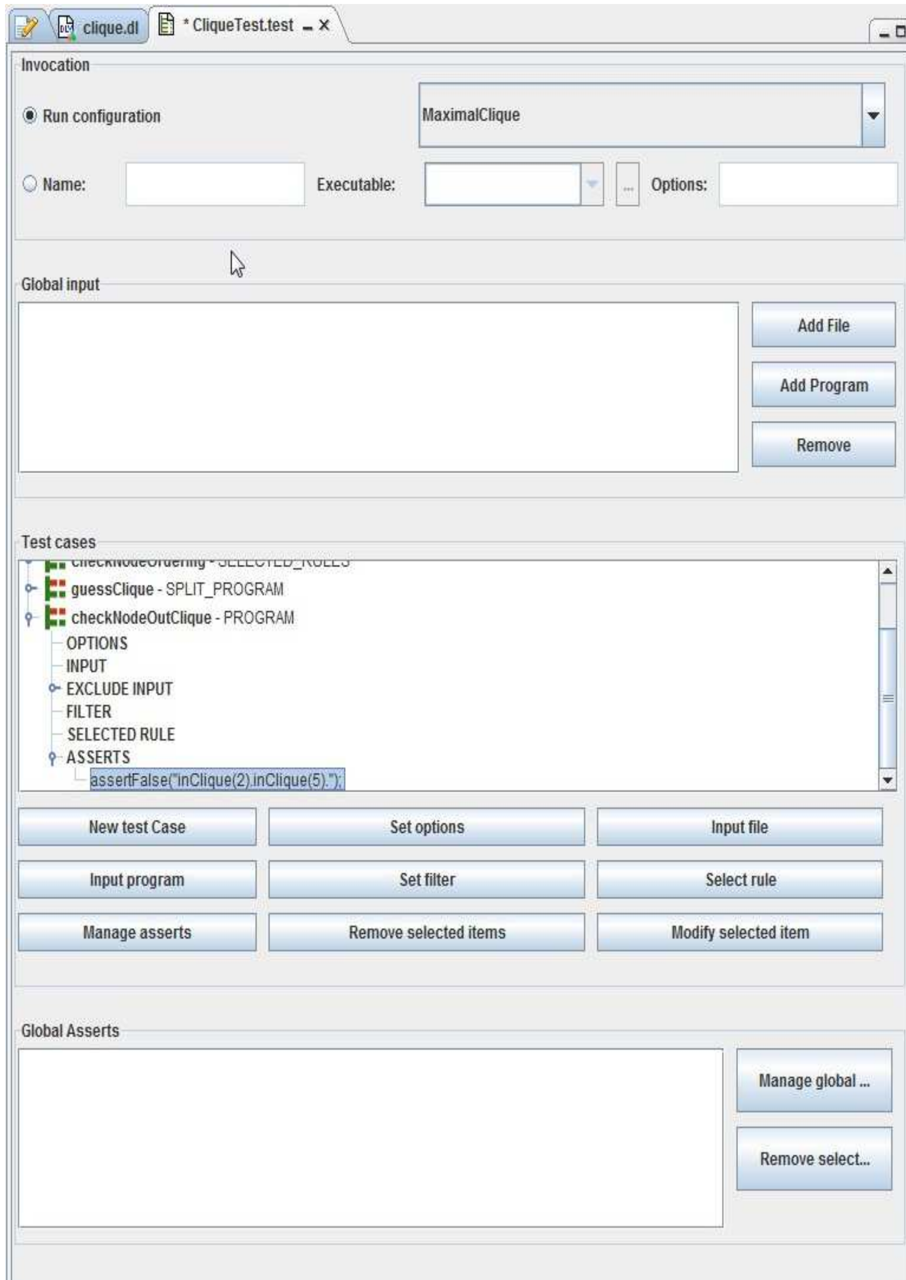


Figure 6.4: Visual Editor for *TEST File* definition.

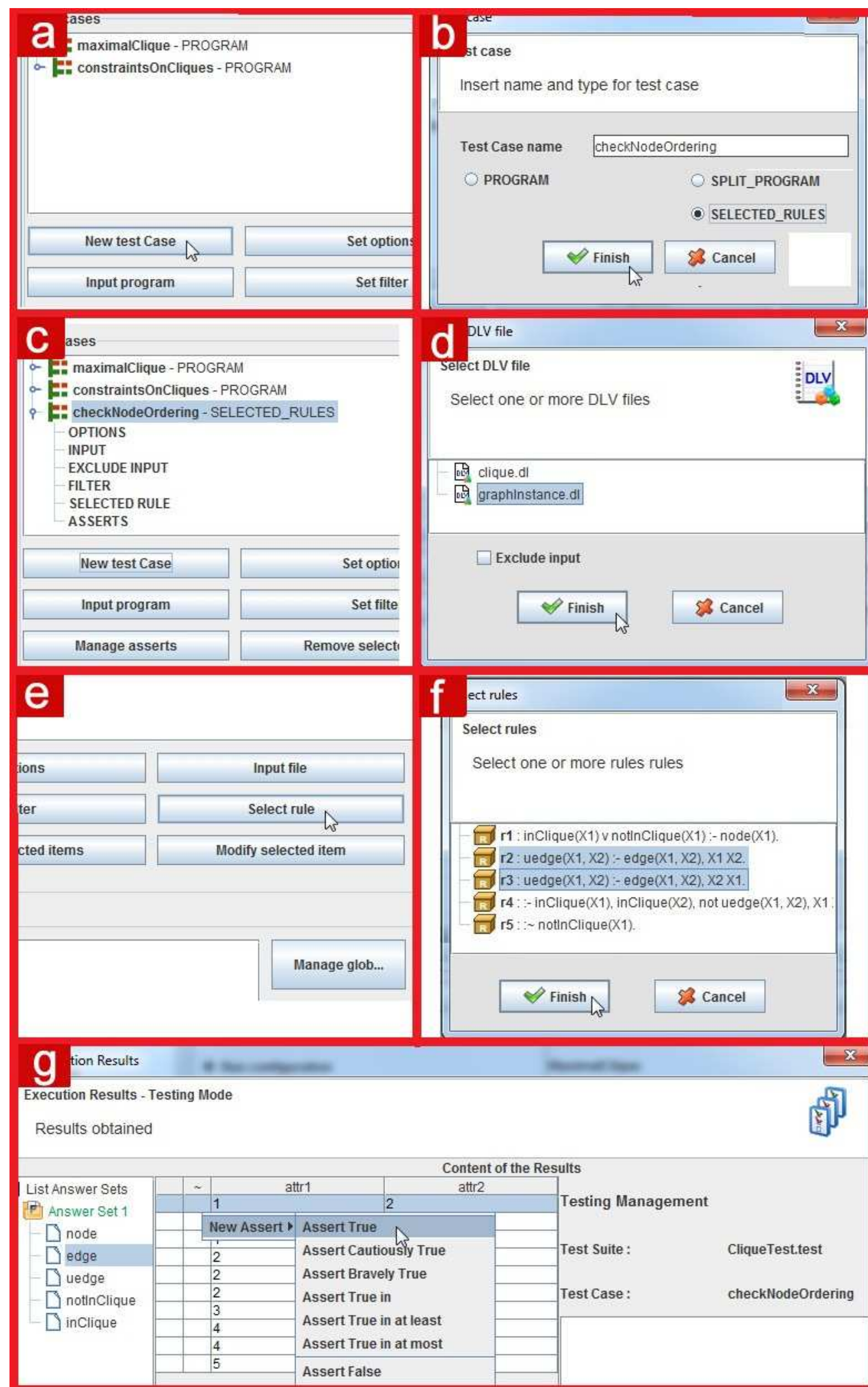


Figure 6.5: Creating a test case by exploiting the Visual Editor.

Chapter 7

Extending ASPIDE with user-defined Plugins

7.1 Motivation and Contribution

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 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.

An important feature of *ASPIDE*, devised with the goal of improving the support to application development, is that it allows one to extend it with user defined plugins. Developers can create libraries that extend *ASPIDE* for:

- handling new input formats;
- performing program rewritings;
- customizing the format of solver results.

A rewriting plugin may encode a procedure that can be applied to rules in the editor (e.g., disjunctive rule shifting can be applied on the fly by selecting rules in the editor and applying the mentioned rewriting).

Now consider a scenario where: input data is generated from a spreadsheet; some complex reasoning task has to be performed on it, and the output has to be loaded back on the spreadsheet for further processing. Thus, data has to be first transformed in a database of facts by applying a suitable knowledge representation. One might export data in CSV and apply a transformation script to obtain this; in turn, the programmer develops an ASP program for reasoning on the input data; finally, the results printed by an ASP solver might be converted back in CSV. An input plugin can take care of the CSV input files that appear in *ASPIDE* as a logic program, and an output plugin can handle the external conversion of the computed answer sets in CSV.

An entire ASP-based application can, in this way, be developed and tested in *ASPIDE* with minimal (or no) need for external conversion tools. To extend *ASPIDE* by introducing features for custom program rewritings, new input/output formats and application-specific features, a programmer must exploit an SDK contained in a JAR library named *AspidePluginKit.jar*. The SDK is distributed under the LGPL licence (freely available at the *ASPIDE* web site `https://www.mat.unical.it/ricca/aspide/`) and provides Java classes and interfaces to be exploited. Everyone can extend ASPIDE with custom features, since the plugin development kit is publicly available to the community.

This Chapter describes the creation process of three plugins examples for: (*i*) handling the ASP RuleML input format, (*ii*) performing disjunctive program shifting, and (*iii*) generating custom XML output. Afterwards, the complete SDK library for plugins development is described in details and deployment/installation processes on *ASPIDE* are shown.

7.2 Input Plugin

The goal of an Input Plugin is to allow *ASPIDE* to load new kind of files and to manage them in the environment. Files stored in an *ASPIDE* workspace can be bound with an Input Plugin so that, new editors can be opened, new input formats can be managed and new commands on those files can be introduced.

The class diagram of Figure 7.1 summarizes the interfaces, contained in the SDK, which must be used to create a new Input Plugin. In the next Subsection the Input Plugin creation process will be illustrated by exploiting a usage example, showing a possible way to use the classes of the SDK for creating an Input Plugin.

7.2.1 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 [32]. The goal of the plugin 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, in 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 the default 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 to the ASP version and shown to the user. In this way the user has the possibility of editing 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 saved in the original format of the source file.

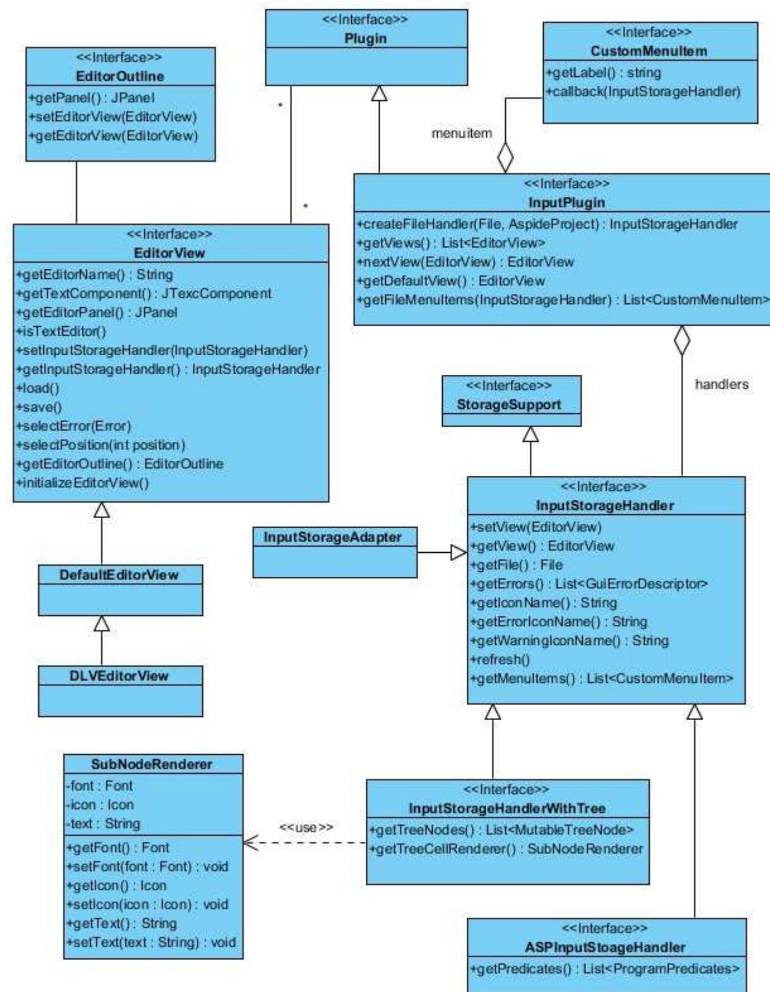


Figure 7.1: Input Plugin interfaces Diagram.

Creating the plugin

To define this plugin we develop a new class *RuleMLPlugin* (the core of our plugin) that implements the interface *InputPlugin* (fig. 7.1). We specify also a new class *RuleMLProgramHandler* that implements *InputStorageHandler* and represents our *ASP RuleML* input file (*ASPIDE* manages instances of *InputStorageHandler* as storage handlers).

On the class *RuleMLProgramHandler* we implement the function *getIconName()* to return the name of the icon that will be associated with the file. *ASPIDE* will retrieve the corresponding icon by accessing to the *Utils* folder that must be contained in the jar library defining our plugin.

On *RuleMLPlugin* we implement the function *createFileHandler* that, given an object *File* and an object *AspideProject* (representing a Project of *ASPIDE*), creates a new instance of *RuleMLProgramHandler* and returns it to the *ASPIDE* environment.

As said before, we need two kinds of editors: a simple XML editor, that shows an *ASP RuleML* file, and a simple ASP editor which shows the ASP syntax version of the file. To design the editors we define two classes named *SimpleEditorXML* and *SimpleEditorASP* that extend *DefaultEditorView* and *DLVEditorView* (both extending the class *EditorView*) respectively. Both editors include a reference to the current Input Plugin and to a *RuleMLProgramHandler* indicating the current storage handler that the editor will load (or has loaded in the case where the editor is currently shown in *ASPIDE*). The two editors are textual editors, so the function *getTextComponent()* must return an object of type *JTextComponent* and the function *isTextEditor()* must return *true* (*false* in case the editor is a generic *JPanel*); in this way *ASPIDE* can get and show these editors to the GUI. The editors that we have just designed need to be managed by *ASPIDE*. To this end we implement, in *RuleMLPlugin*, the functions:

- *getViews()* to return the editor views of the plugin; in this case *SimpleEditorXML* and *SimpleEditorASP*;
- *nextView(EditorView)* to return the *EditorView* that is the successor of the past *EditorView*; this is useful when the user has opened an editor in *ASPIDE* and he wants to switch explicitly to another editor of the same plugin;
- *getDefaultView()* to return the editor view that we want to consider as default; in this case we consider *SimpleEditorASP* as default.

When the content of an editor changes, the editing must be notified to the *ASPIDE* environment, so that the GUI can, for example, say to the user that the file can be saved. To notify editings, we add a *caret listener* on both the *JEditorPane* instances of the editor views, containing a call to the method *notifyModified(EditorView)* of the singleton class *AspideEnvironment*.

The next step of the creation of the *ASP RuleML* plugin is the saving and the loading process of an editor view content. When the user opens or saves the content of an editor view in *ASPIDE*, the *load()* and *save()* methods of *EditorView* are called. If the editor view is a *SimpleEditorXML*, the method *save* (*load*) has to save (load) the pure content of the editor (file), because it is already on the *ASP RuleML* format. If the editor view is a *SimpleEditorASP*, the load method consists on translating the content of the file in the ASP format and showing the translated program on the editor. On the other hand, the save method consists in translating the content of the editor (written in the ASP syntax) to the ASP RuleML syntax before the writing to the file. As a consequence we need to implement a rewriting procedure that exploits a rewriting plugin (see next Section for a detailed description of the rewriting plugin creation). For this purpose we make our class *RuleMLPlugin* to implement also the interface *RewritingPlugin* (fig. 7.3) that contains methods, to be implemented, supporting the rewriting; we implement those methods to make a rewriting procedure from *ASP RuleML* format to ASP format. Finally, we insert, in the class *SimpleEditorASP* a reference to this rewriting plugin so that the method load can (i) read the content of the file, (ii) rewrite the content in the ASP format using the plugin, (iii) show the rewritten content in the editor. When a *SimpleEditorASP*, showing the content in the ASP format, is open in ASPIDE, the

saving procedure has to rewrite the program in the *ASP RuleML* format. To do this we create another class, named *ASPToRuleML*, which works as rewriting plugin and translates a program written in ASP to the syntax of *ASP RuleML*. We include this new plugin in *SimpleEditorASP* and enable the method *save* to call this new rewriting procedure for getting the *ASP RuleML* version to store on the file.

The last step of the *ASP RuleML* plugin creation process is the managing of errors. When a user saves a file using the *SimpleEditorXML*, if the content of the file contains some syntax errors or wrong tags, an error notification is needed in *ASPIDE*. To this end, we implement the function *getErrors()* on the class *RuleMLProgramHandler* to return a list of errors. When an error is detected in the file, the list of errors is updated; *ASPIDE* will call *getErrors()* when it needs to update the program error panel.

Use in *ASPIDE*

We now show the use of the *ASP RuleML* plugin in *ASPIDE*.

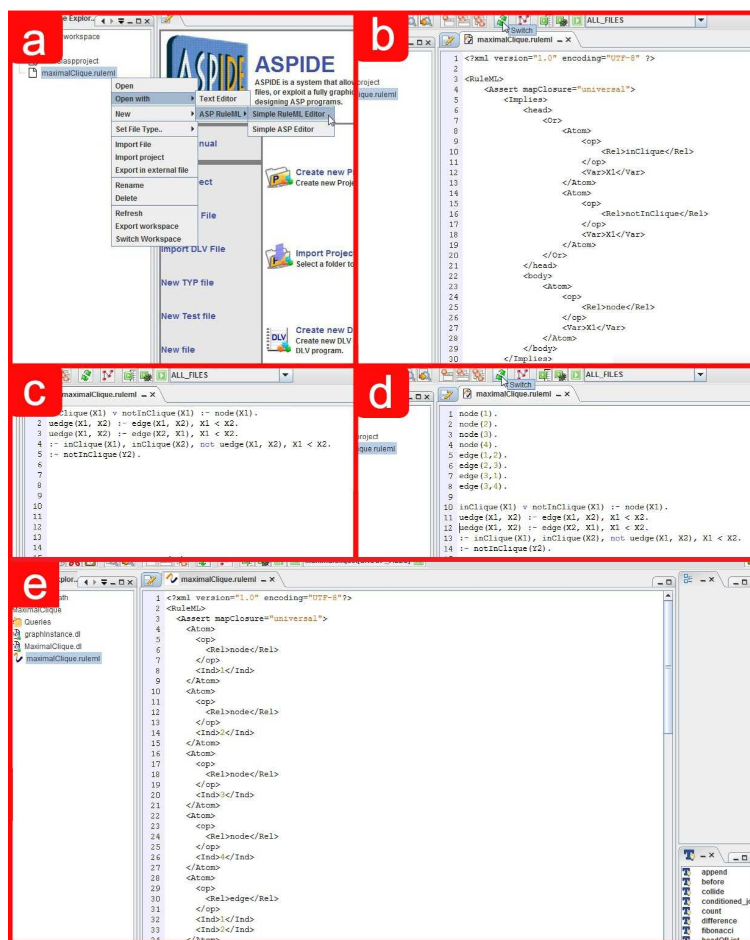


Figure 7.2: *ASP RuleML* plugin at work.

After the installation of it, we create a new project that includes an XML file containing the *MaximalClique* problem encoding (described in Chapter 3) in the *ASP RuleML* format. By a right-click on the file we choose to open it using the plugin and selecting *Simple RuleML Editor* (fig. 7.2a). The file that we are opening is still a simple file; by loading it using an Input Plugin, the type of the file is changed in a new *PLUGIN file* related to the *ASP RuleML* plugin and, consequently, the icon is updated. *ASPIDE* shows the editor (see the *SimpleEditorXML* class) with the content on the file (fig. 7.2b). To switch to the ASP editor to show the program in the DLV format we click on the *switch* button; *ASPIDE* opens the editor (see the *SimpleEditorASP* class) showing the translated version of the program (fig. 7.2c). We now modify the program by adding facts in the *Simple ASP Editor* (fig. 7.2d) and we switch again (after saving) to the *ASP RuleML Editor*; the editor will show the modified *ASP RuleML* version of the file that also includes the facts we have added (fig. 7.2e).

7.3 Rewriting Plugin

The role of a Rewriting Plugin consists in enriching *ASPIDE* with new rewriting procedures. A user can perform rewriting procedures to:

- *workspace files*, by acting on the *Project* and *Workspace* explorer panels; in this case new files will be created containing the rewritten version of the selected files;
- *portions of text* or *sub-programs* contained in text editors; selected portions will be substituted with the rewritten version of them;
- *files inserted in Run Configurations*; files will be rewritten before the execution; the rewritten versions will be passed to the executable system;
- *queries*, before executing a query using the query panel.

The class diagram of figure 7.3 summarizes the interfaces, contained in the SDK, which must be used to create a new Rewriting Plugin. In this Section we describe the creation process of a Rewriting Plugin in *ASPIDE*; as before, a simple scenario representing a typical use of rewriting plugins is described, showing a possible way to use the classes of the SDK for creating a Rewriting Plugin.

7.3.1 A Rewriting Plugin for Shifting ASP Rules

We design a Rewriting Plugin that allows one to rewrite disjunctive rules in order to obtain a “shifted” version, where disjunction is replaced by cyclic nonmononic negation (this rewriting produces an equivalent encoding in case of Head-Cycle Free programs [11]). For example, rule `a v b` is rewritten in `a :- not b` and `b :- not a`, the intuition here is that “disjunction is shifted in the body”.

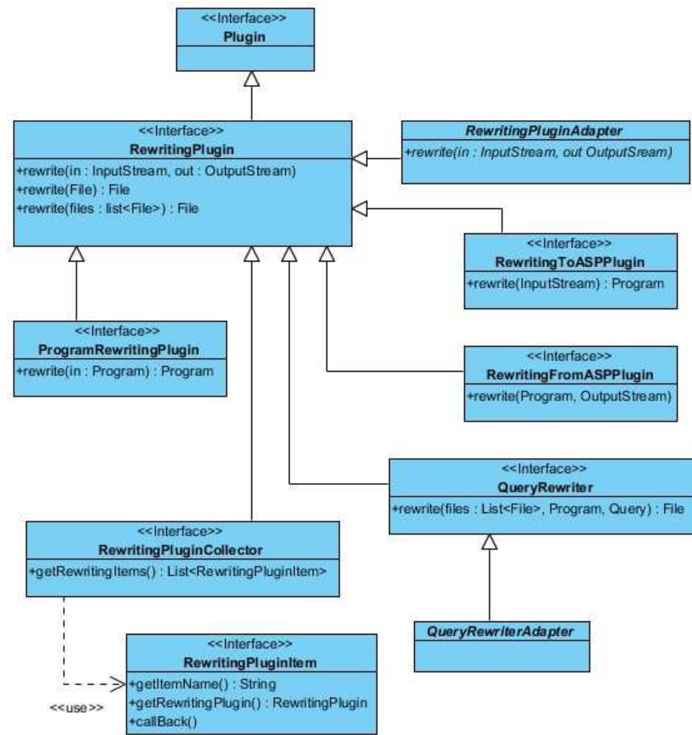


Figure 7.3: Rewriting Plugin interfaces Diagram.

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 to the shifted version. This operation can also be directly performed on a part of the program by selecting rules involved in the DLV editor; in this way only the selected rules are translated into the shifted version.

The user can also find that a solver does not support the disjunction while the files contain disjunctions. In the case that the user does not want to edit the files, the execution can be configured to ask for applying the shifting procedure internally before executing.

Creating the plugin

We specify a new class *ShifterPlugin* that implements the interface *RewritingPlugin* (fig. 7.3). 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.

Use in *ASPIDE*

We now show the use of the *ASP Shifter* plugin in *ASPIDE*.

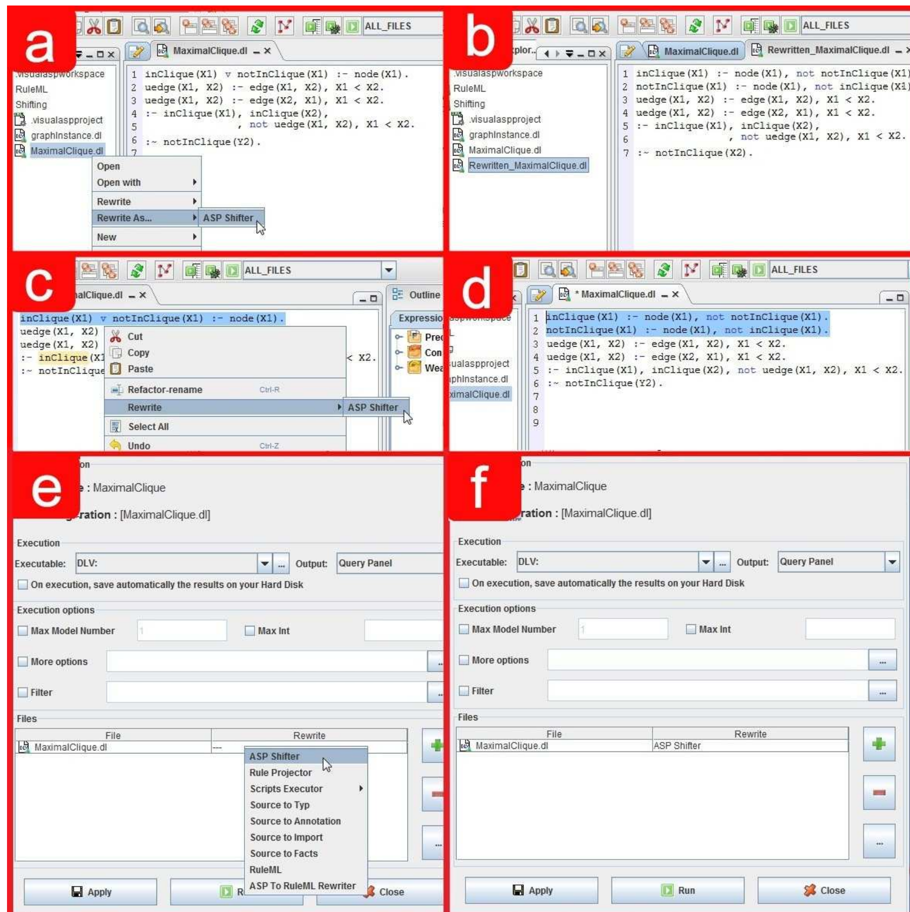


Figure 7.4: *Shifter Plugin* at work.

After the installation, new menu items are introduced in *ASPIDE*, allowing one to use rewriting procedures. A first way of using the *ASP Shifter* 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. 7.4a). The result of this action is an automatic creation of a new rewritten file (fig. 7.4b). 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. 7.4c); selected code is rewritten and replaced with the corresponding shifted code (fig. 7.4d). The shifting rewriter can be applied also to files before execution; by opening the Run Configuration dialog, the user sets, on the file *MaximalClique* and by using a popup menu, the *ASP Shifter* as rewriter to be applied before execution (fig. 7.4e/f).

7.4 Output Plugin

The goal of an Output Plugin is to handle the solver output, generated on the execution process, and to write it in a new format. The generated output can

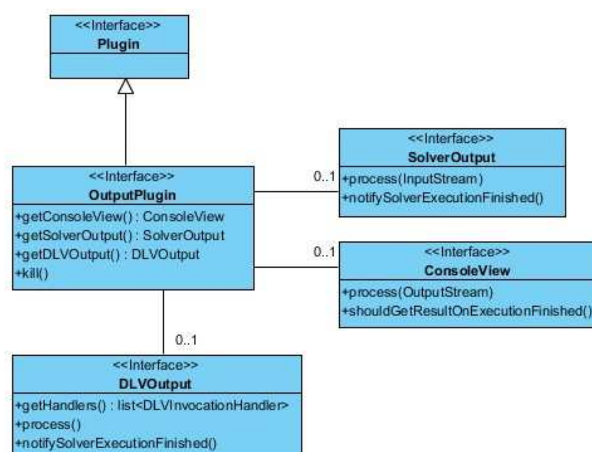


Figure 7.5: Output Plugin interfaces Diagram.

be:

- piped to some other solver as input;
- translated into some other format;
- shown in the Output Console of *ASPIDE* (also after a translating process);
- shown in a new user-defined window created by the plugin developer;
- handled by the *DLVWrapper* for some easy results management;

The class diagram of the figure 7.5 summarize the interfaces, contained in the SDK, which must be used to create a new Output Plugin. In this Section we exploit an example for creating an Output Plugin giving a step-by-step description of the creation process. A simple scenario representing a typical use of output plugins is described, showing also a possible way to exploit the classes of the SDK for creating an Output Plugin.

7.4.1 An Output Plugin for a Custom XML Output

We design an *Output Plugin* that captures the answer sets generated by DLV and rewrites them in a custom XML format to be shown in the Console Window of *ASPIDE*.

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. 7.5). This class allows one to:

- capture the answer sets of the DLV solver;
- rewrite the answer sets in the XML format;
- print the result to the Console Window;
- notify the plugin that the execution is finished.

To capture the answer sets of the solver for applying, if needed, a rewriting procedure, we can exploit the *DLVWrapper* by creating the class *WrapperConsole* that implements *DLVOutput* (fig. 7.5). In the class we implement the function *getHandlers()* for making it to return a list containing a *DLVInvocationHandler* (belonging to the *DLVWrapper*). The *DLVInvocationHandler* handles the answer sets of the solver as Java objects, so that a rewriting procedure can be easily applied (by working directly with data structures) and the results stored to an XML object. The function *getHandlers()* can be implemented in this way:

```
public List<DLVInvocationHandler> getHandlers(){
  List<DLVInvocationHandler> handlers
      = new LinkedList<DLVInvocationHandler>();
  handlers.add(new ModelHandler() {
    public void handleResult(...  , ModelResult model) {
      ...REWRITING CODE FOR THE MODEL...
    }
  });
  return handlers;
}
```

For showing the rewritten results to the console we make the same class *WrapperConsole* also implement *ConsoleView* (fig. 7.5) and we override the method *process(OutputStream)* allowing it to write the custom XML output results 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. To complete the procedure, in the class *CustomXMLOutput* we make the functions *getDLVOutput* (which returns a *DLVOutput* object) and *getConsoleOutput* (which returns a *ConsoleOutput* object) to both return *WrapperConsole* so that *ASPIDE* can correctly manages all the implemented procedures of the plugin.

Use in *ASPIDE*

In *ASPIDE* we open the Run Configuration window by the menu *Execute*.

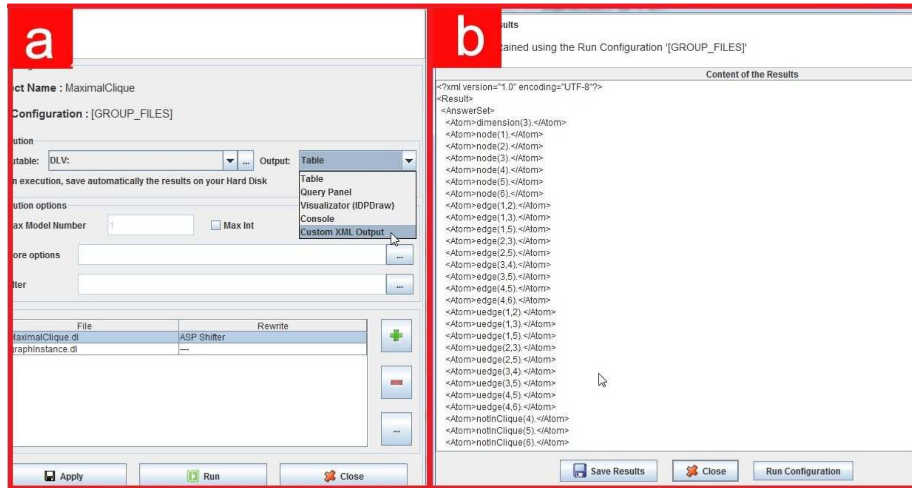


Figure 7.6: *Custom XML* plugin at work.

After the setting of the solver and the specification of the Maximal Clique encoding, we choose to show the results in the *Custom XML Output* format. We select the plugin (fig. 7.6a) 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 in the console (fig. 7.6b).

7.5 The SDK Library for Plugins Development

The *ASPIDE* plugins creation process consists in creating a library containing classes and interfaces that extend and implement the ones defined in the SDK. To install a plugin, *ASPIDE* reads a special folder containing JAR files representing plugin packages; internally *ASPIDE* instantiates the classes of the plugin libraries and calls methods contained in the plugins. On the other hand, installed plugins need also to notify some event or use explicitly some functionalities of *ASPIDE*; to this end, a special interface, named *AspideEnvironment* (fig. 7.9) and provided by the SDK, can be used by plugins to make an explicit notification or request to the *ASPIDE* environment (the *AspideEnvironment* interface will be later described in details).

In this Section, all classes/interfaces of the SDK, involved in new input/ rewriting/output plugins definition, are presented and described including a detailed description of relevant functions/methods[1]. Afterwards, procedures for deploying and installing a created plugin in *ASPIDE* are shown.

7.5.1 Input Plugins classes description

- **InputPlugin**. The main interface of the Input Plugin management; by creating a new class that implements this interface, a new Input Plugin is defined and made available in *ASPIDE*.

  Functions/Methods:

[1] Functions and methods signatures are written using the UML notation.

  - `createFileHandler(File, AspideProject):InputStorageHandler`. Given a *File* and an *AspideProject*, creates a new *InputStorageHandler*, which can be managed by the plugin, and returns it to the *ASPIDE* environment.
  - `getViews():List<EditorView>`. Returns the list of the editors defined in the plugin.
  - `nextView(EditorView):EditorView`. Returns the next *EditorView* of the one given as parameter.
  - `getDefaultView():EditorView`. Returns the *EditorView* promoted as default editor of the plugin.
  - `getFileMenuItems(InputStorageHandler):List<CustomMenuItem>`. Given an *InputStorageHandler* file of the plugin, returns a list of *CustomMenuItem* objects representing additional actions that the user can do on the *PLUGIN file*.

- **CustomMenuItem**. *CustomMenuItem* objects represent custom menu items defined in the plugin. In particular, when the user is doing a *right-click* on an *InputStorageHandler* file of the current plugin, the popup menu includes also *CustomMenuItem* objects.

  Functions/Methods:

  - `getLabel():String`. Returns a string representing the menu item name that will be used to identify the action on the popup menu.
  - `callBack(InputStorageHandler)`. This method is called when the user clicks on the menu item in *ASPIDE*. The method should consequently be implemented properly by considering what action should happen. The parameter is the *PLUGIN file* on which the action is performed.

- **InputStorageHandler**. Instances of any classes that implement this interface represent *PLUGIN files* associated with the Input Plugin that we are currently creating. Only classes of type *InputPlugin* must create *InputStorageHandler* instances. The interface implements *StorageSupport* that represents any files handled in *ASPIDE* (*DLV files*, *TYP files*, ...).

  Functions/Methods:

  - `setView(EditorView)`. When the open action of the *PLUGIN file* is done in *ASPIDE*, this method is called that sets the current view in the *InputStorageHandler*.
  - `getView():EditorView`. Returns the current view used to open the *PLUGIN file*.
  - `getFile():File`. Returns the *File* Java object representing, phisically, the *PLUGIN file*.
  - `getErrors():List<GuiErrorDescriptor>`. Returns a list of the errors contained in the *PLUGIN file*.
  - `getIconName():String`. Returns the name of the icon to be associated to the *PLUGIN file*. The icon must be inserted in the *Utils* folder included in the final JAR distribution package of the plugin.

  - `getErrorIconName():String`. Returns the name of the icon to be associated with the *PLUGIN file* as icon signaling that there is an error on the file. The icon must be inserted in the *Utils* folder included in the final JAR distribution package of the plugin.
  - `getWarningIconName():String`. Returns the name of the icon with be associated to the *PLUGIN file* as icon signaling that there is a warning on the file. The icon must be inserted in the *Utils* folder included in the final JAR distribution package of the plugin.

- **InputStorageHandlerWithTree**. This interface extends *InputStorageHandler* and provides functions/methods that allows one to define a sub-tree on the *PLUGIN file*. The sub-tree must be composed of a list of *MutableTreeNode* Java objects; complex sub-trees can be built thanks to the possibility of inserting other nodes as children of the *MutableTreeNode* objects. Trees will be visible in the *Project Explorer Panel* and in the *Workspace Explorer Panel*.

  Functions/Methods:

  - `getTreeNodes():List<MutableTreeNode>`. Returns a list of *MutableTreeNode* objects to be inserted as a sub-tree of the *PLUGIN file*.
  - `getTreeCellRenderer():SubNodeRenderer`. Returns a *SubNodeRenderer* object specifying the rendering of the sub-tree nodes.

- **SubNodeRenderer**. Represents the node renderer of the sub-trees specified in *InputStorageHandlerWithTree* objects.

  Functions/Methods:

  - `getText():String`. Returns the text label value to be assigned to the node.
  - `getFont():Font`. Returns the font type to be assigned to the node.
  - `getIcon():Icon`. Returns the icon to be assigned to the node.

- **ASPInputStorageHandler**. This interface extends *InputStorageHandler* with ASP specific methods and functions. A *PLUGIN file* that implements this interface can be seen as a simple ASP file.

  Functions/Methods:

  - `getPredicates():List<ProgramPredicate>`. Returns a list of ASP predicates contained in the *PLUGIN file*.

- **EditorView**. Any editors of the plugin can be created by implementing this interface. The editors are shown in *ASPIDE* every time the user chooses to open a *PLUGIN file* using one of them. The editors can be configured to show either a text editor or a generic custom panel containing editing components.

  Functions/Methods:

  - `getEditorName():String`. Returns the name of the editor. The name is used in *ASPIDE* to allow the user to easily identify the editor.

- `getTextComponent():JTextComponent`. In case the editor works as text editor, the function must return a *JTextComponent* object to be visualized and used in *ASPIDE*.
- `getEditorPanel():JPanel`. In case the editor does not work as text editor, *ASPIDE* calls this function for getting a custom *JPanel* object to be visualized as editor.
- `isTextEditor():boolean`. The function is used by *ASPIDE* to determinate which function between `getTextComponent()` and `getEditorPanel()` should be called for getting the editor component.
- `setInputStorageHandler(InputStorageHandler)`. Sets the *InputStorageHandler* file to be loaded in the editor.
- `getInputStorageHandler():InputStorageHandler`. Returns the *InputStorageHandler* file that was loaded in the editor.
- `load()`. The method is called when *ASPIDE* opens a file with the editor. Using this method the plugin can decide in which way the file should be interpreted and visualized in the editor. For example a rewriting procedure can be performed before the real loading.
- `save()`. The method is called when the user chooses to save the content of the editor from the GUI. Using this method the plugin can decide in which way the editor content should be interpreted before the physical writing on the file. For example, if the editor is a *JPanel* the plugin can perform a building procedure to a language (eg. XML) and write the result to the file.
- `selectError(Error)`. Highlights the error (syntax or generic error), in the editor, which is passed as a parameter.
- `selectPosition(position:int)`. Inside the editor, moves the cursor, or sets the focus, to the specified position. The position can be seen as a line number in the case of text editors, while in the case of *JPanel* editors, the plugin developer can decide the meaning of the *position* parameter.
- `getEditorOutline():EditorOutline`. Returns the outline associated with the editor as *JPanel* object.

• **EditorOutline**. The interface represents an outline that can be associated with one or more editors of the plugin. In general, outlines give a graphical representation of the content of an editor and can be used to quickly access a specific point of the editor that contains a selected graphical object.

Functions/Methods:

- `getPanel():JPanel`. Returns the *JPanel* object that works as outline.
- `setEditorView(EditorView)`. Sets the current editor view associated with the outline.
- `getEditorView():EditorView`. Returns the editor view currently associated to the outline.

- **InputStorageAdapter**. A utility class that implements the *InputStorageHandler* interface. It already implements, by default, the main functions and methods of the interface, so that a plugin developer can, by subclassing *InputStorageAdapter*, define a *PLUGIN file* more quickly (by avoiding implementing all the methods).

- **DefaultEditorView**. An utility class that implements the *EditorView* interface. It already implements, by default, the main functions and methods of the interface, so that a plugin developer can, by subclassing *EditorView*, define an editor with default features. In particular, this implementation of the editor gives line numbering and a minimal coloring on some keywords.

- **DLVEditorView**. A utility class that extends the *DefaultEditorView* class. In addition to the features offered by the superclass *DefaultEditorView*, it also offers minimal suggestions for auto-completion and more coloring for DLV keywords like `v` and `not`.

7.5.2 Rewriting Plugins classes description

- **RewritingPlugin**. The main interface of the Rewriting Plugin management; by creating a new class that implements this interface, a new Rewriting Plugin is defined and made available in *ASPIDE*. The interface offers functions suitable for implementing rewriting algorithms.

  Functions/Methods:

  - `rewrite(in:InputStream,out:OutputStream)`. Receives an input stream object representing the data flow to be rewritten. When a user asks to perform a rewriting procedure in *ASPIDE* using the plugin, in the most cases this method is called giving as input stream the content to be rewritten. The method must write the result of the rewriting procedure to the output stream *out*; *ASPIDE* will manage the output stream getting the result for a purpose (e.g. storing it in a file, showing it in editors).
  - `rewrite(files:List<File>):File`. Reads the contents of the passed files and rewrites them to a new file that will be returned by the function. The destination folder of the output file can be decided by the function itself. The function is called when a user asks for a rewriting procedure to files using the *Workspace explorer* panel.
  - `rewrite(in:File):File`. Reads the content of the passed file and rewrites it to a new file that will be returned by the function. The destination folder of the output file can be decided by the function itself. The function is called when a user asks for a rewriting procedure before the execution.

- **ProgramRewritingPlugin**. Classes that implement this interface will be considered by *ASPIDE* as specific ASP rewriters. The interface allows the programmer to exploit ASP Java objects offered by the *DLVWrapper* [79] for performing rewriting procedures.

  Functions/Methods:

   - `rewrite(in:Program):Program`. Receives a *Program* object of the *DLVWrapper* that must be transformed into another *Program* object representing the rewritten version of the passed *Program*. Exploiting objects makes the rewriting procedure easy, because a parsing procedure is not necessary. The function is called by *ASPIDE* when the user asks, on the DLV editor, to perform the rewriting procedure to a sub-program.

- **QueryRewriter**. Allows one to perform query rewriting procedures for an ASP program.

  Functions/Methods:

   - `rewrite(files:List<File>,Program,Query):File`. Given a list of files, a program and a query, the function returns a new file containing the results of the rewriting procedure. *ASPIDE* expects the resulting file to contain a rewritten query. The function is called by *ASPIDE* when the user asks through the query panel, to carry out a query by applying, at first, a query rewriting procedure. The list of files passed as parameter represent either files contained in a Run Configuration or a file resulting from a previous query rewriting procedure. The *Program* parameter represents an additional program inserted to the query panel, and the *Query* parameter represents either the query specified to the query panel or a query resulting from a previous query rewriting procedure.

- **RewritingToASPPlugin**. The interface is used by Input Plugins. In particular, if an Input Plugin implements this interface, it becomes also a Rewriting Plugin that allows files to be loaded, written in a format, in the ASP format to be shown directly to the DLV Editor of *ASPIDE* (by enabling the possibility to exploit the DLV Editor features for editing the program). The Rewriting Plugin must consequently implements a rewriter to the ASP format. By exploiting this interface the programmer does not need to define a custom editor in the case when the user needs only to open files (of some format) in *ASPIDE* using the ASP syntax.

  Functions/Methods:

   - `rewrite(InputStream):Program`. Given an input stream representing the content of a *PLUGIN* file, rewrites the content to ASP by building an object *Program* of the *DLVWrapper*. *ASPIDE* will open the content of the rewritten program to the DLV Editor.

- **RewritingFromASPPlugin**. Differently from the interface *RewritingToASPPlugin*, this interface allows one to perform the reverse operation, from the ASP syntax to the original format. If an Input Plugin implements *RewritingToASPPlugin* only, when the user opens a *PLUGIN file* (handled by the same Input Plugin) in *ASPIDE* by exploiting the "Open in ASP" procedure, the save procedure is disabled; if the programmer makes the Input Plugin to implement also *RewritingFromASPPlugin*, a reverse rewriting from ASP to the original format is introduced. In this way the "save" procedure (became enabled) can exploit it to store to the file the rewritten DLV Editor content.

  Functions/Methods:

  - `rewrite(in:Program,out:OutputStream)`. Given an object *Program* of the *DLVWrapper*, representing the content of a DLV Editor, rewrites the content to the original format of the *PLUGIN file* and writes it to the *OutputStream*.

- **RewritingPluginCollector**. The interface extends *RewritingPlugin* and allows the programmer to define a collection of rewriting plugins by using a unique rewriting plugin. In addition to the collection, the programmer can also add custom commands that the user can invoke in *ASPIDE*; in this case the programmer can, for example, allow the user to set some property on the rewriting process. *ASPIDE* collects both the collection of rewriting plugins and the custom commands in sub menu items where the parent menu is the Rewriting Plugin collector.

  Functions/Methods:

  - `getRewritingItems():List<RewritingPluginItem>`. Returns a list of *RewritingPluginItem* representing, each one, either other Rewriting Plugins to be added to the collector or action components that the user can invoke in *ASPIDE*.

- **RewritingPluginItem**. A class that implements this interface can represent either a Rewriting Plugin or a custom command; the class can decide, itself, if it is a Rewriting Plugin or not.

  - `getItemName():String`. Returns the name of the rewriting plugin item.
  - `getRewritingPlugin():RewritingPlugin`. If the rewriting plugin item represents a Rewriting Plugin, the function returns a not null *RewritingPlugin* object. In the case where the item represents a custom command, the function returns *null*.
  - `callBack()`. In the case where the item represents a custom command, this method will be called by *ASPIDE* when the user exploits this item. In this method a custom procedure can be implemented.

- **RewritingPluginAdapter**. A utility class that already implements the rewriting functions for files. In particular, they call the function `rewrite (in:InputStream,out:OutputStream)` by passing, as input stream, the content of the received files; the results are written to a new file that will be saved to the same project folder. The function `rewrite` that receives the input and output stream still must be implemented by the programmer.

- **QueryRewriterAdapter**. A utility class, extending the *RewritingPluginAdapter* class, that implements *QueryRewriter*. In this class, the function `rewrite(f:List<File>,p:Program,q:Query):File`, of the *QueryRewriter*, as default behaviour, calls the method `rewrite(List<File>): File` passing, as parameters, the list of files `f` and a temporary file containing both the passed program `p` and the query `q`. By using this class the programmer has to implement, only, the method `rewrite(InputStream, OutputStream)` of the *RewritingPlugin* interface.

Example 7.5.1. Let us consider the *ASP RuleML* plugin of the previous Section. Suppose we want to open an *ASP RuleML* file directly to the DLV Editor.

We make the class *RuleMLPlugin* to implement the interface *RewritingToASP-Plugin* and we implement the contained method. In this way a user can open an *ASP RuleML* file by exploiting the command *Open in ASP*, *Text Editor* of the popup menu; the DLV Editor shows, consequently, the rewritten content of the file (fig. 7.7). □

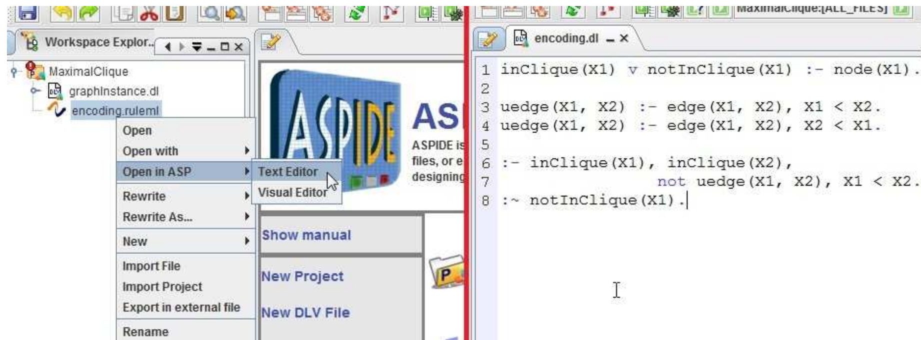


Figure 7.7: Opening of an *ASP RuleML* file using the DLV Editor.

Example 7.5.2. ASPIDE *Scripts Executor* is a Rewriting Plugin, already available in *ASPIDE* as built-in plugin, which has been implemented using *Rewriting-PluginCollector*. In particular, the collection of rewriting plugins are represented by script files contained in a folder; each script file is a rewriting plugin belonging to the collection. The folder can be also changed by exploiting a particular command that is added to the same collection. For example, suppose that the folder, set previously, contains two script files, named *shifter* and *magicSet*, performing the shifting and the magic set rewriting procedures respectively; the plugin will create two Rewriting Plugins, namely *shifter* and *magicSet* and will put them to the collection. When the user exploits the *Script Executor* plugin, he can use one of the two plugins (fig. 7.8). □

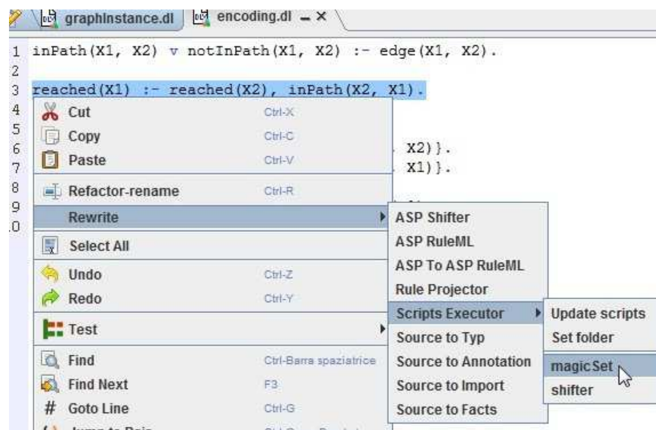


Figure 7.8: *ASPIDE* Script Executor plugin at work.

7.5.3 Output plugins classes description

- **OutputPlugin**. The main interface of the Output Plugin management; by creating a new class that implements this interface, a new Output Plugin is defined and made available in *ASPIDE*. The interface offers functions suitable for getting directly the solver results (in this case the plugin can choose to parse or pipe it to a destination) or handling the results by using the Java object of the *DLVWrapper*.

  Functions/Methods:

  – `getSolverOutput():SolverOutput`. Returns a *SolverOutput* object that will use *ASPIDE* to pass the solver results directly to the plugin.

  – `getDLVOutput():DLVOutput`. Returns a *DLVOutput* object that contains handlers of the DLVWrapper. In this way, by using the solver results, *ASPIDE* builds objects of the *DLVWrapper* and calls the handlers passing those objects.

  – `getConsoleView():ConsoleView`. If the function returns a not null *ConsoleView*, *ASPIDE* opens the console writing the results inserted to the *ConsoleView* object by the plugin.

  – `kill()`. If *ASPIDE* detects that the execution was killed (by the user or by the solver itself), this method is called to notify the plugin that the execution was killed.

- **SolverOutput**. Receives the solver results directly in the execution phase.

  Functions/Methods:

  – `process(InputStream)`. When the execution is started, the solver results are passed to this method via the *InputStream* object.

  – `notifyExecutionFinished()`. When the execution of the solver is finished, this method is called to notify the end of the execution.

- **DLVOutput**. Provides, to the *ASPIDE* environment, a way to give to the plugin the solver results as *DLVWrapper* Java objects.

  Functions/Methods:

  – `getHandlers():list<DLVInvocationHandler>`. Returns a list of *DLVWrapper* handlers that will be called passing the *DLVWrapper* Java objects generated using the solver results.

  – `process()`. When the execution is started, the plugin is notified by this method.

  – `notifyExecutionFinished`. When the execution of the solver is finished, this method is called to notify the end of the execution.

- **ConsoleView**. It is used to get, from the plugin, the content that must be written to the console of *ASPIDE*.

  Functions/Methods:

  – `process(OutputStream)`. The content that this method writes to the *OutputStream* will be visualized to the console.

– shouldGetResultsOnExecutionFinished():boolean. Returns *true* if the results should be written to the console only when the execution is finished; *false* if everything is written to the *OutputStream* passed to the method *process* should immediately be written to the console.

7.5.4 The *AspideEnvironment* Java interface

Many times, *ASPIDE* needs to be notified for some event from plugins; for this purpose some calls to functions of the *AspideEnvironment* interface are needed for notification or for getting some request. For example, an Input Plugin must call the method *notifyModified* to tells *ASPIDE* that an editor was modified by the plugin. We now describe in detail the function/methods available in the *AspideEnvironment* interface[2] (fig. 7.9).

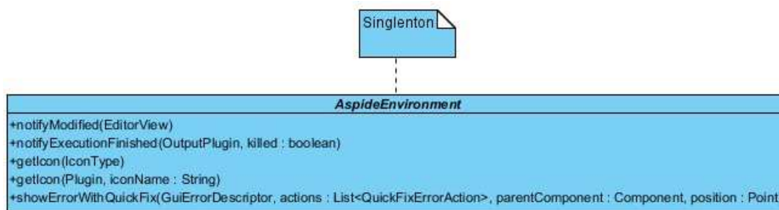


Figure 7.9: *AspideEnvironment* Java interface.

- notifyModified(EditorView). In the case where *ASPIDE* has opened a custom editor defined by the plugin and it edits something on the editor, the plugin must also call this method to notify the event to *ASPIDE*.
- notifyExecutionFinished(p:OutputPlugin,killed:boolean). In the case where an Output Plugin has finished working with the output of the solver, the plugin must call this method to notify the ending to *ASPIDE*.
- getIcon(IconType):ImageIcon. A plugin can invoke this function to get an icon that is already available in *ASPIDE* by default.
- getIcon(p:Plugin,iconName:String):ImageIcon. The function provides a quick way to get icons contained in the *Utils* folder of the JAR library defining the plugin.
- showErrorWithQuickFix(GuiErrorDescriptor,actions :List<QuickFixErrorAction>,parentComponent:Component ,position:Point). Every plugin can exploit an easy way to use the quick fixes mechanism of *ASPIDE*. By calling this method, an error with possible custom quick fixes can be visualized to the user.

7.5.5 Implementation, Deploy and Installation of *ASPIDE* plugins

When a plugin was totally implemented, the final step that allows one to use it in *ASPIDE* consists in preparing a JAR (Java Archive) file to be installed in

[2]Functions/methods signatures are written using the UML notation.

the environment. The JAR file must contain all compiled classes that compose the plugin; moreover, an XML configuration file (called *plugin.xml*) must be inserted to the root of the JAR file providing specific information. In the follow an XML configuration file example:

```
<pluginpack name="packName" version="X.Y.Z" author="Name">
                 sdk_version="U.W.T">
  <plugin name="PluginName1" type="input|rewrite|output"
                 class="package1.ClassName1">
    <filetype extension="fileExtension1"/>
    <filetype extension="fileExtension2"/>
    ...
    <lib jarname="jarName1.jar"/>
    <lib jarname="jarName2.jar"/>
    ...
  </plugin>
  <plugin name="PluginName2" type="input|rewrite|output"
                 class="package2.ClassName2">
    ...
  </plugin>
</pluginpack>
```

An *ASPIDE* plugin JAR file may contain more than one plugin. In the XML configuration file, for each plugin defined to the JAR file, a tag *<plugin>* must be specified containing the name, the type (input and/or output and/or rewrite), the version, the author of the plugin and the version of the SDK used to build the plugin (useful for compatibility of plugins with different versions of *ASPIDE*). File extensions for associating each plugin with specific files can be introduced using a tag *<filetype>*. To enable plugins to exploit external libraries, a tag *<lib>* for each library must be specified with the library name. For including icons to plugins, a folder *Utils* must be created, to the root of the JAR file, containing icon files (for setting inserted icons to the PLUGIN files in *ASPIDE*, see Section 7.2).

When the JAR file is complete, it can be installed in *ASPIDE* by exploiting a dedicated dialog for plugins management.

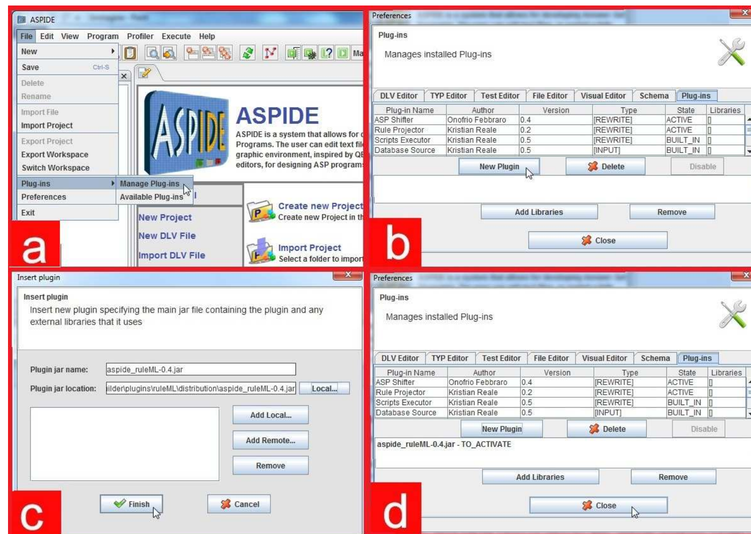


Figure 7.10: Installing a new plugin in *ASPIDE*.

We open the installation dialog by selecting the menu *File* and clicking on *Manage Plug-ins* contained on the sub-menu *Plug-ins* (fig. 7.10a). The dialog shows plugins that are already installed in *ASPIDE* (fig. 7.10b). In the dialog we click on the *New Plugin* button; *ASPIDE* opens a new dialog where we choose, from either the local hard disk or remotely, a JAR plugin file and other needed JAR libraries (if any) (fig. 7.10c). We confirm the operation and the choosen JAR plugins become candidate to be installed (fig. 7.10d); the plugin will be effectively installed only when *ASPIDE* is restarted. The installation process can detect incompatibility issues, missing libraries, and allows the user to enable/disable plugins.

An alternative way to install plugins consists in checking the ones that are available on the web. To install a new plugin in this way, we open a dedicated dialog by selecting the menu *File* and clicking on *Available Plug-ins* contained on the sub-menu *Plug-ins*. We select a url location and click on the *Get Plug-ins* button; a list of plugins will be shown. We select one plugin and install it immediately (fig. 7.11).

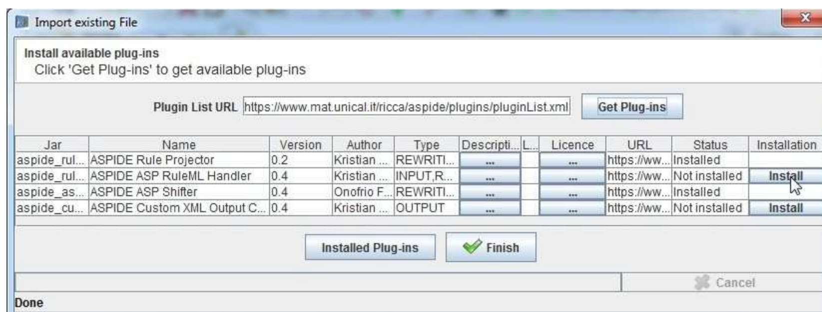


Figure 7.11: Installing a new plugin in *ASPIDE* by checking available plugins on the web.

Chapter 8

Database Management in *ASPIDE*

The high expressiveness of Answer Set Programming allows the modelling of advanced knowledge-based tasks arising in modern application-areas. When the applications of interest become data intensive, it is important to exploit systems like databases in order to access large amounts of data for performing reasoning tasks. However, ASP systems which work in main memory only, cannot be well suited for huge data and, consequently, reasoning tasks need more efficient solutions and more working space. To deal with this problem, some interesting solutions have been proposed and implemented in the DLV^{DB} [88, 87] system, allowing reasoning tasks in mass memory to be performed by exploiting DBMSs. Moreover, since in real-world applications big input data is usually stored in databases, for performing reasoning in those data, tuples must be ported in ASP systems as facts. Also the results of a reasoning task specified by an ASP program (for example a query) is expected to be stored in a database. DLV^{DB} allows tables to be mapped in predicates, tuples to be transformed in ASP facts and update data stored in database with, for example, new inferred tuples[1].

To deal with these mentioned issues regarding databases, in *ASPIDE* we offer graphical solutions for generating mappings between database tables and predicates, managing predicate schemas, and performing database access for easy retreiving data and metadata of tables. As a consequence, users can exploit *ASPIDE* for *Database Management* tasks customized for ASP solutions. Database Management features allow, in a intuitive way, following actions:

- creation/editing of schemas, *TYP files*, and *Import/Export* directives by advanced editors;
- automatic generation of schemas, *TYP files*, and *Import/Export* directives by direct connection to DBMSs;
- rewritings between schemas, *TYP files*, and *Import/Export* directives;
- retrieving tuples to be rewritten to a set of ASP facts.

[1] See Chapter 3 for more details.

Schemas can be defined in the *DLV Text Editor* exploiting specific annotations and in the *Visual Editor* in a graphical way (see Chapter 5). In the follow we deal in-dept with the schema management of *ASPIDE* and describe database interaction. Regarding database, an input/rewriting plugin[2] has been implemented allowing to access to database and perform mappings. In such a way, database oriented applications can be run by setting DLV^{DB} as solver in a Run Configuration and a data integration scenario [62] can be implemented by exploiting these features.

8.1 Schema Management and Table Mappings

In Chapter 3 we overviewed *Import/Export* directives, exploited by DLV, and *TYP files*, exploited by DLV^{DB}, for defining mappings between database tables and predicates. These features are exploited by *ASPIDE* for schema management that allow one to have support for schema definitions, *Import/Export* directives of DLV and fully-graphical composition of *TYP files*. We now describe these features in detail.

8.1.1 TYP Files

The DLV^{DB} solver can manage advanced mappings by exploiting *TYP directives*. These directives are stored in *TYP files* which are passed to the DLV^{DB} solver together with *DLV files*. The solver will perform mappings by retrieving data and storing the inferred data to one or more databases. The grammar of the *TYP directives* is specified in the following:

```
Init − Section ::=
1. USEDB databaseName : username : password [System − Like]?.
System − Like ::=
LIKE [POSTGRES|ORACLE|DB2|SQLSERVER|MYSQL]
Table − Definition (EDB definition) ::=
2. USE tableName [( attribute [, attribute]∗)]?
3. [AS (SQL − Statement)]?
4. [FROM DatabaseName : UserName : Password]?
5. MAPTO predName [(type [, type])]? [ALLOW_APPEND(1)]?.
Table − Definition (IDB definition) ::=
6. CREATE tableName [(attribute [, attribute]∗)]?
7. MAPTO predName [(type [, type])]?
8. [KEEP_AFTER_EXECUTION]?.
Table − Definition (Querydefinition) ::=
9. QUERY tableName.
Final − Section ::=
10. DBOUTPUT DatabaseName : UserName : Password.
```

[2]See Chapter 7 for more details about *ASPIDE* plugins.

```
|
11. OUTPUT [Write − Option]? predName
12. [AS AliasTableName]?
13. IN DatabaseName : UserName : Password.
Write − Option ::=
14. APPEND
|
15. OVERWRITE
```

(1) *ALLOW_APPEND clause is allowed only if the table is on the working database.*

The `USEDB` command (line 1) is used to define connection properties with the working database. For example `USEDB MyDatabase:scott:tiger LIKE POSTGRES.` allows connection to the database `MyDatabase` telling explicitly that the database is POSTGRES in order to let the system know the SQL dialect. The DBMSs that can be specified are POSTGRESQL, MySQL, ORACLE, SQL Server, DB2.

The `USE` directive (line 2) is used to specify a mapping between a table and a predicate of the program. When a `[AS (SQL-Statement)]` (line 3) clause is specified, the `tablename` is expected to be updated by means of the tuples resulting from execution of the `SQL-Statement`.

The `CREATE` directive (line 6) allows definition of the mapping of a logic predicate `predName` with a table; this table will be created in the working database. If the directive `KEEP_AFTER_EXECUTION` (line 8) is present, the table is not deleted at the end of the evaluation. Intuitively, a `USE` directive is intended to map a predicate that the tuples cannot change during processing. In this case the associated table is read-only. On the other hand, a `CREATE` directive creates a new table on the working database in order to store new data inferred for the related predicate. In the case where we need also to add new tuples derived during the computation, we have to explicitly allow for "append" operations. In particular, we have to specify a `USE ... ALLOW_APPEND` directive.

The `QUERY` option (line 9), allows definition of the name of the query table where the results of the query possibly processed by the solver are stored. The `DBOUTPUT` directive (line 10) allows specification of the working database in which the program stores the computation. The `OUTPUT` directive (line 11) allows a specified `predName` only to be stored, that is mapped to a table.

To write *TYP files*, an advanced text editor can be exploited in *ASPIDE*, offering syntax coloring and auto-completion, and an outline identifying, with a tree representation, elements contained in the *TYP file* (fig. 8.1). *ASPIDE* offers also a **Visual Editor** (fig. 8.2) allowing easy composition of *TYP files* in a graphical way.

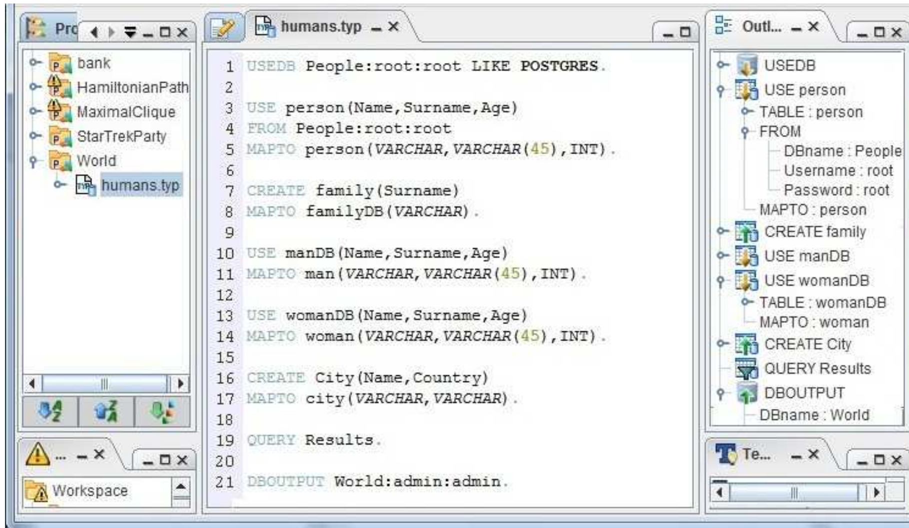


Figure 8.1: Text Editor and outline of a *TYP file*.

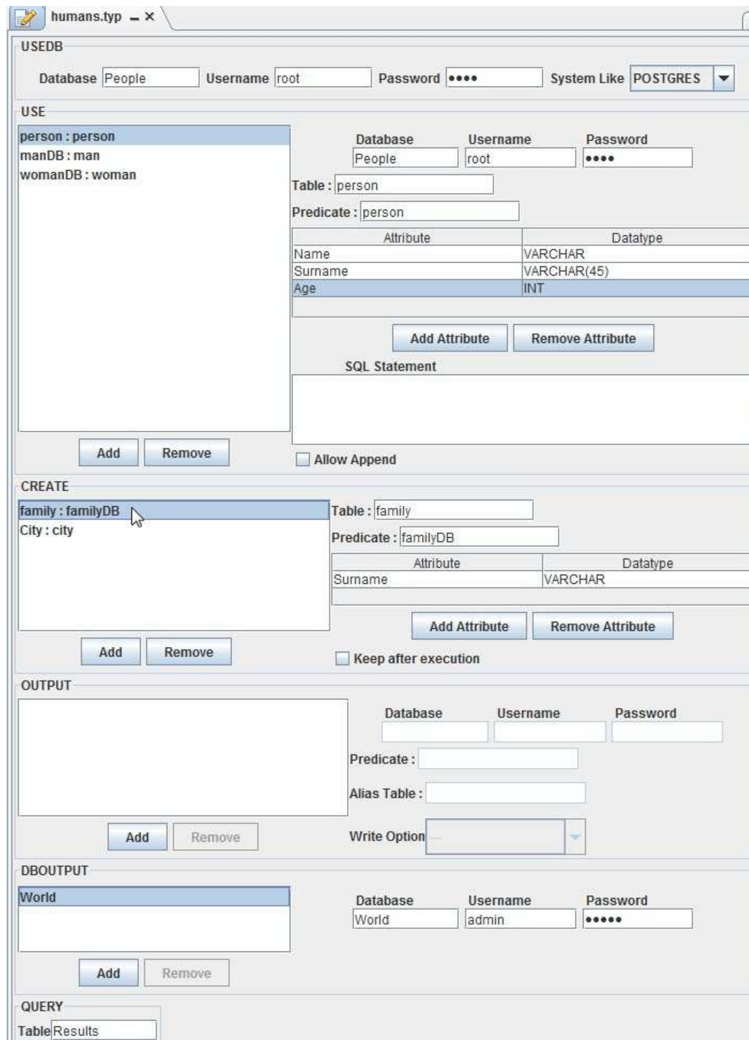


Figure 8.2: Visual Editor of a *TYP file*.

The *Visual Editor* panel is partitioned in sections where each section identifies a *TYP directive* (e.g. `USEDB`, `USE`, `CREATE`, ...). The panel allows easy composition of single directives using text fields and buttons. In the same way as *DLV files*, users can switch from visual mode to textual mode (and vice-versa) everytime he needs thanks to the reverse-reengineering process of the *TYP files* content.

8.1.2 Import/Export Directives

Import/Export Directives are exploited by DLV for mapping database tables in predicates. In particular, the syntax of the *Import* directive of DLV is:

#import(databasename, username, password, query, predname, typeConv).

The name of each imported atom (tuple) is set to predicate `predname`, and is considered as a fact of the program. `typeConv` represents the data conversion rules to be applied for converting datatypes of the database into DLV datatypes.

Example 8.1.1. Supposing information about a person is stored in a specific table of a database, the predicate `person` of Example 8.1.2 can be specified on a *DLV file* in the following way:

#import(People, "root", "root", "SELECT * FROM person",
person, type : CONST, CONST, U_INT).

Here, we are telling the program to populate the predicate `person` with the result of the specified query on the database identified by the ODBC source *People*. Note that attributes names cannot be directly set since the directive allows only simple "conversion" datatypes on attributes to be specified.

□

The *Export* directive generates a new tuple into a table for each new truth value derived for that predicate by the program evaluation. The syntax of the command is:

#export(databasename, username, password, predname, tablename).

`predname` indicates the predicate to be stored and `tablename` the destination table.

These directives are written in *DLV files* and, when the file is passed to DLV, the solver performs database access, retrieves data specified by the *Import* directives and inserts inferred data to tables specified by the *Export* directives.

In the case where a *DLV file* is open in *ASPIDE*, all predicates specified in *Import* directives are marked with a different color because these predicates come from external sources. *ASPIDE* can also detect arity errors in the case where a predicate used in a rule has a different arity compared to the one specified to the *Import* directive.

8.1.3 Schema Annotations

In *ASPIDE* we can define schemas to predicates of ASP programs by introducing specific annotations[3] in *DLV files*. These annotations are listed below:

- `@schema predicateName(attr1:Datatype,attr2:Datatype,...)`: allows one to define a schema for a given predicate by assigning attributes names and datatypes (possibly followed by precision and scale values);
- `@workingDatabase database:username:password`: allows one to set the specific working database by setting a database name, a username and a password;
- `@useTable tableName MAPTO predicateName(attr1:VARCHAR, attr2 :VARCHAR) FROM database:username:password`: allows one to map an external table, defined by *tableName*, in a predicate, defined by *predicateName*, having the specified attributes names with datatypes;
- `@createTable tableName MAPTO predicateName(attr1:VARCHAR) KEEP_AFTER_EXECUTION`: allows one to map an internal predicate, defined by *predicateName*, to an external table, defined by *tableName* and stored in the working database;
- `@dbQuery queryTable`: allows a table name to be specified where results of a query execution must be stored;
- `@outputDatabase databaseName:username:password`: allows setting of the output database where some inferred tables will be stored;
- `@outputTable predicateName AS tableName IN databaseName :username:password`: allows setting of an output table for a specified predicate where results inferred by this predicate will be stored.

The most important annotation is `@schema` which allows to specify, directly, schemas on predicates. *ASPIDE* parses the *schema* annotation and gets the schema of a predicate (attributes names and datatypes).

Example 8.1.2. Suppose we have predicate `person`, with arity 3 (a name, a surname and an age) the following annotation can be used to represent the schema information of the predicate:

```
%@schema person(name : VARCHAR, surname : VARCHAR(45), age : INTEGER)
```

□

The other annotations are used to specify *TYP file* directives directly on *DLV files*; in fact they have the same semantics as *TYP directives* as shown in the follow:

- `@workingDatabase` corresponds to the `USEDB` directive;
- `@useTable` corresponds to the `USE` directive;

[3]Note that annotations introduces specific ASP meta-information which are managed by *ASPIDE*; see Chapter 4 for more details.

- `@createTable` corresponds to the `CREATE` directive;
- `@dbQuery` corresponds to the `QUERY` directive;
- `@outputDatabase` corresponds to the `DBOUTPUT` directive;
- `@outputTable` corresponds to the `OUTPUT` directive of *TYP files*.

In *ASPIDE*, the DLV^{DB} solver can be executed by passing a *DLV file* containing annotations defining *TYP directives*; in this case, by setting an option, a *TYP file* is automatically generated from the annotations and passed to DLV^{DB} as a new file.

Schemas associated with predicates are visualized in the *Outline* panel of *ASPIDE* in two different views:

- on the *Expressions* view, showing, near each predicate with an associated schema, attributes and datatypes, and, as child of the predicate, information stating that the schema was specified by exploiting an annotation;
- on the *Schemas* view, where all schemas contained in the program are visualized in a tree representation. Attributes and datatypes of schemas are also visualized.

Example 8.1.3. Supposing we have defined schemas for predicates `node` and `edge` of the Hamiltonian Path program, Figure 8.3 shows, on the *Expressions* view, the schema spefication for the predicates `node` and `edge` (fig. 8.3a) and, on the *Schemas* view, the specified schemas (fig. 8.3b). □

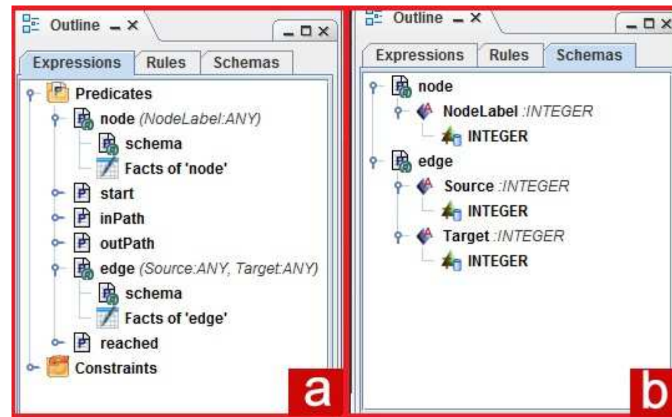


Figure 8.3: Schema visualization in the *Outline* panel.

Since the `@schema` and `@useTable` annotations define a predicate schema, arity errors are detected if a predicate used in a rule has a different arity compared to its schema defined by them. Note that, in a *DLV file* a user can write different *schema* annotations for the same predicates or a mix between an *Import* directive, a `@schema` annotation and a `@useTable` annotation, defining the same predicate. In this case *ASPIDE* checks errors also between them for finding arity problems on schemas or different attributes names and datatypes. Moreover, predicates schemas can be also easy edited in a graphical way by exploiting the *Visual Editor*: see Chapter 5 for more details.

8.2 Database Interaction Plugin

In the previous Section we showed how schema annotations, *Import/Export* directives and *TYP files* can be composed for defining predicate schemas and external sources. However, it is important to note that those operations were done directly in *ASPIDE* "by hand", supposing the existence of the database containing those tables, attributes and datatypes. Suppose now users do not know any details about the database, in this case it could be useful to access directly the interested database and automatically map containing tables in predicates. In *ASPIDE* we have implemented this feature through the definition of a input/rewriting plugin allowing the user to easy direct access tables contained in a database through the ODBC interface. The plugin allows one to:

- access to databases and create mappings between tables and predicates;
- automatically generate *TYP files*, *schema annotations* and *import directives* from the mappings;
- retrieve tuples from mapped tables and transform them in ASP facts.

The architecture of the plugin, depicted in Figure 8.4, is composed by an *Input Module* and a *Rewriting Module*.

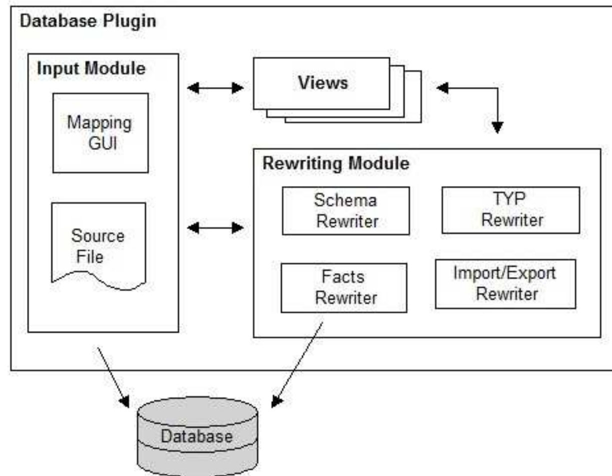


Figure 8.4: Database Plugin Architecture.

The Input Plugin module allows one to create *Source files* where each file represents a database source. The user can open the *Source file* by using the *Mapping GUI*, which is the main visual editor of the plugin where he can perform a database connection, through the ODBC interface, for retrieving tables to be mapped. The user chooses some tables and decide to which predicates they have to be mapped. By exploiting the switch button placed on the top-center of *ASPIDE*, a specific editor is opened in accordance with one of the following rewritings, performed by the *Rewriting Module*, from mappings to:

- Schema annotations;

- *TYP files*;
- *Import/Export* directives;
- ASP facts.

For ASP facts rewriting, a connection to the database is performed to retrieve tuples from the database.

When a table is mapped in predicates, each predicate becomes available, in the *Workspace Explorer*, to be dragged, e.g., in other files. *DLV files* of the same project will check, in this way, schema and arity errors by considering also these predicates. Moreover, DLV text and visual editors highlight the mapped predicates evidencing the mapped predicates using a specific color.

Once a user has created a set of *Source files* for mapping predicates of different databases, he can execute them together with one or more *DLV files*. For this purpose, the user can exploit a Run Configuration and specify, on files, which rewritings he wants to perform before the execution. For example, in the case where the user wants to start execution using the DLV solver, he can choose to rewrite *Source files* in ASP facts, so that tuples are retrieved as facts and passed to the solver. Also in the case where the user wants to exploit the DLV^{DB} solver he can choose to rewrite *Source files* to *TYP files*.

8.3 Use Case: A Data Integration Scenario

In the case where information about a domain is distributed among various sources, a data integration scenario is useful to integrate them into just one global schema which unifies the sources. In more detail, data integration consists in combining data residing at different sources and providing the user with a unified view of them, called *global schema*. Users formulate queries over the global schema, and the system queries the sources in a suitable way, providing an answer to the user; the user is not obliged to have any information about the sources. Recent improvements in Information Technology such as expansion of the Internet and the World Wide Web, have made available to users a large number of information sources which are generally autonomous, heterogeneous and widely distributed. Consequently, information integration has emerged as a crucial issue in many application domains, e.g., distributed databases, cooperative information systems, data warehousing, or on-demand computing.

By exploiting database features, *ASPIDE* can be used as data integration system for integrating information contained in different database sources. In particular, users can:

- define mappings from different database sources;
- create a global schema by defining GAV/LAV mappings in *DLV files*;
- perform a query on the global schema and retrieve query results.

We now show how to exploit *ASPIDE* for data integration by exploiting a use case scenario of data integration. Suppose we want to unify information of two database related to bank branches. The first database is stored in the *MySQL DBMS* and contains table `employees (code, name, role)`. The second database is stored in the *Postgres DBMS* and contains table `emp(code,`

`name)` and `manager(code, name)`. Suppose we want to create a global schema in accordance with the following rules:

$$\mathtt{e(C,N)} :- \mathtt{emp(C,N)}.$$
$$\mathtt{e(C,N)} :- \mathtt{employees(C,N,_)}.$$
$$\mathtt{m(C)} :- \mathtt{manager(C,_)}.$$
$$\mathtt{m(C)} :- \mathtt{employees(C,_,man)}.$$

In this way the predicate `e` will have the tuples containted in both tables `emp` and `employees`, while the predicate `m` will have the tuple contained in `manager` and `employees` by filtering employees that are managers. To define these mappings by exploiting the plugin, we create a new project named *bank* and we define a new source; we name the source *bankPostgres.source* (fig. 8.5). This source has to contain mappings for our database *Postgres*.

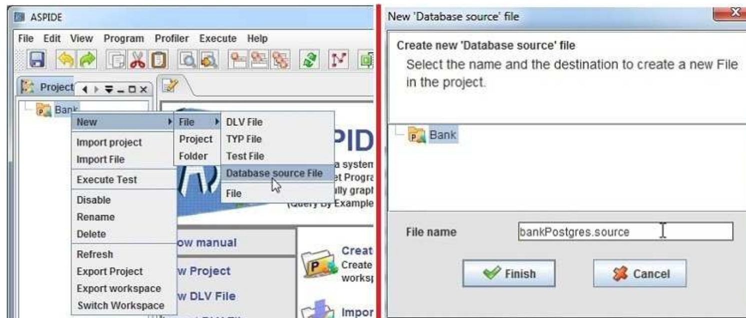


Figure 8.5: Creating a new source.

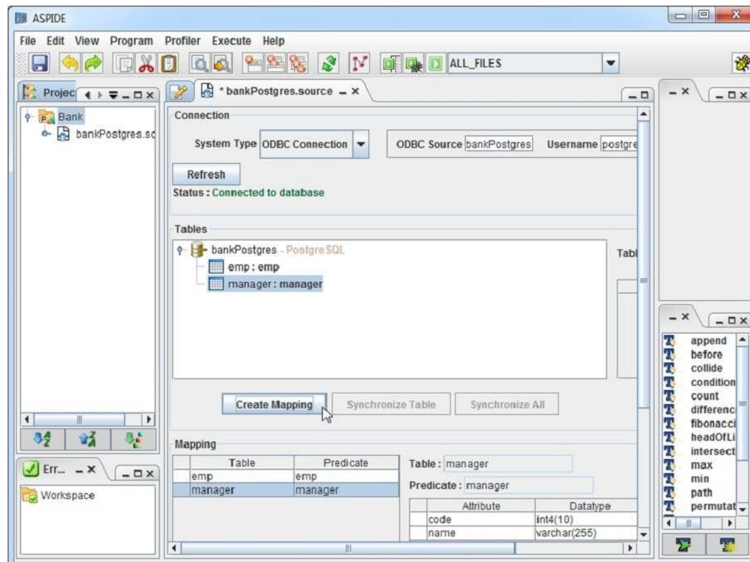


Figure 8.6: Mapping of database tables.

We now connect to the database in order to see tables contained in it and we choose the tables to be mapped (fig. 8.6). To map the tables of *MySQL*, the same actions have to be made. The mapping procedure is finished. If we want to have a look at the possible rewritings that the plugin performs we can click on the *switch* button placed on the toolbar. At every click, the current editor is changed with other editors (fig. 8.7).

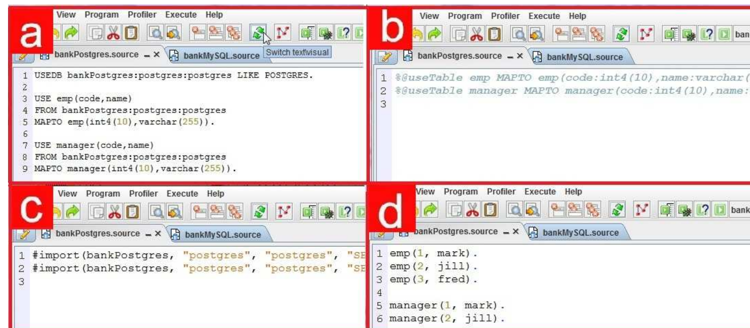


Figure 8.7: Switch among mapping rewritings.

Figure 8.7a shows the *TYP directives* version of the mappings, Figure 8.7b the correspóning annotations, Figure 8.7c the *#import directives* and, finally Figure 8.7d shows facts representing tuples contained in the tables.

The global schema presented above can be easily defined by creating a *DLV File*. Supposing we want to use the *Visual Editor*, we have to drag the predicates contained under the sources in the *Workspace Explorer panel* and make the specific operations (fig. 8.8).

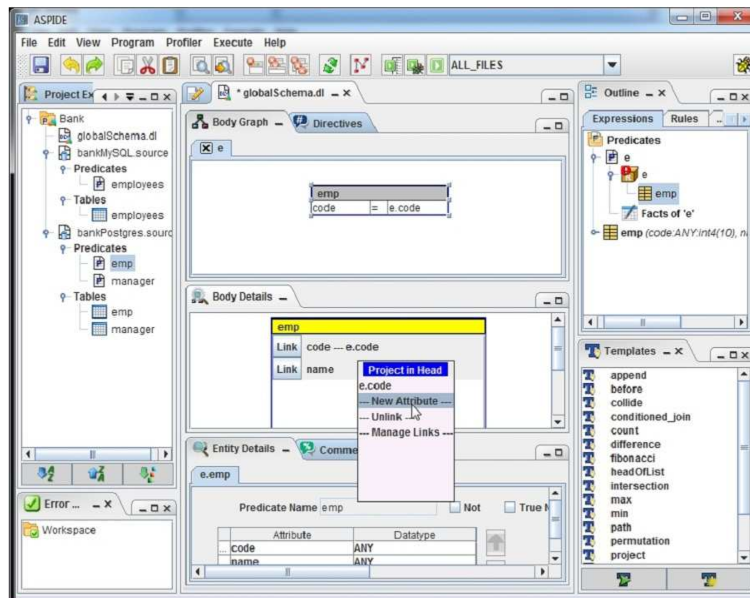


Figure 8.8: Use mapped tables.

We now execute the described program. We create a new *Run Configuration* by including the two sources and the created *DLV file* (fig. 8.9a). The Run Configuration is open where we ask to rewrite the sources before the execution. In this case, we set the facts rewriting for the *bankMySql.source* file and the *TYP* rewriting for the *bankPostgres.source* (fig. 8.9b). As solver we set DLV^{DB} because we are passing a source as *TYP file*. The results are shown in figure 8.10.

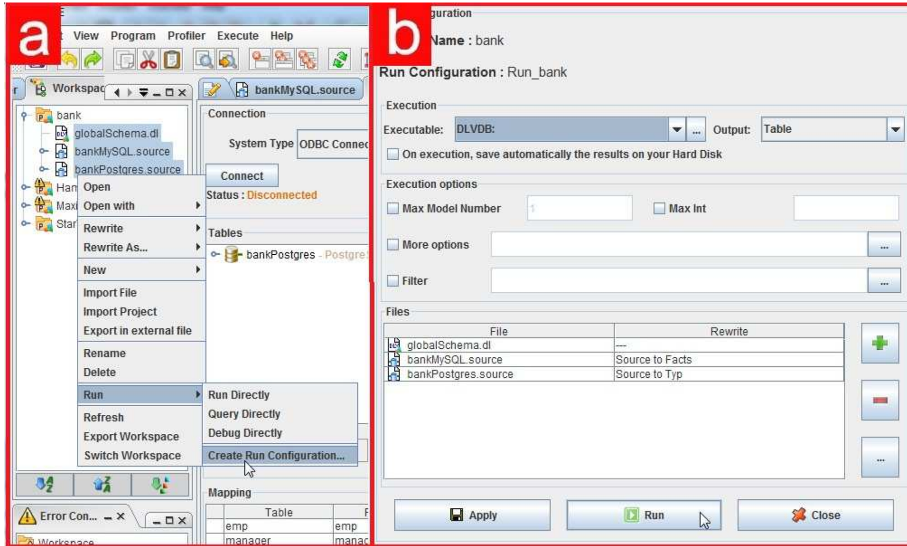


Figure 8.9: Execution of the global schema.

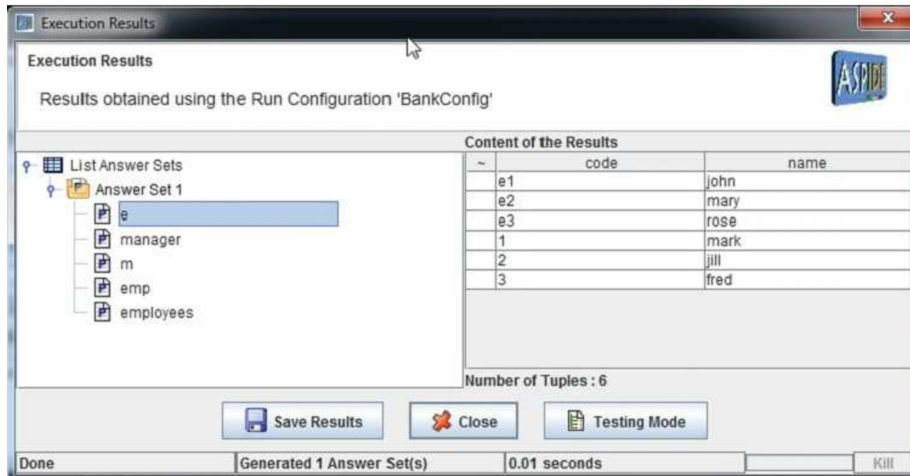


Figure 8.10: Results of the global schema execution.

Chapter 9

Related Work

In the literature, some support for assisted program development of logic programs is present but the available tools do not completely support the entire life-cycle of logic programs development. Despite particular attention being given to environments supporting the language Prolog, they still provide little graphical support for the end-user. Regarding databases systems, owing to the declarative nature of the most-used query language SQL, different environments were introduced to help users with easy writing queries. In the field of ontologies, which is also correlated to logic programming, some tools for ontology definition and querying were also proposed. Regarding ASP, in the last few years different tools for developing ASP programs have been proposed, including editors [54, 74, 85, 75, 73, 80] and debuggers [15, 13, 33].

Subsequently to *ASPIDE*, since developing tools for ASP is a brand-new trend of the ASP community, more advanced IDEs are now under development but they still do not totally implement a wide set of features for helping ASP users in the Software Development Cycle. In summary, development of Integrated Development Environments for declarative and logic programming languages received less attention compared with the wide set of environments available for Imperative Languages.

The enviroments for Prolog, database querying, and ontologies, are closer to *ASPIDE* because of their declarative nature. As as a consequence, comparing the features of *ASPIDE* with the features of these tools makes sense. The aim of this Chapter, in fact, is to compare *ASPIDE* with the available tools for ASP and with the other tools just mentioned. Known IDEs for declarative and logic programming are described in detail and their features are compared with the ones available in *ASPIDE* in order to see which features are offered by our tool that are not offered by the others and vice-versa.

9.1 IDEs for Declarative and Logic Programming Languages

In this Section a description of the most complete and important tools for supporting applications oriented to declarative and logic programming languages, and in particular for ASP, is made. In particular, we describe:

- Logic and Database-based environments;
- Ontology-based environments;
- Prolog-based envoronments;
- Answer Set Programming-based environments.

9.1.1 Logic and Database-based environments

Database oriented declarative programming tools are in general specialized to support the designing of queries. Different methodologies for designing queries were proposed and implemented in a variety of tools. In particular, the methodologies for graphical composition of queries can be generally classified in *Query by Example (QBE)*, *Query by Navigation (QBN) Query by Construction (QBC)*, *Query by Browsing (QBB)*, *Query by Diagram (QBD)*. All these methods offer different approaches to building queries, and the most used, for writing simple queries, is QBE that allows the user to search for database tuples by inserting, as result example, a text string in a form. QBE is clearly easier to learn than formal query languages such as SQL. In the following the tool *Datalog Educational System* is described since it supports in particular the Datalog language for reasoning in deductive databases.

Datalog Educational System

The Datalog Educational System (DES)[1] is a multiplatform implementation of a deductive database system and offers the languages *Datalog*, *Relational Algebra* and *SQL* [45]. The system supports:

- ODBC connections for external relational database management systems (RDBMSs) interoperability;
- Datalog and SQL tracers;
- textual API for external applications;
- declarative debugging of Datalog/SQL queries;
- aggregate predicates;
- test case generation for SQL

The declarative debugger relies on program semantics rather than on the computation mechanism and exploits views for asking the user whether the result of a given view is as expected. The system is implemented on top of Prolog, can be used from a Prolog interpreter and supports predicate persistency in external databases where the processing is directed). Finally, as several ODBC connections are allowed at a time, different predicates can be made persistent in different DMBSs, which allows for interoperability among external relational engines and the local deductive engine. The system was developed for students, so that they can get the fundamental concepts behind a deductive database (exploiting Datalog), Relational Algrebra and SQL as query languages.

[1]`http://www.fdi.ucm.es/profesor/fernan/des/`

9.1.2 Ontology-based environments

An ontology describes concepts and relationships that are important in a particular domain and provides a vocabulary for that domain as well as a computerized specification of the meaning of terms used in the vocabulary. Ontologies, generally can consist in taxonomies, classifications and database schemas and, recently, have been adopted in many business and scientific communities as a way to share, reuse and process domain knowledge. Some tools for designing and managing ontologies were proposed and this paragraph describes someone of them.

OntoDLV

The *OntoDLV* system [80] is an ASP-based system for ontology management and reasoning on top of ontologies. It implements a logic-based ontology representation language, called *OntoDLP*, which is an extension of (disjunctive) ASP with all the main ontology constructs (classes, inheritance, relations and axioms). The language is strongly typed, and includes also complex type constructors, like lists and sets. *OntoDLV* supports a powerful interoperability mechanism with OWL, allowing the user to retrieve information from external OWL Ontologies and to exploit this data in *OntoDLP* ontologies and queries. Moreover, *OntoDLV* facilitates the development of complex applications in a user-friendly visual environment; it is endowed with a persistency-layer for saving information transparently on a DBMS, and it seamlessly integrates the DLV system. Using *OntoDLV*, domain experts can create, modify, store, navigate and query ontologies by exploiting the user-friendly visual environment.

OntoStudio

OntoStudio[2] is a modeling environment for creating and maintening ontologies. It allows the user to exploit comprehensive functions in an intuitive ontology modeling and it is also able to import many structures, schemas and models. Important functions of *OntoStudio* are the mapping tool, which can be used to quickly match heterogeneous structures, and an intuitive graphic rule editor which can be used to model complex correlations between concepts. *OntoStudio* offers also a graphical mapping tool for an easy connection of databases and knowledge bases and the possibility to expand the environment with additional plugins. The system allows one to manage object in the ontology formats of OWL, RDF(S), RIF, SPARQL, F-Logic and ObjectLogic. Created queries can be also exported as a Web service and integrated into any applications. There is also a light version of *OntoStudio*, namely *Web OntoStudio* which can be easily used via browsers; it is ideal for large distributed teams who edit ontologies collaboratively.

Protégé

Protégé[3] provides a suite of tools to construct domain models and knowledge-based applications with ontologies. It allows creation, visualization, and manipulation of ontologies in different representation formats. The *Protégé* platform

[2]`http://www.ontoprise.de`
[3]`http://protege.stanford.edu`

is composed of two main ways of modeling ontologies depending whether the ontologies are classical or Semantic Web based (in the last case the system exploits the Web Ontology Language OWL). For classical ontologies the provided editor supports users in constructing and storing domain ontologies, customizing data entry forms, and entering instance data; for Semantic Web-based ontologies a wide set of user interface elements that can be customized to enable users to model knowledge and enter data in domain-friendly forms. For building knowledge-based tools and applications, *Protégé* can also be extended by exploiting a plugin architecture based on a Java-based Application Programming Interface (API). The architecture allows one to introduce graphical components (e.g., graphs and tables), media (e.g., sound, images, and video), various storage formats (e.g., RDF, XML, HTML, and database back-ends), and additional support tools (e.g., for ontology management, ontology visualization, inference and reasoning, etc.).

9.1.3 Prolog-based envoronments

SWI-Prolog[4] is an open source implementation of the programming language Prolog. The tool, usable by exploiting a command line, offers the following features for prolog programming:

- libraries for constraint logic programming;
- multi-threading;
- unit testing;
- a graphical user interface;
- interfacing to Java, ODBC and others;
- a web server, SGML, RDF, RDFS, developer tools (including an IDE with a GUI debugger and GUI profiler), and extensive documentation.

Several environments for Prolog programming use the *SWI-Prolog* tool for offering new suitable graphical interfaces capable to exploit the power of *SWI-Prolog*.

In this paragraph, different tools for Prolog, both oriented and non-oriented to SWI-Prolog, are described.

J-Prolog

J-Prolog[5] is a very simple editor for SWI-Prolog providing syntax highlighting, an embedded Prolog interpreter and other simple features like text coloring and a console view for textual results visualization. Ways for debugging, testing and profiling are also offered and can be activated by exploiting buttons.

[4] `http://www.swi-prolog.org/`

[5] `http://www.trix.homepage.t-online.de/JPrologEditor`

ProDT

Prolog Development Tools (ProDT)[6] is a Prolog Integrated Development Environment that attempts to offer rich functionalities like the Eclipse Java platform, by giving the developer a single environment where it can control the Prolog project development from code edition, test execution, debugging, and more. The environment is an Eclipse plugin and, consequently, takes advantage of its already existent features. A ProDT project is organized in *Source Folders* containing directories and project source files. The editor offers highlighting for predicates and for matching brackets so that a user has an easy way to identify which bracket corresponds to another. By pressing Ctrl-SPACE in the editor a list of rules and predicates defined in the file will be displayed, followed by the list of predicates and operators built-in of the Prolog interpreter. By exploiting the enviroment, the user can configure interpreters by specifying also which one of them will be default; also wizards allow one to easily set interpreters. The outline is a schematic listing of the predicates and rules defined in a file which can be used to have a general overview of a file and as a navigation tool. The outline is composed of two different views: *Outline View*, offering a fast way to jump the cursor of the editor in the corresponding position by selecting an element in the Outline View, and a *Quick Outline*, which can be activated, as popup view. Detected compilation errors are marked in different views like the *Outline* and *Quick Outline* views, used for direct navigation to the problematic predicate, the *Problems* view, collecting the list of problems and warnings of all the projects, and the *editor* view that marks errors in the left and right side bars and by underlining the line containing the error. Source compilation is integrated with this IDE using the Eclipse builder mechanism. The environment also offers a *Console* View showing results of the interpreter execution. By using the console, the user can exploit an autocompletion mechanism for displaying the predicates loaded in the current file; finally, the user can save its content on a file using the toolbar of the view.

PDT

The Prolog Development Tool (PDT)[7] is a Prolog IDE provided as a plugin for the Eclipse Platform. The system offers a *Project Explorer* showing source and external files, and a highlighting for entry points and consulted files. The editor offers syntax highlighting, singleton variable highlighting, code completion, errors/warnings annotations, breakpoints for the debugger and keyboard shortcuts. *Outline* and *Quick Outline* can be exploited; in particular, the Quick Outline displays predicate documentation. The console of the system offers bash-like completion and command history and allows one to interact with multiple Prolog processes, for direct access to code with errors in case of errors and warnings, and to interact with SWI-Prolog tools. A reusable Java API for communiating with SWI-Prolog can be also exploited. A Context View visualizes call relations, dead code and predicate properties (exported, dynamic, ...). Finally and also importantly, the system allows one to exploit debugging at source level by introducing breakpoints (connected to PDT editor), to see variable bindings and to use the SWI-Prolog Profiler.

[6] `http://prodevtools.sourceforge.net`

[7] `http://roots.iai.uni-bonn.de/research/pdt`

ProClipse

ProClipse [12] is a Prolog plugin for Eclipse providing many features such as semantic-aware syntax highlighting, outline view, error marking, content assist, hover information, documentation generation, and quick fixes. The system, mainly, offers syntax highlighting by exploiting full syntactical and semantical analysis, and other important tools for helping the user to compose Prolog programs. In particular, an *Outline* view is an overview of the Prolog file or module, containing exported or non-exported predicates, and import directives. Each element in the outline view can be used to access the respective line code quickly. Syntax and semantic errors are highlighted in the editor view by underlining the erroneous part of the source code. A dedicated view lists all errors of all Prolog files in a Prolog project and can also be used to recall an erroneous source code line directly. A *Content Assist* tool offers sensitive proposals to automatically complete the word the developer types and can also be used to retrieve information about predicates. Quick fixes offer the possibility of auto-correcting an erroneous source code directly. A suitable set of fixes is shown to the user. For automatic generation of documentation, the system offers *PrologDoc* used to attach a documentation to a predicate; a comment containing PrologDoc entries must be written above the first appearance of a clause defining this predicate. Documentation can also be attached to entire modules; in this case the PrologDoc comment field must be written above the module definition. Another important feature regards *Text Hovering* used for quick retrieving of information of a lexical token in the editor view. For instance, if a developer wants to know how a certain predicate has been defined, he or she has simply to point the cursor at the predicate.

Amzi! Prolog + Logic Server

Amzi! Prolog[8] is an Eclipse plugin composed of core components that are available on all platforms, and additional components that are platform dependent. The system allows organizing files in projects, running, debugging and advanced execution that exploits compiling, linking and running. The editor offers color highlighting of system predicates, comments, constants, mathematical operators and functions. Syntax errors are marked and error messages are inserted in a Tasks View. By pressing CTRL-SPACE, system predicate names, constants, mathematical operators and functions are suggested with content assistant describing system predicates and their arguments. Text hovering is also offered by the system.

The Logic Server part allows one to deploy projects as part of a stand-alone application. It can run by connecting directly to software written in languages such as Java, C++, VB, C# and Delphi.

9.1.4 Answer Set Programming based environments

In the following, known tools for Answer Set Programming are described.

[8] `http://www.amzi.com/products/prolog_products.htm`

SeaLion

SeaLion [74] is an IDE, developed as Eclipse plugin, for ASP supporting a large extent of the languages of DLV and Gringo. The editor provides syntax highlighting, syntax code checking, error reporting, error highlighting, and a program outline. External tools like answer set solvers can be managed by defining, also, arbitrary pipes between them (this feature is important in case of using separate grounders and solvers). To run an answer set solver on the created programs, run congurations can be defined in which the user can choose input files, a solver, command line arguments and output-processing strategies. Resulting answer sets can be either parsed and stored in a view, or can be displayed in the Eclipse console view. An important feature of the system is the capability for visualisation and visual editing of interpretations through the system *Kara* [61]. This follows ideas from the visualisation tools like IDPDraw [92] where a visualisation program is joined to an interpretation to be visualised. However, the editing feature of SeaLion allows one also to graphically manipulate the interpretations under consideration. SeaLion exploits program comments to extract meta-statements like assigning properties to elements; an example of meta-statements is rule name assignments.

Visual DLV

Visual DLV [75] is a graphical programming environment which integrates several tools for developing, testing and executing DLV programs in a quite simple way. It helps programmers during the development phases, supports the interaction with external DBMS and features a naïve debugging tool. The system organizes files in a project for gathering in a single logical unit several DLV program files. The editor helps the user with automatic completion that suggests to the user how to complete the portions of programs he is writing. In the editing phase, dynamic syntax checking goes into action for checking syntactical correctness of the program by warning the user in case of errors. The system is able to interact with databases, so it allows the user to easily, and graphically, specify which input data resides in external databases, and which parts of the program output must be permanently stored in a database. A Run Configuration for executing programs can be set with execution options for DLV and the results can be visualized on the same environment. Finally, the systems allows the user to debug programs, an activity consisting in interacting with DLV in order to understand why a program does not produce the expected output.

APE

AnsProlog Environment (APE) [85] is an Eclipse plugin that supports users in the development phases of ASP programs in the lparse/gringo language. It offers a general user interface for running SMODELS, exploiting LPARSE, and for generating/visualizing dependency graphs. The interface defines a working directory, a current editor with open files, different views of the active program like the syntactic outline or the dependency graph, a console with either the answer sets of the program (in case SMODELS is called) or errors/warnings that the system has detected in the active program. The editor offers syntax highlighting and automatic syntax checking.

iGROM

iGROM [54] is an IDE, developed as Eclipse plugin, for ASP, in particular it supports DLV and SMODELS programs. It can be also extended with user-defined editors for such languages. The system allows the user to organize programs in projects and offers features like syntax highlighting for DLV and its dialects, errors detection both syntactic and semantic like safety, testing, debugging and run configurations management. Moreover, the Eclipse platform offers extension mechanisms and Subversion support.

VIDEAS

VIDEAS [73] focuses its attention on the separation of answer set programs in three parts: *facts*, representing some knowledge base, *rules* for some reasoning task, and *constraints* for asserting integrities on facts and rules. Building these three parts requires users to have ASP skills. The solution proposed by VIDEAS regards the separation of these parts with the purpose of assigning the facts representation (with integrity constraints on facts) to people having modeling skills; ASP skills are not needed. Graphical visualisation is easy for human users and models can be easy transformed into executable code in an automatic way. This mean that there are no inconsistencies between the models and the code. Using VIDEAS, users can define facts (with integrity constraints) by exploiting E-R diagrams; the E-R diagrams are translated into facts and constraints and included in an ASP program for further execution in ASP solvers. The purpose of VIDEAS is to encourage programmers to construct their programs as data models in order to draw attention to design decisions.

DLV!sual

DLV!sual[9] is a GUI frontend for DLV that allows one to experiment with DLV easily. The system, indeed, allows users to graphically-browse the answer sets in a comfortable way. If the user changes selected files, the output will be automatically refreshed.

Digg

Digg[10] is a simple Java application conceived for learning ASP and experimenting with DLV. It offers a minimal set of tools like a text editor, allows for token pair highlighting and offers a textual result visualization.

9.2 Comparison with *ASPIDE*

This Section describes detailed comparisons between the features of *ASPIDE* and the features offered by the systems described in the previous Section. Different tables that evidence different perspectives of comparison are presented. In particular, a table shows general features comparisons whereas other tables compare some single tools commonly offered by all the IDEs, like text editor and visual editor. Finally, also languages that are supported by the IDEs are

[9] `http://thp.io/2009/dlvisual`
[10] `http://www.ezul.net/2010/09/gui-for-dlv.html`

compared. For each table, a check symbol indicates that a system provides (in a more or less sophisticated way) a feature.

Table 9.1 shows a comparison of general desirable features.

General Features	**ASPIDE**	**OntoDLV**	**OntoStudio**	**Protégé**	**VIDEAS**	**DES**	**SeaLion**	**iGrom**	**DLV!SUAL**	**VISUAL DLV**	**Digg**	**APE**	**J-Prolog**	**ProDT**	**PDT**	**ProClipse**	**Amzi! Prolog**
Text Editor	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
Visual Editor	V	V	V	V	V												
Project Management	V		V	V	V	V	V	V				V		V	V	V	V
File Content Outline	V	V	V	V	V		V			V		V		V	V	V	V
Test Suite	V		V	V		V	V							V			
Integration with databases	V		V	V		V				V						V	
Debugger	V		V	V		V				V			V	V	V		V
Command line						V							V	V	V	V	V
Visual Dependency Graph	V		V	V								V					
Profiler/Tracer	V			V		V							V	V	V		
Global Error Console	V	V	V				V	V		V		V		V	V	V	V
Textual Result Visualization	V	V			V	V	V	V	V	V	V	V	V	V	V	V	V
User Friendly Result Visualization	V	V	V	V			V										
Visual Workflow Definition	V																
Detect Error on editing	V		V	V			V	V		V				V	V	V	V
Dynamic Layout	V		V	V	V		V	V				V		V	V	V	V
Refactor variables and predicates	V		V	V	V		V										
Configuration of the execution	V		V	V		V	V	V	V	V		V		V	V		V
Data Integration	V		V	V													
Datatype Management	V	V	V	V		V				V							

Table 9.1: System Comparison for General Features.

We first note that *ASPIDE* is the most complete proposal, followed by the products *OntoStudio* and *Protégé*; and, if we restrict our attention to competing systems tailored for ASP, *SeaLion* and *APE* strictly follow *ASPIDE*.

Note that, execution results are reported by most systems only in a textual form whereas *ASPIDE*, *OntoDLV*, *OntoStudio*, *Protégé* and *VIDEAS* offer a graphical view of them in intuitive tables. The outline of the program is often missing, and the execution of systems/solvers is not handled in an effective way. Moreover, only *ASPIDE*, *OntoStudio*, *Protégé* and *APE* show the dependency graphs in a graphical way, and the detection of errors during the editing phase, as well as (some form of) debugging and testing are offered by few systems. Interaction with databases, often required by applications, is supported by only 6 systems out of 17 (data integration only by three).

Conversely, text editing is supported by all systems and, in order to provide a more precise picture regarding this central feature, the analysis has been deepened by considering also more advanced editing functionalities and support

for project management. Table 9.2 evidences that almost all systems support multi-files management but fewer systems support project management.

Project Management Features																	
Feature	**ASPIDE**	**OntoDLV**	**OntoStudio**	**Protégé**	**VIDEAS**	**DES**	**SeaLion**	**iGrom**	**DLV!SUAL**	**VISUAL DLV**	**Digg**	**APE**	**J-Prolog**	**ProDT**	**PDT**	**ProClipse**	**Amzi! Prolog**
Multi Project	V		V	V	V		V	V				V		V	V	V	V
Multi File	V		V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
Sub Folder Organization	V		V		V		V	V				V		V	V	V	V

Table 9.2: System Comparison for the Project Management Features.

Text Editor Features																	
Feature	**ASPIDE**	**OntoDLV**	**OntoStudio**	**Protégé**	**VIDEAS**	**DES**	**SeaLion**	**iGrom**	**DLV!SUAL**	**VISUAL DLV**	**Digg**	**APE**	**J-Prolog**	**ProDT**	**PDT**	**ProClipse**	**Amzi! Prolog**
Text Coloring	V			V		V	V	V	V			V	V	V	V	V	V
Parenthesis pair highlighter	V								V			V		V	V	V	
Token pair highlighter	V						V		V		V	V		V	V	V	
Undo/Redo	V				V		V	V	V			V	V	V	V	V	V
Syntactic Error highlighter	V						V	V				V		V	V	V	
Find/Replace	V				V	V	V	V	V					V	V	V	V
Quick Fix	V													V	V	V	
Auto completion	V			V						V		V		V	V	V	V
Code Templates	V			V						V				V	V		
Dynamic Code Template Definition	V			V											V		
Code Annotation	V			V			V										
Automatic Code intentation	V																
Text Hover (quick info of predicates)																V	
Code Documentation (like JavaDoc)	V						V									V	

Table 9.3: System Comparison for the Text Editor Features.

Table 9.3 summarizes features offered by the text editors of the systems; it evidences that also in this case *ASPIDE* is the system offering more features. Surprisingly, *OntoStudio* lacks advanced text editing features, which are conversely provided by the more mature environments for Prolog. It is worth noting that, systems based on the Eclipse platform, which eases the development of text editors, provide quite a number of editing and project management features (see, e.g., *APE* and *SeaLion*). In general, many existing tools provide syntax coloring, but few systems for ASP support code completion and quick fix of er-

rors. It is quite strange that very basic features like undo/redo and find/replace are not supported by all systems. Code annotation is offered only by *ASPIDE*, *Protégé* and *SeaLion*. The *SeaLion* system in particular supports the language *Lana* [90] which, similarly to *ASPIDE*, allows one to add meta-information to rules, terms and atoms. By exploiting the *Lana* language, documentation for ASP programs and test cases for selected blocks of rules can be automatically generated.

Although every considered system supports a textual editor, the systems which offer a complete graphical editing environment for writing programs (or part of them) are *ASPIDE*, *OntoDLV*, *OntoStudio*, *Protégé* and *VIDEAS*. Thus, in Table 9.4 we report only the systems allowing for graphic composition of programs.

Visual Editor Features																	
Feature	**Building of Rules**	**Building of Queries**	**Editing of facts/instances**	**QBE/Diagram like style**	**Collapsing predicates**	**Join Attributes**	**Templates**	**Disjunction**	**Aggregates**	**Built-ins**	**Constraint**	**Weak Constraint**	**True negation**	**Negation as failure**	**Basic arithmetic function**	**Error Management**	**Reverse engineering(text/visual)**
ASPIDE	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
OntoDLV		V	V	V		V			V					V			V
OntoStudio	V	V	V	V		V	V		V	V	V			V	V	V	V
Protégé			V	V			V		V	V	V			V	V	V	V
VIDEAS			V	V			V			V	V						

Table 9.4: System Comparison for the Visual Editor Features.

Focusing on ASP-based systems, *ASPIDE* easily beats *OntoDLV*, which supports only queries, and VIDEAS, which supports only modeling of schemas and relations between schemas. VIDEAS does not allow logic rules to be drawn up but, on the other hand, it supports a more advanced schema definition compared with *ASPIDE*. An interesting plan that can be made could be a fusion between VIDEAS and the *Visual Editor* of *ASPIDE* in order to have the possibility of exploiting the same environment for both schema modeling and rules definition. *OntoStudio*, which supports a different logic language, clearly misses many ASP-specific constructs, but provides a rich environment that supports ontology constructs. *Protégé* offers many features but it also lacks a graphical way to build logic rules or queries.

Table 9.5, finally, summarizes languages and system/solvers supported by the IDEs.

System Compare for Supported Languages and Solvers																	
Language/Solver	**ASPIDE**	**OntoDLV**	**OntoStudio**	**Protégé**	**VIDEAS**	**DES**	**SeaLion**	**iGrom**	**DLV!SUAL**	**VISUAL DLV**	**Digg**	**APE**	**J-Prolog**	**ProDT**	**PDT**	**ProClipse**	**Amzi! Prolog**
DLV/DLVDB	V				V		V	V	V	V	V						
Lparse					V		V	V				V					
Datalog	V	V			V	V	V	V	V	V	V	V					
Prolog													V	V	V	V	V
SQL						V											
DLP^+		V															
DLV^K								V									
ASP RuleML	V																
OWL			V	V													
RDF(S)			V	V													
RIF			V														
Object Logic			V														
Source-to-source transformation	V		V	V		V											

Table 9.5: System Comparison for Supported Languages and Solvers.

All the systems for ASP (and also OntoDLV) support the Datalog language while only *DES* supports the SQL language. Compared with *OntoStudio* and *Protégé*, *OntoDLV* does not support well-known languages for ontologies like OWL; it supports Datalog and DLP^+ only. The Source-to-Source transformation feature merits a particular mention, which allows for translating one language to another. Actually, despite *OntoStudio* and *DES* supporting this feature for specific translation (e.g. from Dataolog To SQL or between F-Logic and RDF), *ASPIDE* allows the introduction of user-defined plugins for extending it with rewriters useful for the translating process (see Chapter 7). Currently *ASPIDE* plugins allow translation between ASP RuleML and the syntax of ASP and, moreover, for convertion of tuples of an external database to ASP facts.

The feature-wise comparison with existing environments for developing logic programs clearly shows that *ASPIDE* is a step forward in the present state of the art of tools for ASP programs development.

Chapter 10

Conclusion

The most diffused programming languages always come with the support of software development tools. These tools are often collected in powerful IDEs that significantly simplify both programming and maintenance tasks. In this thesis we have presented *ASPIDE*, an advanced Integrated Delelopment Environment supporting the entire life-cycle of the ASP software development process.

ASPIDE is the result of a careful study and analysis of existing IDEs for imperative programming languages and of tools which perform graphical database queries. The features offered by *ASPIDE* are summarized in the following:

- **Text Editor**: *ASPIDE* offers dynamic syntax highlighting, on-line syntax correction, auto-completion, code-templates and so on. Moreover, annotations have been also introduced for annotating rule names, predicates schemas and database connectivity;

- **Error Detecting**: Syntax errors are automatically detected, signaled directly to the editor and collected to an Error Console. Errors can be managed by applying quick fixes;

- **Execution and presentation of results**: external solvers are called from *ASPIDE* and the results are shown to users in: (i) tabular form; (ii) a console window; (iii) a graphical way; (iv) a custom way by exploiting an output plugin. A pre-configuration of the execution can be set that allows users to specify files to be executed and solver options. For query execution, having different reasoning modes (brave and cautious reasoning) [30], results are displayed in a comfortable view and a specific output for Epistemic queries [48, 49] is available;

- **Unit Testing**: the crucial task of testing ASP programs has received less attention in the literature and the current proposals [56, 57, 72, 90] do not support users on programs development. *ASPIDE* offers a solution to test ASP programs consisting in a framework inspired by the JUnit framework for Java. For this purpose a testing language has been defined and the concept of "Unit" in ASP programs has been introduced;

- **Visual Editor**: Since ASP lacked a full graphical tool for ASP programs composition, we have defined a visual language, inspired by QBE editors,

which supports all the constructs available in ASP, and we have implemented a Visual Editor which exploits the language. In this way users can draw ASP programs on the screen in a full graphical environment;

- **Extensibility via Plugins**: in real-world applications input data is usually not encoded in ASP and, during the development of an ASP program, the developer might need to apply a rewriting to some rules", e.g., by applying magic sets, disjunctive rule shifting, etc., for optimizing performance. To deal with these issues we have implemented an SDK which allows users to introduce plugins in *ASPIDE* which allow one to deal with new imput formats, performing program rewriting and even customize the format of solver results;

- **DBMS access**: database interactions allows import of schemas and metadata, data retrieval, definition of mappings between predicates and database tables, exploitation of the language directives of DLV^{DB} [88] and interaction with it. *ASPIDE* offers a graphical interface that helps to interact with databases in an intuitive way. For example, the user can easily map tables in predicates in order to import facts from the table. With these features, a data integration scenario [62] can be implemented;

- **Additional Features** : we also implemented the following features to make aspide a complete IDE:

  - *Projects organization*: organizes programs in files and folders and helps programmers when a software is big and needs some project organization;
  - *Outline navigation*: *ASPIDE* creates a graphic outline showing language statements. By exploiting the outline the user can directly access to a specific line of code;
  - *Debugging*: since many solutions were proposed in the literature for debugging ASP programs, we give a contribution by embedding the existing debugging tool *spock* [15] and providing a user-interface;
  - *Tracing* and *Profiling*: we provide a contribution of profiling by embedding the graphical tool proposed in [20] which allows the DLV solver to be traced in the execution phase;
  - *Workflow Execution*: we have introduced a prototypical graphic tool for building an execution process consisting in combining several/system calls or for piping results between solvers/files;
  - *Dependency Graph*: the system offers several graphical variants of the dependency graph of ASP programs.

Related implementation of IDEs for ASP came only subsequently to *ASPIDE* and currently there are other IDEs under development and improvement [54, 74, 85]. Features comparisons of *ASPIDE*, made in Chapter 9, among these IDEs and other IDEs for Prolog and ontologies, evidenced that *ASPIDE* is the most complete and powerful available in the state-of-the-art of IDEs for ASP.

ASPIDE was the object of several publications in international conferences, in particular in the following [41, 36, 39, 38, 42, 37, 35, 40].

Bibliography

[1] Mario Alviano. The maze generation problem is np-complete. In *ICTCS*, pages 12–18, 2009.

[2] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[3] Yuliya Babovich and Marco Maratea. Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. Available on `http://www.cs.utexas.edu/users/tag/cmodels.html`, 2003.

[4] Marcello Balduccini, Michael Gelfond, Richard Watson, and Monica Nogeira. The USA-Advisor: A Case Study in Answer Set Planning. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *LNCS*, pages 439–442. Springer, 2001.

[5] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[6] Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[7] Chitta Baral and Michael Gelfond. Reasoning Agents in Dynamic Domains. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.

[8] Chitta Baral and Cenk Uyan. Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *Lecture Notes in AI (LNAI)*, pages 186–199. Springer Verlag, 2001.

[9] Victor A. Bardadym. Computer-Aided School and University Timetabling: The New Wave. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling, First International Conference 1995*, volume 1153 of *LNCS*, pages 22–45. Springer, 1996.

[10] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[11] Rachel Ben-Eliyahu and Luigi Palopoli. Reasoning with Minimal Models: Efficient Algorithms and Applications. In *Proceedings Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 39–50, 1994.

[12] J. Bendisposto, I. Endrijautzki, M. Leuschel, and D. Schneider. A Semantics-Aware Editing Environment for Prolog in Eclipse. In *Proc. of WLPE'08*, 2008.

[13] Martin Brain and Marina De Vos. Debugging Logic Programs under the Answer Set Semantics. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation*, Bath, UK, July 2005.

[14] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. Debugging asp programs by means of asp. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483 of *Lecture Notes in Computer Science*, pages 31–43, Tempe, Arizona, May 2007. Springer Verlag.

[15] Martin Brain, Martin Gebser, Jorg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. That is Illogical Captain! The Debugging Support Tool spock for Answer-Set Programs: System Description. In Marina De Vos and Torsten Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 71–85, 2007.

[16] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[17] Francesco Calimeri and Giovambattista Ianni. Template programs for Disjunctive Logic Programming: An operational semantics. *AI Communications*, 19(3):193–206, 2006.

[18] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third answer set programming system competition, since 2011. `https://www.mat.unical.it/aspcomp2011/`.

[19] Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition: Preliminary report of the system competition track. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2011.

[20] Francesco Calimeri, Nicola Leone, Francesco Ricca, and Pierfrancesco Veltri. A Visual Tracer for DLV. In *Proc. of SEA'09*, Potsdam, Germany, September 2009.

[21] Claudio Cancinos. Prolog Development Tools - ProDT. `http://prodevtools.sourceforge.net`.

[22] Potassco answer set solving collection. `http://potassco.sourceforge.net/`.

[23] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, A. Nico Habermann, and Edited Peter Fritzson. Overview of software development environments.

[24] Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz. Viewing a programming environment as a single tool. *SIGSOFT Softw. Eng. Notes*, 9(3):49–56, April 1984.

[25] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, Acapulco, Mexico, August 2003. Morgan Kaufmann Publishers.

[26] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczyński. The second answer set programming competition. In *LPNMR*, pages 637–654, 2009.

[27] A. Dovier and E. Erdem. Report on application session @lpnmr09, 2009. `http://www.cs.nmsu.edu/ALP/2010/03/report-on-application-session-lpnmr09/`.

[28] Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-Driven Disjunctive Answer Set Solving. In Gerhard Brewka and Jérôme Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 422–432, Sydney, Australia, 2008. AAAI Press.

[29] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[30] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[31] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172:1495–1539, August 2008.

[32] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. 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.*

[33] Omar El-Khatib, Enrico Pontelli, and Tran Cao Son. Justification and debugging of answer set programs in ASP. In Clinton Jeffery, Jong-Deok Choi, and Raimondas Lencevicius, editors, *Proceedings of the Sixth International Workshop on Automated Debugging*, California, USA, September 2005. ACM.

[34] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In José Júlio Alferes and João Leite, editors, *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in AI (LNAI)*, pages 200–212. Springer Verlag, September 2004.

[35] Onofrio Febbraro, Giovanni Grasso, Nicola Leone, Kristian Reale, and Francesco Ricca. Datalog development tools. In Pablo Barcel and Reinhard Pichler, editors, *Datalog in Academia and Industry*, volume 7494 of *Lecture Notes in Computer Science*, pages 81–85. Springer Berlin / Heidelberg, 2012.

[36] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Unit testing in aspide. *CoRR*, abs/1108.5434, 2011.

[37] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. ASPIDE the Integrated Development Environment for Answer Set Programming: Progress Report. In *Proceedings of 14th International Workshop on Non-Monotonic Reasoning (NMR'12)*, Rome, Italy, July 8 – July 10 2012.

[38] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Extending aspide with user-defined plugins. In *CILC*, pages 236–240, 2012.

[39] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Unit testing in aspide. *(LNAI) In Printing*, 2012.

[40] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. A Visual Interface for Drawing ASP Programs. In *Proc. of CILC2010*, Rende(CS), Italy, July 2010.

[41] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: Integrated Development Environment for Answer Set Programming. In James Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning — 11th International Conference, LPNMR'11, Vancouver, Canada, May 2011, Proceedings*, volume 6645 of *Lecture Notes in AI (LNAI)*, pages 317–330. Springer Verlag, May 2011.

[42] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. Testing ASP programs in ASPIDE. In *Proc. of CILC2011*, Pescara, Italy, sep 2011.

[43] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census Data Repair: a Challenging Application of

Disjunctive Logic Programming. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 561–578. Springer, 2001.

[44] G. Friedrich and V. Ivanchenko. Diagnosis from first principles for workflow executions. Technical report, Alpen Adria University, Applied Informatics, Klagenfurt, Austria, 2008. http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf.

[45] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[46] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 386–392. Morgan Kaufmann Publishers, January 2007.

[47] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Mirosław Truszczyński. The first answer set programming system competition. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483 of *Lecture Notes in Computer Science*, pages 3–17, Tempe, Arizona, May 2007. Springer Verlag.

[48] Michael Gelfond. Strong introspection. In *Proceedings of the ninth National conference on Artificial intelligence - Volume 1*, AAAI'91, pages 386–391. AAAI Press, 1991.

[49] Michael Gelfond. New semantics for epistemic specifications. In *LPNMR'11*, pages 260–265, 2011.

[50] Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence*, 138(1–2):3–38, 2002.

[51] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[52] Giovanni Grasso, Salvatore Iiritano, Nicola Leone, and Francesco Ricca. Some DLV Applications for Knowledge Management. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2009*), volume 5753 of *Lecture Notes in Computer Science*, pages 591–597. Springer, 2009.

[53] Giovanni Grasso, Nicola Leone, Marco Manna, and Francesco Ricca. *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, volume 6565 of *Lecture Notes in AI (LNAI)*. Springer Verlag, 2010.

[54] iGrom. iGrom on sourceforge, 2010. `http://igrom.sourceforge.net/`.

[55] Oliver Jack. *Software Testing for Conventional and Logic Programming.* Walter de Gruyter & Co., Hawthorne, NJ, USA, 1996.

[56] Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. On testing answer-set programs. In *Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 951–956, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

[57] Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Random vs. structure-based testing of answer-set programs: An experimental comparison. In James P. Delgrande and Wolfgang Faber, editors, *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 242–247. Springer, 2011.

[58] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM Transactions on Computational Logic*, 7(1):1–37, January 2006.

[59] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. *J. Artif. Intell. Res. (JAIR)*, 35:813–857, 2009.

[60] JUnit.org community. JUnit, Resources for Test Driven Development. `http://www.junit.org/`.

[61] Christian Kloimüllner, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A system for visualising and visual editing of interpretations for answer-set programs. *CoRR*, abs/1109.4095, 2011.

[62] Nicola Leone, Georg Gottlob, Riccardo Rosati, Thomas Eiter, Wolfgang Faber, Michael Fink, Gianluigi Greco, Giovambattista Ianni, Edyta Kałka, Domenico Lembo, Maurizio Lenzerini, Vincenzino Lio, Bartosz Nowicki, Marco Ruzzi, Witold Staniszkis, and Giorgio Terracina. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, June 2005. ACM Press.

[63] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.

[64] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.

[65] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[66] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[67] Vladimir Lifschitz and Hudson Turner. Splitting a Logic Program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[68] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002. AAAI Press / MIT Press.

[69] V. Wiktor Marek and V.S. Subrahmanian. The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In *Proceedings of the 6th International Conference on Logic Programming – ICLP'89*, pages 600–617. MIT Press, 1989.

[70] Ilkka Niemelä and Patrik Simons. Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.

[71] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog Decision Support System for the Space Shuttle. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001)*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2001.

[72] Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and Hans Tompits. On the small-scope hypothesis for testing answer-set programs. In *KR*, 2012.

[73] Johannes Oetsch, Jörg Pührer, Martina Seidl, Hans Tompits, and Patrick Zwickl. Videas: A development tool for answer-set programs based on model-driven engineering technology. In *LPNMR*, pages 382–387, 2011.

[74] Johannes Oetsch, Jörg Pührer, and Hans Tompits. The sealion has landed: An ide for answer-set programming—preliminary report. In *INAP2011/WLP2011*, volume abs/1109.3989, 2011.

[75] Simona Perri, Francesco Ricca, Giorgio Terracina, D. Cianni, and P. Veltri. An integrated graphic tool for developing and testing DLV programs. In Marina De Vos and Torsten Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 86–100, 2007.

[76] S. Polyviou and G. Samaras P. Evripidou. Query by Browsing: A Visual Query Language Based on the Relational Model and the Desktop User Interface Paradigm. University of Cyprus, Department of Computing, 2004.

[77] H. A. Proper. Interactive Query Formulation using Query by Navigation. *Asymetrix Research Report 94-4, Asymetrix Research Laboratory*, 1994.

[78] Teodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.

[79] Francesco Ricca. The DLV Java Wrapper. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 305–316, Messina, Italy, September 2003. Online at `http://CEUR-WS.org/Vol-78/`.

[80] Francesco Ricca, Lorenzo Gallucci, Roman Schindlauer, Tina Dell'Armi, Giovanni Grasso, and Nicola Leone. OntoDLV: an ASP-based system for enterprise ontologies. *Journal of Logic and Computation*, 2009.

[81] G. Santucci and P. A. Sottile. Query by Diagram: a Visual Environment for Querying Databases. Dipartimento di Informatica e Sistemistica, Universitá degli Studi di Roma ′La Sapienza′, 1993.

[82] Patrik Simons. Smodels Homepage, since 1996. `http://www.tcs.hut.fi/Software/smodels/`.

[83] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[84] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2004.

[85] Adrian Sureshkumar, Marina De Vos, Martin Brain, and John Fitch. APE: An AnsProlog* Environment. In Marina De Vos and Torsten Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 101–115, 2007.

[86] Tommi Syrjänen. Lparse 1.0 User's Manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[87] Giorgio Terracina, Erika De Francesco, Claudio Panetta, and Nicola Leone. Enhancing a DLP system for advanced database applications. In *Proceedings of the International Conference on Web Reasoning and Rule Systems (RR 2008)*, Karlsruhe, Germany, 2008. Springer Verlag.

[88] Giorgio Terracina, Nicola Leone, Vincenzino Lio, and Claudio Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming*, 8:129–165, 2008.

[89] Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[90] Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating answer-set programs in lana. *TPLP*, 12(4-5):619–637, 2012.

[91] Jan Wielemaker. Prolog Unit Tests. `http://www.swi-prolog.org/pldoc/package/plunit.html`.

[92] Johan Wittocx. IDPDraw, a tool used for visualizing answer sets, since 2009. `http://dtai.cs.kuleuven.be/krr/software/visualisation`.

[93] D. Young and B. Shneiderman. A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation. *Human-Computer Interaction Laboratory & Department of Computer Science*, 1993.

[94] Yuting Zhao. ASSAT homepage, since 2002. `http://assat.cs.ust.hk/`.