Testing ASP programs in *ASPIDE*

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

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

Abstract. Answer Set Programming (ASP) is a declarative logic programming formalism, which nowadays counts several advanced real-world applications, and has stimulated some interest also in industry. Although some environments for ASP-program development have been proposed in the last few years, the crucial task of *testing* ASP programs received less attention, and is an Achilles' heel of the available programming environments.
In this paper we present a new language for specifying and running *unit tests* on ASP programs. The testing language has been implemented in *ASPIDE*, a comprehensive IDE for ASP, which supports the entire life-cycle of ASP development with a collection of user-friendly graphical tools for program composition, *testing*, debugging, profiling, solver execution configuration, and output-handling.

1 Introduction

Answer Set Programming (ASP) [1] is an expressive [2] logic programming paradigm proposed in the area of non-monotonic reasoning. ASP allows one to declaratively specify a complex computational problem by a logic program whose answer sets correspond to solutions and then use a solver to find such a solution [3]. The high expressive power of ASP has been profitably exploited for developing advanced applications belonging to several fields, from Artificial Intelligence [4–10] to Information Integration [11], and Knowledge Management [12–14]. Interestingly, these applications of ASP recently have stimulated some interest also in industry [15].

On the one hand, the effective application of ASP in real-world scenarios was made possible by the availability of efficient ASP systems [4, 16–26]. On the other hand, the adoption of ASP can be further boosted by offering effective programming tools capable of supporting the programmers in managing large and complex projects [27].

In the last few years, a number of tools for developing ASP programs have been proposed, including editors and debuggers [28–38]. Among them, *ASPIDE* [38] –which stands for Answer Set Programming Integrated Development Environment– is one of the most complete development tools[1] and it integrates a cutting-edge editing tool (featuring dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, etc.) with a collection of user-friendly graphical

[1] For an exaustive feature-wise comparison with existing environments for developing logic programs we refer the reader to [38].

tools for program composition, debugging, profiling, DBMS access, solver execution configuration and output-handling.

Although so many tools for developing ASP programs have been proposed up to now, the crucial task of *testing* ASP programs received less attention [39], and is an Achilles' 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 [38]), and also the one present in the first versions of *ASPIDE* is far from being effective.

In this paper we present a pragmatic solution for testing ASP programs. In particular, we present a new language for specifying and running *unit tests* on ASP programs. The testing language presented in this paper is inspired to the JUnit framework [40]: the developer can specify the rules composing one or several units, specify one or more inputs and assert a number of conditions on the expected outputs. 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 chosen ASP solver. Note that test case specification is applicable independently of the used ASP solver. The testing language was implemented in *ASPIDE*, which also provides the user with some graphic tools that make the development of test cases simpler. The testing tool described in this work extends significantly the one formerly available in *ASPIDE*, and enriches its collection of user-friendly graphical tools for program composition, debugging, profiling, database management, solver execution configuration, and output-handling.

As far as related work is concerned, the task of testing ASP programs was approached for the first time, to the best of our knowledge, in [39] where the notion of structural testing for ground normal ASP programs is defined and a method for automatically generating tests is introduced. The results presented in [39] are, somehow, orthogonal to the contribution of this paper. Indeed, no language/implementation is proposed in [39] for specifying/automatically-running the produced test cases; whereas, the language presented in this paper can be used for encoding the output of a test case generator based on the methods proposed in [39]. Finally, it is worth noting that, testing approaches developed for other logic languages, like prolog [41–43], cannot be straightforwardly ported to ASP because of the differences between the languages.

The remainder of this paper is organized as follows: in Section 2 we overview *ASPIDE*; in section 3 we introduce a language for specifying unit tests for ASP programs; in Section 4 we describe the user interface components of *ASPIDE* conceived for creating and running tests; finally, in Section 5 we draw the conclusion.

2 *ASPIDE*: Integrated Development Environment for ASP

ASPIDE is an Integrated Development Environment (IDE) for ASP, which features a rich *editing tool* with a collection of user-friendly *graphical tools* for ASP program development. In this section we first summarize the main features of the system and then we overview the main components of the *ASPIDE* user interface. For a more detailed description of *ASPIDE*, as well as for a complete comparison with competing tools, we refer the reader to [38] and to the online manual published in the system web site `http://www.mat.unical.it/ricca/aspide`.

System Features. *ASPIDE* is inspired to Eclipse, one of the most diffused programming environments. The main features of *ASPIDE* are the following:

- *Workspace management*. The system allows one to organize ASP programs in projects, which are collected in a special directory (called workspace).
- *Advanced text editor*. The editing of ASP files is simplified by an advanced text editor. Currently, the system is able to load and store ASP programs in the syntax of the ASP system DLV [16], and supports the `ASPCore` language profile employed in the ASP System Competition 2011 [44]. *ASPIDE* can also manage *TYP files* specifying a mapping between program predicates and database tables in the DLV^{DB} syntax [45]. 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, like: *Automatic completion*, *Dynamic code templates*, *Quick fix*, and *Refactoring*. Indeed, the system is able to complete (on request) predicate names, as well as variable names. Predicate names are both 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. When several possible alternatives for completion are available the system shows a pop-up dialog. Moreover, the writing of repeated programming patterns (like transitive closure or disjunctive rules for guessing the search space) is assisted by advanced auto-completion with code templates, which can generate several rules at once according to a known pattern. Note that code templates can be also user defined by writing DLT [46] files. The refactoring tool allows one to modify in a guided way, among others, predicate names and variables (e.g., variable renaming in a rule is done by considering bindings of variables, so that variables/predicates/strings occurring in other expressions remain unchanged). Reported errors or warnings can be automatically fixed by selecting (on request) one of the system's suggested quick fixes, which automatically change the affected part of code.
- *Outline navigation*. *ASPIDE* creates an outline view which graphically represents program elements. Each item in the outline can be used to quickly access 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 visual editor (see below).
- *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. Note that the checker considers the entire project, and warns the user by 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 in multiple files are run together.
- *Dependency graph*. The system is able to display several variants of the dependency graph associated to a program (e.g., depending on whether both positive and negative dependencies are considered).
- *Debugger and Profiler*. Semantic errors detection as well as code optimization can be done by exploiting graphic tools. In particular, we developed a graphical user interface for embedding in *ASPIDE* the debugging tool *spock* [30] (we have also adapted spock for dealing with the syntax of the DLV system). Regarding the profiler, we have fully embedded the graphical interface presented in [47].

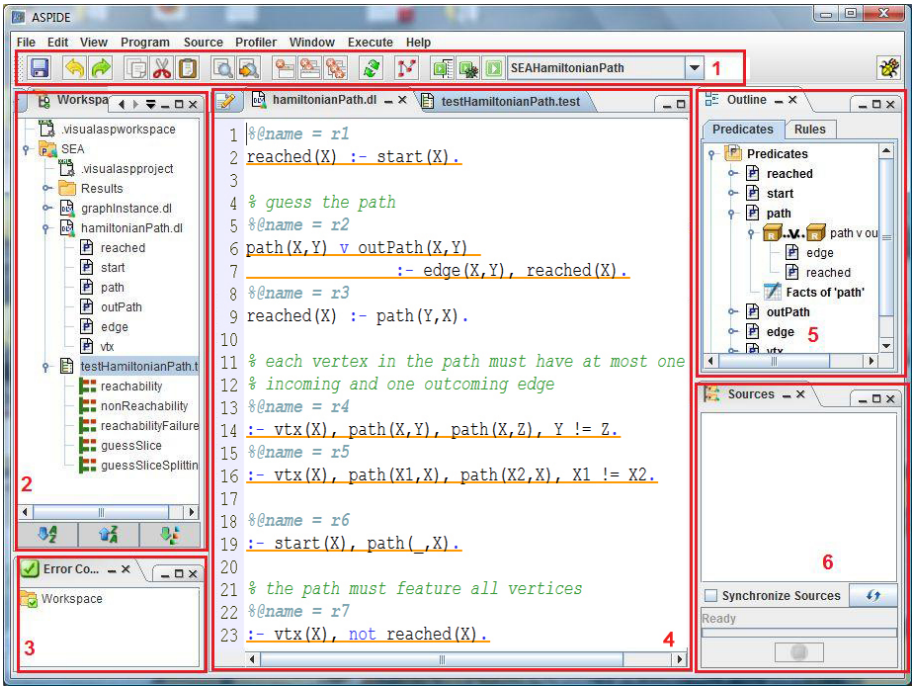

Fig. 1. The *ASPIDE* graphical user interface.

- *Unit Testing*. The user can define unit tests and verify the behavior of programs units. The language for specifying unit tests, as well as the graphical tools of *ASPIDE* assisting the development of tests, are described in detail in the following sections.
- *Configuration of the execution*. This feature allows one to configure and manage input programs and execution options (called *run configurations*).
- *Presentation of results*. The outputs of the program (either answer sets, or query results) are visualized in a tabular representation or in a text-based console. The result of the execution can be also saved in text files for subsequent analysis.
- *Visual Editor*. The users can *draw* logic programs by exploiting a full graphical environment that offers a QBE-like tool for building logic rules [48]. The user can switch, every time he needs, from the text editor to the visual one (and vice-versa) thanks to a reverse-rengineering mechanism from text to graphical format.
- *Interaction with databases.* Interaction with external databases is useful in several applications (e.g., [11, 15, 8]). *ASPIDE* provides a fully graphical import/export tool that automatically generates mappings by following the DLV^{DB} Typ files specifications [45]. Text editing of Typ mappings is also assisted by syntax coloring and auto-completion. Database oriented applications can be run by setting DLV^{DB} as solver in a run configuration.

Interface Overview The system interface of *ASPIDE* is depicted in Figure 1. The most common operations can be quickly executed through a toolbar present in the upper part

of the *ASPIDE* interface (zone 1). From left to right there are buttons allowing one to: save files, undo/redo, copy & paste, find & replace, switch between visual to text editor, run the solver/profiler/debugger. The main editing area (zone 4) is organized in a multi-tabbed panel possibly collecting several open files. On the left there is the explorer panel (zone 2) which allows one to browse the workspace; and 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 sources panel (zone 6). The first shows an outline of the currently edited file, while the latter reports a list of the database sources connected with the current project. Note that the layout of the system can be customized by the user, indeed panels can be moved and rearranged as the user likes.

ASPIDE is written in Java and runs on the most diffused operating systems (Microsoft Windows, Linux, and Mac OS) and can connect to any database supporting Java DataBase Connectivity (JDBC).

3 A language for testing ASP programs

Software testing [49] 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 mean for establishing whether the program is totally correct; conversely testing is a pragmatic and cheap way of finding errors by executing some test. A test case is the specification of some input I and corresponding expected outputs O. A test case fails when the outputs produced by running the program do not correspond to O, it passes otherwise.

One of the most diffused white-box[2] testing techniques is *unit testing*. The idea of unit testing is to asses an entire software by testing its subparts called *units* (and corresponding to small testable parts of a program). In a software implemented by using imperative object-oriented languages, unit testing corresponds to assessing separately portions of the code like class methods. The same idea can be applied to ASP, once the notion of unit is given. We intend as unit of an ASP program P any subset of the rules of P corresponding to a splitting set [50] (actually the system exploits a generalization of the splitting theorem by Lifschitz and Turner [50] to the non-ground case [51]). In this way, the behavior of units can be verified (by avoiding unwanted behavioral changes due to cycles) both when they run isolated from the original program as well as when they are left immersed in (part of) the original program.

In the following, we present a pragmatic solution for testing ASP programs, which is a new language, inspired to the JUnit framework [40], for specifying and running *unit tests*. The developer, given an ASP program, can select the rules composing an 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 automatically

[2] A test conceived for verifying some functionality of an application without knowing the code internals is said to be a black-box test. A test conceived for verifying the behavior of a specific part of a program is called white-box test. White box testing is an activity usually carried out by developers and is a key component of agile software development [49].

verified by calling an ASP solver and checking its output. In particular, we allow three test execution modes:

- *Execution of selected rules*. The selected rules will be executed separated from the original program on the specified inputs.
- *Execution of split program*. The program corresponding to the splitting set containing the atoms of the selected rules is run and tested. In this way, 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).
- *Execution in the whole program*. The original program is run and specific assertions regarding predicates contained in the unit are checked. This corresponds to filter test results on the atoms contained in the selected rules.

Testing Language. A test file can be written according to the following grammar:[3]

```
1  : invocation("invocationName" [ ,"solverPath", "options" ]?);
2  : [ [ input("program"); ] | [ inputFile("file"); ] ]*
3  : [
4  : testCaseName([ SELECTED_RULES | SPLIT_PROGRAM | PROGRAM ]?)
5  : {
6  : [newOptions("options");]?
7  : [ [ input("program"); ] | [ inputFile("file"); ] ]*
8  : [ [ excludeInput("program"); ]
9  : | [ excludeInputFile("file"); ] ]*
10 : [
11 : [ filter | pfilter | nfilter ]
12 : [ [ (predicateName [ ,predicateName ]* ) ]
13 : | [SELECTED_RULES] ] ;
14 : ]?
15 : [ selectRule(ruleName); ]*
16 : [ assertName( [ intnumber, ]? "program" ); ]*
17 : }
18 : ]*
19 : [ assertName( [ intnumber, ]? "program" ); ]*
```

A test file might contain a single test or a test suite (a set of tests) including several test cases. Each test case includes one or more assertions on the execution results.

The *invocation* statement (line 1) sets the global invocation settings, that are applied to all tests specified in the same file (name, solver, and execution options). In the implementation, the invocation name might correspond to an *ASPIDE* run configuration, and the solver path and options are not mandatory.

The user can specify one or more global inputs by writing some *input* and *inputFile* statements (line 2). The first kind of statement allows for writing the input of the test in the form of ASP rules or simply facts; the second statement indicates a file that contains some input in ASP format.

[3] Non-terminals are in bold face; token specifications are omitted for simplicity.

A test case declaration (line 4) is composed by a name and an optional parameter that allows one to choose if the execution will be done on the entire program, on a subset of rules, or considering program corresponding to the splitting set containing the selected rules.

The user can specify specific solver options (line 6), as well as specific inputs (line 7) which are valid in a given test case. Moreover, global inputs of the test suite can be excluded by exploiting *excludeInput* and *excludeInputFile* statements (lines 8 and 9).

The optional statements *filter*, *pfilter* and *nfilter* (lines 11, 12 and 13) are used to filter out output predicates from the test results predicates, specified as parameter, on the execution result when assertions will be executed.[4]

The statement *selectRule* (line 15) allows one for selecting rules among the ones composing the global input 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 (see Figure 1). *ASPIDE* adds automatically the comments specifying rule names. If a set of selected rules does not belong to the same splitting set, the system has to print a warning indicating the problem.

The expected output of a test case is expressed in term of assertions statements (lines 16/19). The possible assertions 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;

together with the corresponding negative assertions: *assertFalse*, *assertCautiouslyFalse*, *assertBravelyFalse*, *assertFalseIn*, *assertFalseInAtLeast*, *assertFalseInAtMost*. Assertions can be global (line 19) or local to a single test (line 16).

In the following we report an example of test case file.

Test case example. The Hamiltonian Path problem is a classical *NP-complete* problem in graph theory. *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?* Suppose that the graph G is specified by using facts over predicates vtx (unary) and $edge$ (binary), and the starting node a is specified by the predicate $start$ (unary). The program in Figure 1 solves the problem.

The disjunctive rule (r_2) guesses a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary

[4] *pfilter* selects only positive literals and excludes the strong negated ones, while *nfilter* has opposite behavior.

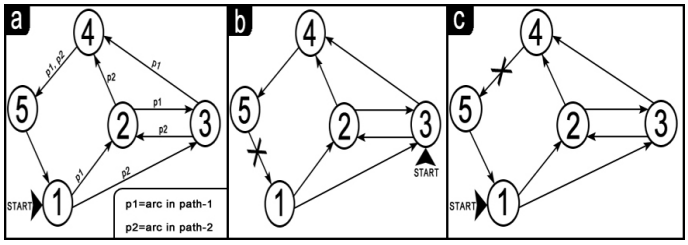

Fig. 2. Input graphs.

predicate *reached* is defined, which specifies the set of nodes which are reached from the starting node. In the checking part, the two constraints r_4 and r_5 ensure that the set of arcs S selected by *path* meets the following requirements, which any Hamiltonian Path must satisfy: (i) a vertex must have at most one incoming edge, and (ii) a vertex must have at most one outgoing edge. The constraints r_6 and r_7 enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S and the start node must be the first node of the path.

In order to test this encoding we define a test suite file. Suppose that the encoding is stored in a file named *hamiltonianPath.dl*. Suppose also that the graph instance of Figure 2a is stored in a file named *graphInstance.dl*, and is composed by the following facts: *vtx(1). vtx(2). vtx(3). vtx(4). vtx(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).*.

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

```
invocation("SEAHamiltonianPath", "/usr/bin/dlv", "");
inputFile("hamiltonianPath.dl");
reachability()
{
inputFile("graphInstance.dl");
input("start(1).");
assertTrue("reached(1).reached(2).reached(3).reached(4).reached(5).");
}
guessSlice(SELECTED_RULES)
{
inputFile("graphInstance.dl");
input("reached(1).");
selectRule("r2");
selectRule("r3");
assertBravelyFalse("path(1, 2).");
assertBravelyTrue("path(1, 2).");
}
guessSliceNonReachability(SPLIT_PROGRAM)
{
inputFile("graphInstance.dl");
input("start(3).");
```

```
excludeInput( "edge(5, 1)." );
selectRule("r2");
selectRule("r3");
assertCautiouslyFalse( "reached(1)." );
}
assertFalse("path(1, 2).");
```

Here, we first setup the invocation parameters by indicating DLV as solver, then we specify the file to be tested *hamiltonianPath.dl* by exploiting a global input statement; then, we add the test case *reachability*, in which we verify that if node 1 is the starting node than nodes {1,2,3,4,5} are reached (see Figure 2a). To this end we specify *graphInstance.dl* as local input file and *start(1)* as local input and write some assertion requiring that atoms {*reached(1), reached(2), reached(3), reached(4), reached(5)*} are (cautiously) true.

In the second test case, named *guessSlice*, we select rules r_2 and r_3 and we require to test selected rules in isolation. The (local) inputs in this case are: the file *graphInstance.dl* and the fact {*reached(1).*}. In this case we are testing only the part of the program that guesses the paths, and we specify a couple of additional assertions (i.e., *path(1, 2)* has to be true in some answer set and false in some other).

Test case *guessSliceNonReachability* is run in *SPLIT_PROGRAM* modality, which requires to test the subprogram containing all the rules belonging to the splitting set corresponding to the selection (i.e., {*path, outPath, edge, reached*}). With this test case the sub-program that we are testing is composed by all the rules of the Hamiltonian Path example without the constraints. Also in this case we include the graph instance file *graphInstance.dl*. Additionally, we remove edge from 5 to 1 (see Fig. 2b), so that node 1 is not reached, then we add the assertion corresponding to this observation. Finally, we add a global assertion (*assertFalse("path(1,2).")* to check if *path(1,2)* is not contained in any answer set (note that the graph instance and the starting node are missing in the global inputs).

The test file described above can be created graphically and executed in *ASPIDE* as described in the following section.

4 Unit Testing in *ASPIDE*

In this section we describe the graphic tools implemented in *ASPIDE* conceived for developing and running test cases. Space constraints prevent us from providing a complete description of all the usage scenarios and available commands. However, in order to have an idea about the capabilities of the testing interface of *ASPIDE*, we describe step by step how to implement the example illustrated in the previous section.

Suppose that we have created in *ASPIDE* a project named SEA, which contains the files *hamiltonianPath.dl* and *graphInstance.dl* (see Fig. 1) storing the encoding of the hamiltonian path problem and the graph instance, respectively. Since the file that we want to test in our example is *hamiltonianPath.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. 3a). The system shows the test creation dialog (Fig. 3b), which allows for both

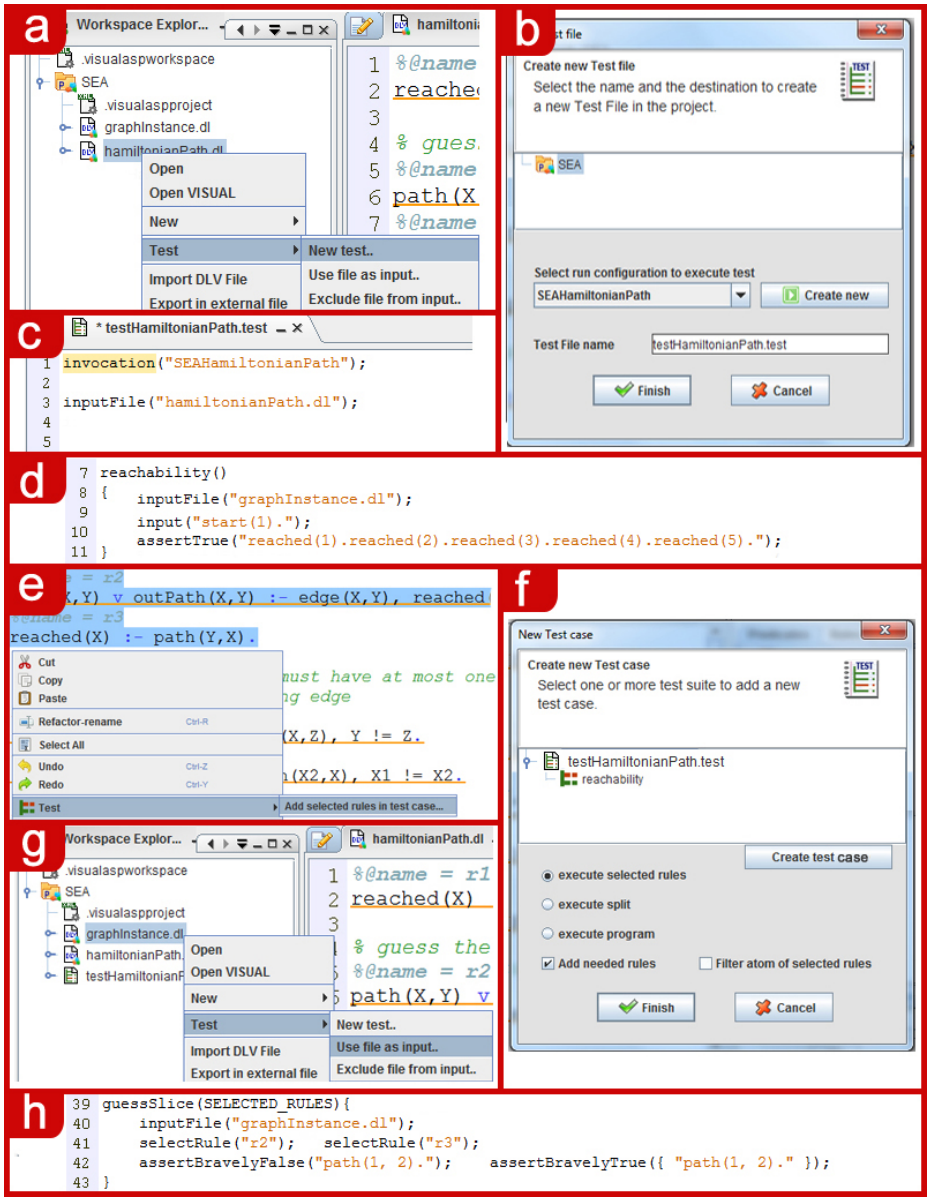

Fig. 3. Test case creation.

setting the name of the test file and selecting a previously-defined run configuration (storing execution options). In this case we select the one named *SEAHamiltonianPath* explicitly referring to the DLV system (*ASPIDE* will get the solver path directly from the selected run configuration). By clicking on the *Finish* button, the new test file is

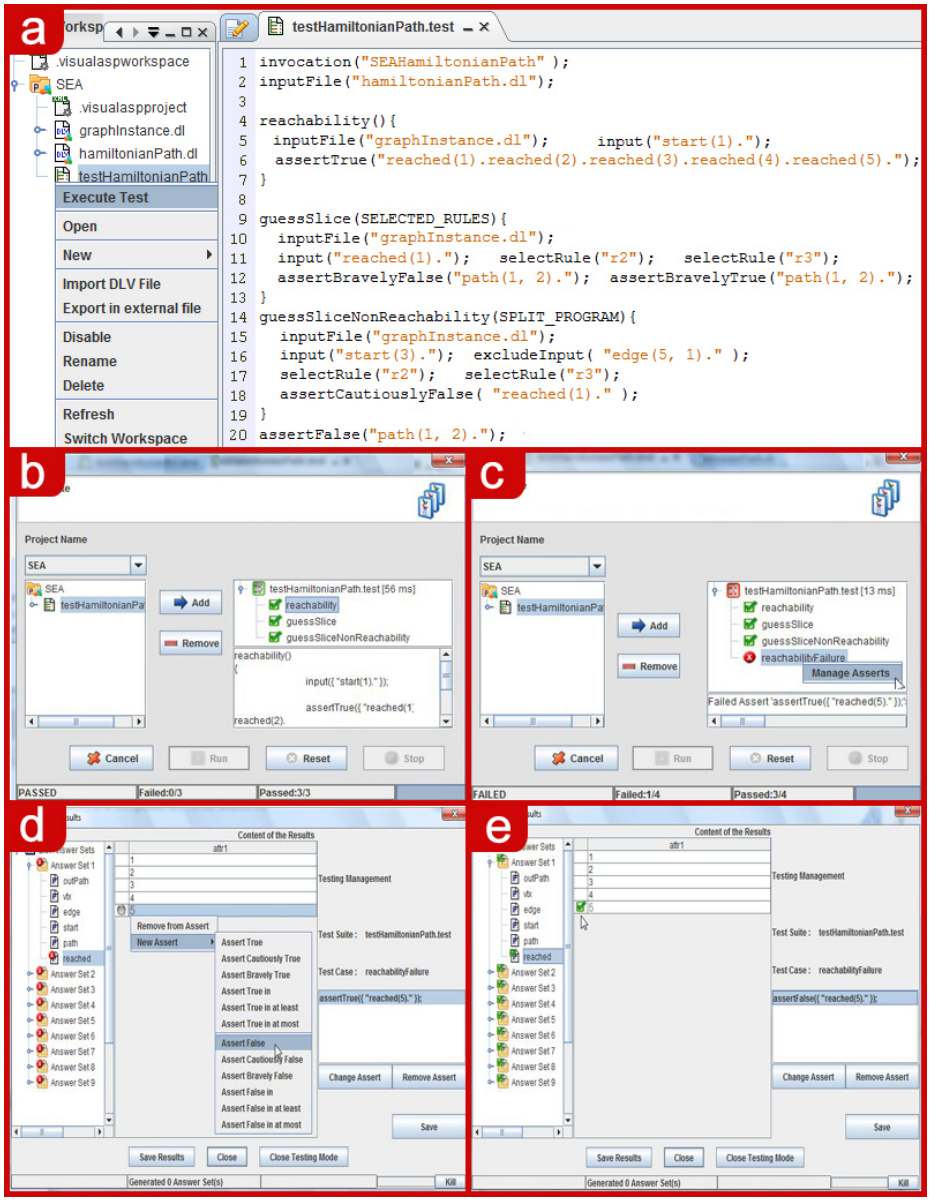

Fig. 4. Test case execution and assertion management.

created (see Fig. 3c) where the statements regarding input files and invocation are added automatically. We add the first unit test (called *reachability*) by exploiting the text editor (see Fig. 3d), whereas we build the remaining ones (working on some selected rules) by exploiting the logic program editor. After opening the *hamiltonianPath.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. 3e). The system opens a dialog window where we indicate the test file in which we want to add the new test case (fig. 3f). We click on the *Create test case*; the system will ask for the name of the new test case and we write *guessSlice*; after that, on the window, we select the option *execute selected rules* and click on the *Finish* button. The system will add the test case *guessSlice* filled with the *selectRule* statements indicating the two 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. 3g). The test created up to now is shown in figure 3h. Following an analogous procedure we create the remaining test cases (see Fig. 4a). 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 (see Fig. 4b). Failing tests are indicated by a red icon, while green icons indicate passing tests. At this point, in order to show how to modify graphically a test case, we add the following additional test that purposely fails:

```
reachabilityFailure()
{
inputFile("graphInstance.dl");
input("start(1).");
excludeInput("edge(4, 5).");
excludeInput(":- vtx(X), not reached(X).");
assertTrue("reached(5).");
}
```

As shown in Figure 2c this additional test (as expected) fails, and the reason for this failure is indicated (see Fig. 4c) 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 matching correctly 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 *assertTrue("reached(5).")*, so we expect that all solutions contain the literal *reached(5)*; however, in our instance, node 5 is never reached, this is because the test case purposely contains an error. We modify this test case by adding the right assertion. This can be obtained by acting directly on the result window (fig. 4d). We remove the old assertion by selecting it and clicking on the *Remove Assert* button. Finally, we save the modifications, and we execute the test suite again (see Fig. 4e).

5 Conclusion

This paper presents a pragmatic environment for testing ASP programs. In particular, we present a new language, inspired to the JUnit framework [40], for specifying and running *unit tests* on ASP programs. The testing language has been implemented in *ASPIDE* together with some graphic tools for easing both the development of tests and the analysis of test execution.

As far as future work is concerned, we plan to extend *ASPIDE* by improving/introducing additional dynamic editing instruments, and graphic tools. In particular, we plan

to further improve the testing tool by supporting (semi)automatic test case generation based on the structural testing techniques proposed in [39].

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

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
3. Lifschitz, V.: Answer Set Planning. In: ICLP'99) 23–37
4. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR'07. LNCS 4483, (2007) 3–17
5. Balduccini, M., Gelfond, M., Watson, R., Nogeira, M.: The USA-Advisor: A Case Study in Answer Set Planning. In: LPNMR 2001 (LPNMR-01). LNCS 2173, (2001) 439–442
6. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Logic-Based Artificial Intelligence. Kluwer (2000) 257–279
7. Baral, C., Uyan, C.: Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In: LPNMR 2001 (LPNMR-01). LNCS 2173, (2001) 186–199
8. Friedrich, G., Ivanchenko, V.: Diagnosis from first principles for workflow executions. Tech. Rep., http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf.
9. Franconi, E., Palma, A.L., Leone, N., Perri, S., Scarcello, F.: Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In: LPAR 2001. LNCS 2250, (2001) 561–578
10. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog Decision Support System for the Space Shuttle. In: Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001). LNCS 1990, (2001) 169–183
11. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005, Baltimore, Maryland, USA, ACM Press (2005) 915–917
12. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
13. Bardadym, V.A.: Computer-Aided School and University Timetabling: The New Wave. In: Practice and Theory of Automated Timetabling, First International Conference 1995. LNCS 1153, (1996) 22–45
14. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV Applications for Knowledge Management. In: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009). LNCS 5753, (2009) 591–597
15. Grasso, G., Leone, N., Manna, M., Ricca, F.: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of M. Gelfond. LNCS 6565 (2010)
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
17. Simons, P.: Smodels Homepage (since 1996) `http://www.tcs.hut.fi/Software/smodels/`.

18. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI **138** (2002) 181–234
19. Zhao, Y.: ASSAT homepage (since 2002) http://assat.cs.ust.hk/.
20. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI-2002, Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002)
21. Babovich, Y., Maratea, M.: Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. http://www.cs.utexas.edu/users/tag/cmodels.html (2003)
22. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007,(2007) 386–392
23. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM TOCL **7**(1) (2006) 1–37
24. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451
25. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), Sydney, Australia, AAAI Press (2008) 422–432
26. Denecher, M., Vennekens, J., Bond, S., Gebser, M., M., M.T.: The second answer set programming system competition. In: Logic Programming and Nonmonotonic Reasoning — 10th International Conference, LPNMR'09. LNCS 5753, Potsdam, Germany, Berlin // Heidelberg (2009) 637–654
27. Dovier, A., Erdem, E.: Report on application session @lpnmr09 (2009) http://www.cs.nmsu.edu/ALP/2010/03/report-on-application-session-lpnmr09/.
28. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 86–100
29. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog* Environment. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 101–115
30. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: That is Illogical Captain! The Debugging Support Tool spock for Answer-Set Programs: System Description. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 71–85
31. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
32. El-Khatib, O., Pontelli, E., Son, T.C.: Justification and debugging of answer set programs in ASP. In: Proceedings of the Sixth International Workshop on Automated Debugging, California, USA, ACM (2005)
33. Oetsch, J., Pührer, J., Tompits, H.: Catching the ouroboros: On debugging non-ground answer-set programs. In: Proc. of the ICLP'10. (2010)
34. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging asp programs by means of asp. In: LPNMR'07. LNCS 4483, (2007) 31–43
35. De Vos, M., Schaub, T., eds.: SEA'07: Software Engineering for Answer Set Programming. In: . Volume 281., CEUR (2007) Online at http://CEUR-WS.org/Vol-281/.
36. De Vos, M., Schaub, T., eds.: SEA'09: Software Engineering for Answer Set Programming. In: . Volume 546., CEUR (2009) Online at http://CEUR-WS.org/Vol-546/.
37. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based system for enterprise ontologies. Journal of Logic and Computation (2009)

38. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated Development Environment for Answer Set Programming. In: Logic Programming and Nonmonotonic Reasoning — 11th International Conference, LPNMR'11, Vancouver, Canada, 2011, Proceedings. LNCS 6645, (May 2011) 317–330
39. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence, Amsterdam, The Netherlands, The Netherlands, IOS Press (2010) 951–956
40. JUnit.org community: JUnit, Resources for Test Driven Development `http://www.junit.org/`.
41. Jack, O.: Software Testing for Conventional and Logic Programming. Walter de Gruyter & Co., Hawthorne, NJ, USA (1996)
42. Wielemaker, J.: Prolog Unit Tests `http://www.swi-prolog.org/pldoc/package/plunit.html`.
43. Cancinos, C.: Prolog Development Tools - ProDT `http://prodevtools.sourceforge.net`.
44. Calimeri, F., Ianni, G., Ricca, F.: The third answer set programming system competition (since 2011) `https://www.mat.unical.it/aspcomp2011/`.
45. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **8** (2008) 129–165
46. Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C.: Answer Set Programming with Templates. In: ASP'03, Messina, Italy (2003) 239–252 Online at `http://CEUR-WS.org/Vol-78/`.
47. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A Visual Tracer for DLV. In: Proc. of SEA'09, Potsdam, Germany (2009)
48. Febbraro, O., Reale, K., Ricca, F.: A Visual Interface for Drawing ASP Programs. In: Proc. of CILC2010, Rende(CS), Italy (2010)
49. Sommerville, I.: Software Engineering. Addison-Wesley (2004)
50. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: ICLP'94, MIT Press (1994) 23–37
51. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artif. Intell. **172** (2008) 1495–1539