TPLP: Page 1–31.

doi:10.1017/S1471068424000103

*Unit Testing in ASP Revisited: Language and Test-Driven Development Environment**

GIOVANNI AMENDOLA, GIUSEPPE MAZZOTTA and FRANCESCO RICCA
University of Calabria, Rende, Italy
(*e-mails:* giovanni.amendola@unical.it, giuseppe.mazzotta@unical.it, francesco.ricca@unical.it, ricca@mat.unical.it)

TOBIAS BEREI
University of Applied Sciences Upper Austria, Campus Hagenberg
(*e-mails:* tobias.berei@students.fh-hagenberg.at, bereitobias@hotmail.com)

submitted 24 May 2023; revised 21 December 2023; accepted 05 March 2024

Abstract

Unit testing frameworks are nowadays considered a best practice, included in almost all modern software development processes, to achieve rapid development of *correct* specifications. Knowledge representation and reasoning paradigms such as Answer Set Programming (ASP), that have been used in industry-level applications, are not an exception. Indeed, the first unit testing specification language for ASP was proposed in 2011 as a feature of the ASPIDE development environment. Later, a more portable unit testing language was included in the LANA annotation language. In this paper we revisit both languages and tools for unit testing in ASP. We propose a new unit test specification language that allows one to inline tests within ASP programs, and we identify the computational complexity of the tasks associated with checking the various program-correctness assertions. Test-case specifications are transparent to the traditional evaluation, but can be interpreted by a specific testing tool. Thus, we present a novel environment supporting test-driven development of ASP programs.

KEYWORDS: answer set programming, unit testing, development environments

1 Introduction

Answer Set Programming (ASP) (Brewka *et al.* 2011) is a well-known logic-based formalism developed in the area of knowledge representation and reasoning. ASP combines a purely declarative language based on the stable models semantics (Gelfond and Lifschitz 1991) with efficient implementations (Lierler *et al.* 2016). ASP is known to be suited for rapid prototyping of complex reasoning tasks and has been effectively used to solve a number of both academic and real-world applications of AI (Erdem *et al.* 2016). ASP

* This work was supported by Italian Ministry of Research (MUR) under PRIN project PINPOINT, CUP H23C22000280006, Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and PNRR projects FAIR "Future AI Research" – Spoke 9 – WP9.1 – CUP H23C22000860006.

allows encoding complex computational problems often in an easier and more compact way than mainstream (imperative) programming languages. For instance, the classical NP-complete problem of 3-Colorability is encoded in ASP using only two rules. Nonetheless, it is also easy to write incorrect ASP programs, which seem correct (often due to a misleading interpretation of rules based on intuition) but do not work as expected.

To speed up the development of correct and robust programs, many modern software development processes support some *test-driven development* (TDD) best practices (Fraser *et al.* 2003) such as *unit testing* (Beck 2002). In TDD the following sequence of actions is repeated while developing a new program (Beck 2002): (i) add a test that defines a function or improvements of a function, which should be very succinct, so that the developer focuses on the requirements before writing the code; (ii) run all tests and see if the new test fails. This confirms that the newly introduced tests indeed bring an improvement in detecting errors; (iii) write the code (maybe not perfect); (iv) run all tests, to become confident that the new code meets the test requirements, and does not introduce bugs; (v) refactor code to improve it while accommodating the new features.

Tests drive the development because the program is considered improved only if it passes new tests, while the repeated execution of tests allows to find problems early in the development cycle and to isolate the incorrect behavior more easily. This intuition has been subject to further empirical studies which proved that programmers who wrote more tests tended to be more productive (Erdogmus *et al.* 2005) and evidenced the superiority of the TDD practice over the traditional test-last approach or testing for correctness approach (Madeyski 2010). Two important best practices of TDD are the following: (i) concentrate on testing possibly small modules/functions/blocks of code; and (ii) adopt frameworks and tools to make the process of testing the program automatically. In the resulting software testing method, complex programs are split into *units*, which are tested in isolation providing (usually) small inputs and checking whether the expected outputs are computed. Tests for program units (i.e., *unit tests*) are directly derived from software requirements and often created before (or while) implementing the corresponding feature. The software is improved to pass the new tests, only.

TDD development practices are nowadays a standard technique used by expert programmers and development teams all over the world, no matter the programming language used. TDD development practices have been already applied also to many AI formalisms, as it is witnessed by the proposals in the literature, such as testing for Description Logics (Bezerra and Freitas 2017), and Constraint Programming (Lazaar *et al.* 2010). In particular, TDD development practices have been applied to ASP program development. Indeed the first unit testing language for ASP has been introduced in 2011 (Febbraro *et al.* 2011a) and was implemented in ASPIDE (Febbraro *et al.* 2011). Nonetheless, this solution presents some limitations from the perspectives of both the language for specifying unit tests and its implementation. Concerning the language, one drawback is the need for specifying unit tests in separate files concerning the program to test, which often is not very comfortable given that ASP programs might contain (comparatively) few lines of code. Having the possibility to write tests *together* with the source code, without interfering with the actual evaluation of the program itself, would provide clear advantages for the developer. Subsequently (De Vos *et al.* 2012), a more versatile approach of specifying unit tests together with ASP programs was included in the LANA annotation language. However, the prototypical implementation of LANA,

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

called ASPUnit, has never been included in a graphical development environment for ASP. A testing framework integrated into a development environment is comfortable for the developer, who can control the results in the same graphical environment she is using.

In this paper, we propose a new unit testing language for ASP, which comprises the key features of both the above-mentioned proposals, and we identify the computational complexity of the tasks associated with checking the various program-correctness conditions, which can be specified in unit tests. To the best of our knowledge, this is the first attempt to identify the complexity for all the tasks connected with unit testing in ASP. The new language is annotation-based as LANA, so to allow the development of test cases inline with ASP code and keep the assertion-based style for expressing test case conditions from the language of ASPIDE. Note that, being capable of defining and running tests together with the program has several clear advantages, such as promoting and exploiting programmer familiarity with the language (there is no need to know both ASP and another language such as python) and enables programmers to define declarative test case specifications. The proposed annotation style is more similar to the JUnit framework for Java (from which both ASPIDE and LANA proposals were inspired) and should look more familiar to developers that are accustomed to XUnit style languages (Beck 2002). Moreover, we present a novel web-based development environment for ASP supporting TDD of ASP programs that features an integrated implementation of unit testing.

We also observe that there is one additional advantage in equipping a programming language with unit tests: they can be used to check and certify the behaviors of programs with respect to *any* requirement of the application (Fraser *et al.* 2003). For example, ethical or trustful behavior can be seen as a software requirement (Sommerville 2007). Unit testing languages can be used to devise tests that check and certify that the behavior of the ASP program respects also application-specific ethical requirements. Programs that are provided without tests of such type might result in (unwanted) violation of ethical requirements (besides potential bugs). In a sense, our contribution can lead to the development of ASP-based AI applications that can be certified with respect also to ethical requirements and other specific expected behaviors by reporting the outcome of executing properly devised test suites.

Conference paper. This work is an extended and revised version of the paper "Testing in ASP: Revisited language and programming environment" accepted for publication in the 17th edition of the European Conference on Logics in Artificial Intelligence (Amendola *et al.* 2021). Preliminary results were illustrated during the 35th International Conference on Logic Programming ICLP 2019 (Bogaerts *et al.* 2019). The current version of the contribution extends the conference paper by:

1. Providing the complexity analysis of the main unit testing tasks.
2. Extending the description of the implementation.
3. Presenting the results of an experimental analysis that validates the applicability and efficiency of the proposed approach.

The investigation of computational complexity makes clear that testing an ASP program can be computationally very expensive (up to Π_2^P-c, cfr. Table 4). This result not only fills a gap in the understanding of unit testing in ASP, but also enabled the development of a (more) efficient implementation, as evidenced empirically by the

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

experimental analysis presented in this paper. Moreover, it serves as a valuable reminder to programmers that they should rely on the small-scope hypothesis (Oetsch *et al.* 2012) when crafting effective test cases. In other words, it is crucial to design small tests that cover the defects, while avoiding the use of large instances that may become impractical due to their high complexity. This understanding aids in promoting efficient and effective testing practices in ASP development.

Paper structure. This paper is organized as follows. Section 2 provides a preliminary introduction to ASP and introduces some basic notation; Section 3 introduces the new unit testing language by resorting to simple examples; Section 4 reports a study of the computational complexity of the main computational tasks related to unit testing; Section 5 describes the implementation of our programming environment; Section 6 reports an experimental evaluation aimed at demonstrate the effectiveness, but also the efficiency, of the proposed system in detecting bugs in common ASP problems. Section 7 discusses related work; Section 8 reports our conclusive statements.

2 Preliminaries on answer set programming

Let $\mathscr{P}$ be a set of predicates, C of constants, and V of variables. A *term* is a constant or a variable. An atom a of arity k is of the form $p(t_1, ..., t_k)$, where $p \in \mathscr{P}$ and $t_1, ..., t_n$ are terms. A *disjunctive rule* r is of the form

$$a_1 \vee \ldots \vee a_l \leftarrow b_1, \ldots, b_m,\ not\ c_1, \ldots,\ not\ c_n, \tag{1}$$

where all a_i, b_j, and c_k are atoms; $l, m, n \geq 0$ and $l + m + n > 0$; *not* represents *negation-as-failure*. The set $H(r) = \{a_1, ..., a_l\}$ is the *head* of r; $B^+(r) = \{b_1, ..., b_m\}$ and $B^-(r) = \{c_1, \ldots, c_n\}$ are the *positive body* and the *negative body* of r, respectively; and $B(r) = B^+(r) \cup B^-(r)$ is the *body* of r. A rule r is *safe* if each of its variables occurs in some positive body atom. A rule r is a *fact*, if $B(r) = \emptyset$ (we then omit $\leftarrow$ from the notation); a *constraint* if $H(r) = \emptyset$; *normal* if $|H(r)| \leq 1$; and *positive* if $B^-(r) = \emptyset$. A *(disjunctive logic) program* P is a finite set of disjunctive rules. P is called *normal* [resp. *positive*] if each $r \in P$ is normal [resp. positive]. Moreover, a program P is *head-cycle-free* (HCF) if there is a level mapping $\|.\|_h$ of P such that for every rule r of P: (*i*) For any l in $B^+(r)$, and for any l' in $H(r)$, $\|l\|_h \leq \|l'\|_h$; and (*ii*) For any pair l, l' of atoms in $H(r)$, $\|l\|_h \neq \|l'\|_h$. In the paper, $At(r)$ denotes the set of all atoms in rule r, and $At(P) = \bigcup_{r \in P} At(r)$ is the set of all atoms in P. We restrict attention to programs built on safe rules only (to avoid well-known issues (Leone *et al.* 2006)).

The *Herbrand universe* of P, denoted by U_P, is the set of all constants appearing in P. If there are no constants in P, we take $U_P = \{a\}$, where a is an arbitrary constant. The *Herbrand base* of P, denoted by B_P, is the set of all ground atoms that can be obtained from the predicate symbols appearing in P and the constants in U_P. Given a rule r of P, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a substitution mapping the variables occurring in r to constants in U_P. The *ground instantiation* of P, denoted by $ground(P)$, is the set of all the ground instances of the rules occurring in P.

Any set $I \subseteq B_P$ is an *interpretation*; it *satisfies* a rule $r \in ground(P)$ if $H_r \subseteq I$ (denoted as, $I \models H(r)$) or $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$ (denoted as, $I \models B(r)$); it

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

is a *model* of a program P (denoted $I \models P$) if for each rule $r \in ground(P)$, $I \models r$. A model M of P is *minimal* if no model $M' \subset M$ of P exists. We denote by $MM(P)$ the set of all minimal models of P. We write P^I for the well-known *Gelfond–Lifschitz reduct* (Gelfond and Lifschitz 1991) of P w.r.t. I, that is, the set of rules $H(r) \leftarrow B^+(r)$, obtained from rules $r \in ground(P)$ such that $B^-(r) \cap I = \emptyset$. We denote by $AS(P)$ the set of all *answer sets (or stable models)* of P, that is, the set of all interpretations I such that $I \in MM(P^I)$. We say that a program P is *coherent*, if $AS(P) \neq \emptyset$, otherwise, P is *incoherent*.

Finally, we recall a useful extension of the answer set semantics by the notion of *weak constraint* (Buccafurri *et al.* 2000). A weak constraint ω is of the form:

$$:\sim b_1, \ldots, \ b_m, \ not\, c_1, \ldots, \ not\, c_n. \ [c@l], \tag{2}$$

where c and l are nonnegative integers, representing a *cost* and a *level*, respectively. Let $\Pi = P \cup W$, where P is a set of rules and W is a set of weak constraints. We call M an answer set of Π if it is an answer set of P. We denote by $W(l)$ the set of all weak constraints at level l. For every answer set M of Π and any l, the *penalty* of M at level l, denoted by $Penalty_\Pi(M, l)$, is defined as $\sum_{\omega \in W(l),\ M \models B(\omega)} c$. For any two answer sets M and M' of Π, we say M is *dominated* by M' if there is l s.t. (i) $Penalty_\Pi(M', l) < Penalty_\Pi(M, l)$ and (ii) for all integers $k > l$, $Penalty_\Pi(M', k) = Penalty_\Pi(M, k)$. An answer set of Π is *optimal* if it is not dominated by another one of Π. We also mention aggregates, an extension of ASP that we do not recall here for keeping simple the description. We refer the reader to Calimeri *et al.* (2020) for more details.

Example 2.1
Consider the following set of facts $F = \{node(1); node(2); node(3); edge(1, 2); edge(1, 3); edge(2, 3)\}$, and the ASP program P:

$$col(X, red) \vee col(X, blue) \vee col(X, green) \leftarrow node(X);$$
$$\leftarrow edge(X, Y), \ col(X, C), \ col(Y, C)$$

The set of facts F models a cycle of length 3, while the two rules of the program P model the 3-colorability problem. It can be checked, that $F \cup \{col(1, red), \ col(2, blue), \ col(3, green)\}$ is an answer set of $P \cup F$.

To illustrate the usage of weak constraints, we now add the following

$$:\sim \ not\ col(X, red), \ preferablyRed(X).[1@1],$$

that prefers solutions having the nodes in *preferablyRed* to be colored in red (note that each violation of the weak constraint increases the solution penalty by 1). Suppose we add to facts in input $preferablyRed(2)$, we obtain that an optimal solution is $F \cup \{col(1, green), \ col(2, red), \ col(3, blue)\}$.

3 Unit testing of answer set programs

We now describe a new annotation-based test specification language that follows the Java annotation style of JUnit and can be fully embedded in programs compliant with the ASP-Core-2 input language format of ASP competitions (Gebser *et al.* 2017), which is nowadays a common syntactic fragment supported by the main ASP implementations. An

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Table 1. *Base constructs of the annotation language*

Annotation	Description
`@rule(name="`*rName*`",block="`*bName*`")`	The name *rName* is assigned to the following rule. Assigning a rule to a block is optional.
`@block(name="`*bName*`",rules={`*rList*`})`	Defines a block with name *bName*. Optionally a block may specify the list of rules that it covers.

`@test(`
` name = "`*testName*`",`
` scope = { `*referenceList*` },`
` programFiles = { `*programFileList*` },`
` input = "`*aspCode*`",`
` inputFiles = { `*inputFileList*` },`
` assert = { `*assertionList*` }`
`)`
	Defines a test case with name *testName* and scope *referenceList* which is a list of strings referencing the rules and/or blocks under test. The target file is the current file, if *programFiles* is not defined. An input for the test can be specified in *aspCode* or several files (property *inputFiles*) can be set optionally. Furthermore *assertionList* is a list of assertions (defined in Table 2) that have to be fulfilled for this test case.

annotation starts with "@" and is enclosed between `%**` and `**%` to distinguish multi-line comments, thus avoiding interference with program execution and to not require a separate test definition file (although in principle one could also collect testcases in separate files). The test specification language consists of base annotations and assertion condition annotations (or simply assertion annotations). The base annotations, described in Table 1, allow one to compose test cases, group subprograms in blocks, label rules and subprograms, and refer to the content of files containing programs. These annotations can be written anywhere in the ASP program, except `@rule(...)`, which has to be followed by an ASP rule in order to be assigned correctly. With regards to the `@test(...)` annotation the property *scope* includes a list of strings as a parameter that reference both rules and blocks under test (by their name). Furthermore the property *assert* holds a list of assertion annotations that are described in Table 2. Basically, the programmer is free to identify the (sub)programs to test, specify the input of a program in a test case, and assert a number of conditions on the expected output, that is, the basic operations supported by a XUnit testing language (Beck 2002). Note that, we have considered in our proposal all the assertions that are both present in the main unit testing languages proposed in the literature and that are more frequent in our experience.

An usage example of the test annotations language can be found in Figure 1, which contains an instance of the graph coloring problem (3-colorability). This instance produces six answer sets according to the color assignments of the colors to the specified nodes. In order to test whether the rules behave as expected, we have to be able to reference the rules under test. As we do not want to test facts, we assign the names *r1* and *r2* to the rules in Lines 8 and 11. Additionally we assign these rules to a block, which has been defined in Line 1. Afterward, we are able to reference the rules under test inside the `@test(...)` annotation starting in Line 13. First we specify the name of the test case and the rules under test, which is the block *ToTest* in this case. While referencing the block is more convenient, we could also reference the rules directly by writing

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Table 2. *Assertions for* `@test(...)` *annotation*

Assertion	Description
`@noAnswerSet`	The test must have no answer set.
`@trueInAll(atoms="`*atoms*`")`	The atoms specified in *atoms* must be true in all answer sets.
`@trueInAtLeast(number=`*n*`,atoms="`*atoms*`")`	The atoms specified in *atoms* must be true in at least n answer sets.
`@trueInAtMost(number=`*n*`,atoms="`*atoms*`")`	The atoms specified in *atoms* must be true in at most n answer sets.
`@trueInExactly(number=`*n*`,atoms="`*atoms*`")`	The atoms specified in *atoms* must be true in exactly n answer sets.
`@constraintForAll(constraint="c")`	The constraint specified in c must be fulfilled in all answer sets.
`@constraintInAtLeast(number=`*n*`,constraint="c")`	The constraint specified in c must be fulfilled in at least n answer sets.
`@constraintInAtMost(number=`*n*`,constraint="c")`	The constraint specified in c must be fulfilled in at most n answer sets.
`@constraintInExactly(number=`*n*`,constraint="c")`	The constraint specified in c must be fulfilled in exactly n answer sets.
`@bestModelCost(cost=`*cv*`,level=`*lv*`)`	The best model has to meet the cost of cv at level lv (for weak constraints).

`scope = { "r1", "r2" }`. Input rules can be defined with the property *input*, which are joined with the rules under test during test execution. They are equivalent to the facts of the program in this case, but can be different for more complex test specifications. With the property *assert* we can now define assertions that have to be fulfilled in order to execute the test with positive result. For this simple instance of the graph coloring problem, we can test whether the atom `col(1, red)` is true in exactly two answer sets while the atoms `col(1, red)` in combination with `col(2, blue)` should be true in exactly one answer set (Lines 17 and 18).

Note that the `@test(...)` annotation is very flexible and allows inputs to be selected freely by picking any subprogram, which plays the role of a *unit* to be tested (and run) in isolation. The *scope* attribute can be filled with any list of references (cfr. Table 1), including single rules, lists of rules (mentioned by name), and rules conveniently collected in a block (as in the example). Thus, it is possible to fine tune tests selecting any subprogram the programmer wants to test. The programmer can also flexibly control the input, inserting specific facts, subprograms, or reading the additional inputs from a file.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

```
1  %** @block(name="ToTest") **%
2
3  %** Test graph **%
4  node(1). node(2). node(3).
5  edge(1,2). edge(1,3). edge(2,3).
6
7  %** @rule(name="r1", block="ToTest") **%
8  col(X,red) | col(X,blue) | col(X,green) :- node(X).
9
10 %** @rule(name="r2", block="ToTest") **%
11 :- edge(X, Y), col(X,C), col(Y,C).
12
13 %**@test(name = "checkRules",
14        scope = { "ToTest" },
15        input = "node(1). node(2). node(3). edge(1,2). edge(1,3). edge(2,3).",
16        assert = {
17        @trueInExactly(number = 2, atoms = "col(1, red)."),
18        @trueInExactly(number = 1, atoms = "col(1, red). col(2, blue)") }
19    )
20 **%
```

Fig. 1. Testing graph coloring.

In spite of being a simple ASP program, Figure 1 shows how our lightweight annotation language can be used to define test cases without the need for a separate test definition file. The annotations do not interfere with program executability as being part of comments according to ASP-Core-2.

To further highlight the helpfulness of the TDD process for an ASP user, consider the example reported in Figure 2. The program is a (buggy) ASP encoding for the Hamiltonian Cycle (HC) problem, a classical problem in graph theory. Given a finite directed graph $G = (N, A)$, and a node $a \in N$, the HC problem asks whether a cycle in G exists starting from a and passing through each node in G. In our encoding, the first rule represents a guess for the set of arcs of the graph. The second and the third rule model the reachability in a graph. Indeed, the starting node X is reached (rule r2, line 7), and if X is reached and (X, Y) is in the cycle, then Y is also reached (rule r3, line 10). Finally, the last three rules are constraints to be satisfied so that the arcs chosen in the cycle form a Hamiltonian cycle. Indeed, the first two constraints state that for each node there is no more than one outgoing arc (rule r4, line 13) and there is no more than one incoming arc (rule r5, line 16); and the last constraint states that there is no node X which is not reached (rule r6, line 19). Now, if we have found a Hamiltonian cycle, we expect to see in each answer set an outgoing arc for each node appearing in the cycle. We can express this condition through a constraint, stating that it is not possible that we have a node X, and there is no arc from X to some other node in the cycle. This condition is modeled by the assertion `@constraintForAll` in line 24, by meaning that each solution (answer set) must satisfy that condition. If we consider the input $A = \{node(1), node(2), node(3), node(4), arc(1,2), arc(1,4), arc(2,4), arc(3,1), arc(4,3), start(1)\}$ reported in lines 23 and 24, we expect that a Hamiltonian cycle exists. Note that any programmer would run such kind of "live" test and maybe more than one, as in any programming language. However, the testing process fails. Indeed, our program on the given input admits the answer set: $A \cup \{inCycle(1,2), inCycle(2,4), inCycle(4,3), outCycle(1,4), outCycle(3,1)\}$, which does not satisfy the assertion. (Actually, the program would be a correct encoding for a

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

```
%** @block(name="hamCycle") **%

%** @rule(name="r1", block="hamCycle") **%
inCycle(X,Y) | outCycle(X,Y) :- arc(X,Y).

%** @rule(name="r2", block="hamCycle") **%
reached(X) :- start(X).

%** @rule(name="r3", block="hamCycle") **%
reached(Y) :- reached(X), inCycle(X,Y).

%** @rule(name="r4", block="hamCycle") **%
:- inCycle(X,Y), inCycle(X,Z), Y<>Z.

%** @rule(name="r5", block="hamCycle") **%
:- inCycle(X,Y), inCycle(Z,Y), X<>Z.

%** @rule(name="r6", block="hamCycle") **%
:- node(X), not reached(X).

%** @test(name = "checkProperty",
        scope = { "hamCycle" },
        input = "node(1). node(2). node(3). node(4). arc(1,2). arc(1,4).
            arc(2,4). arc(3,1). arc(4,3). start(1)."
        assert = { @constraintForAll(":-node(X), #count{Y:inCycle(X,Y)}=0.") }
    )
**%
```

Fig. 2. Bugged encoding of Hamiltonian Cycle.

Hamiltonian Path). Note that, if you are the author of a piece of code, you are less likely to see a mistake without "trying" the program (you are "expecting" your statements to be correct), and common practice is to run a small instance to see if the result is as we expect. Thus, without an automated testing procedure one should check manually all the answer sets or resort to a script. Automatic testing makes this phase of the development easier and declarative. Note that, since unit tests remain in the source code, once the program is updated, they are not lost (as it happens to manually handled result-checking sessions). Tests can be run again gaining all the advantages of *regression* testing (Beck 2002). If a test fails, bugs can be identified with a *debugger* (Busoniu *et al.* 2013).

More examples are reported in Berei (2019).

4 Computational complexity

In this section, we first overview the complexity classes that will be mentioned in the paper; then, we study the computational complexity of the main assertion-checking tasks that one can specify with the specification language described in the previous section.

4.1 Preliminaires on complexity classes

We recall some basic definitions that will be useful in the remainder of the paper. Hereafter, we assume the reader has basic knowledge of computational complexity (Papadimitriou 2007) and focus on the counting complexity classes. Roughly, the counting problems

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Table 3. *Complexity results of the main tasks concerning ASP programs evaluation*

ASP program	Answer set checking	Answer set existence	Answer set counting
Head-cycle-free	P	NP-c	#P-c
Disjunctive	coNP-c	Σ_2^P-c	# · coNP-c

address the issue of computing the number of solutions of an instance of a given problem. For these problems, a number of specific complexity classes were introduced (Valiant 1979; Hemaspaandra and Vollmer 1995). In particular, the class of counting associated with problems in NP is denoted by #P. More formally, according to Valiant (1979), the class #P denotes the set of functions counting the number of accepting paths of NP machines. This idea was generalized to arbitrary complexity classes by resorting to the following definition:

Definition 1 (*Hemaspaandra and Vollmer (1995)*)
For a standard complexity class $\mathscr{C}$, $\# \cdot \mathscr{C}$ denotes the set of functions such that for some $\mathscr{C}$-computable binary predicate R and a polynomial p it holds that for every input string x:

$$f(x) = \|\{y \mid p(\mid x \mid) =\mid y \mid \wedge R(x, y)\}\|.$$

Intuitively, each function $f \in \# \cdot \mathscr{C}$ counts the strings of polynomial length, denoted by y, with respect to the input size $\mid x \mid$, such that the predicate $R(x, y)$ holds. It is worth recalling that, as it has been noted by Hemaspaandra and Vollmer (1995), the class $\#P = \# \cdot P$.

Another useful definition will allow us to consider the cases in which we are interested in verifying lower/upper bounds to the number of solutions.

Definition 2 (*Hemaspaandra and Vollmer (1995)*)
Let $\mathscr{C}$ be a standard complexity class, then $A \in C \cdot \mathscr{C}$ if there exists a function $f \in \# \cdot \mathscr{C}$ and a polynomial-computable function g such that $x \in A \leftrightarrow f(x) \geq g(x)$.

Starting from Definition 2, it can be verified that the complexity class $PP = C \cdot P$ (Hemaspaandra and Vollmer 1995). Concerning the application of these definitions to the realm of ASP, the complexity of the Answer Set Counting problem has been extensively studied over the years (Fichte *et al.* 2016). In Table 3 we recall the complexity of the main tasks associated with the problem of checking, computing, and counting answer sets of ASP programs. The results on the complexity of decision problems are summarized in a survey by Dantsin *et al.* (2001) and derived in previous papers by Marek and Truszczynski (1991) for normal programs, Eiter and Gottlob (1995) for disjunctive programs, and Ben-Eliyahu and Dechter (1994) for the HCF programs. The results on counting were provided by Fichte *et al.* (2016).

4.2 Complexity of testing tasks

For evaluating the complexity of each assertion task we considered the case for propositional ASP programs. In particular, Table 4 reports the corresponding complexity class for each assertion task.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Table 4. *Computational complexity of the assertion tasks*

Assertions	**Head-cycle-free**	**Disjunctive**
`@noAnswerSet`	coNP-c	Π_2^P-c
`@trueInAll(atoms)`	coNP-c	Π_2^P-c
`@trueInAtLeast(atoms,k)`	PP-c	C · coNP-c
`@trueInAtMost(atoms,k)`	PP-c	C · coNP-c
`@trueInExactly(atoms,k)`	PP-c	C · coNP-c
`@constraintForAll(constr)`	coNP-c	Π_2^P-c
`@constraintInAtLeast(constr,k)`	PP-c	C · coNP-c
`@constraintInAtMost(constr,k)`	PP-c	C · coNP-c
`@constraintInExactly(constr,k)`	PP-c	C · coNP-c
`@bestModelCost(cost,level)`	Δ_2^P-c	Δ_3^P-c

Theorem 4.1
All results stated in Table 4 do hold.

In the following, we provide detailed proofs of the computational complexity of the implemented assertion tasks. Note that, some of them can be easily obtained as corollary of well-known results about standard reasoning tasks in ASP, such as *answer set existence*, *brave reasoning*, and *cautious reasoning* (Dantsin *et al.* 2001), but, to the best of our knowledge, many others, in particular tasks that we identify to be PP-complete and C · coNP-complete, are new.

In the following, assume that the program P is propositional.

`@noAnswerSet` asks whether P has no answer set, that is, $AS(P) = \emptyset$. Hence, this problem is the complementary of the answer set existence problem. Since answer set existence is NP-complete for HCF ASP programs and Σ_2^P-complete for disjunctive ASP programs (Dantsin *et al.* 2001), then the assertion is coNP-complete for HCF programs and Π_2^P-complete for disjunctive ASP programs.

`@trueInAll`. Given a set of atoms A, `@trueInAll(`A`)` asks whether each atom in the set A belongs to each answer set of P, that is, for each $M \in AS(P)$, $A \subseteq M$? Hence, this assertion task is equivalent to the cautious reasoning task. Since cautious reasoning is coNP-complete for HCF ASP programs and Π_2^P-complete for disjunctive ASP programs (Dantsin *et al.* 2001), then the assertion is also coNP-complete for HCF programs and Π_2^P-complete for disjunctive ASP programs.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

`@trueInAtLeast`. Given a set of atoms A and a positive integer k, `@trueInAtLeast`(A, k) asks whether there exist $M_1, \ldots, M_k \in AS(P)$ such that $A \subseteq M_i$, with $1 \leq i \leq k$. Note that, this problem is equivalent to check if, the ASP program $P \cup \{\leftarrow not\ a \mid a \in A\}$ has at least k answer sets. Now, it is known that the problem of answer set counting is #P-complete for HCF programs and $\#\cdot$coNP-complete for disjunctive programs (Fichte *et al.* 2016). Hence, the corresponding decision problem of checking if the number of answer sets is at least k is PP-complete for HCF programs and $\mathsf{C} \cdot$ coNP-complete for disjunctive programs.[1]

`@trueInAtMost`. Given a set of atoms A and a positive integer k, `@trueInAtMost`(A, k) asks whether there exist at most k answer sets, $M_1, \ldots, M_k \in AS(P)$ such that $A \subseteq M_i$, with $1 \leq i \leq k$. Moreover, this problem is equivalent to check if, the ASP program $P \cup \{\leftarrow not\ a \mid a \in A\}$ has at most k answer sets. It is known that, the decision problem of checking if the number of satisfying assignments for a given Boolean formula is at most k, has the same complexity of checking if the number of satisfying assignments is at least k (Valiant 1979). Thus, the assertion task @trueInAtMost has the same computational complexity of @trueInAtLeast.

`@trueInExactly`. Given a set of atoms A and a positive integer k, `@trueInExactly`(A, k) asks whether there exist exactly k answer sets, $M_1, \ldots, M_k \in AS(P)$ such that $A \subseteq M_i$, with $1 \leq i \leq k$. Note that the assertion `@trueInExactly`(A, k) is true if, and only if, `@trueInAtLeast`(A, k) and `@trueInAtMost`(A, k) are true. Therefore, the assertion task `@trueInExactly` has the same computational complexity of the assertion tasks `@trueInAtLeast` and `@trueInAtMost`.

`@constraintForAll`. Given a set of constraints C, `@constraintForAll` asks whether each answer set of P is also an answer set of $P \cup C$, that is, $AS(P) = AS(P \cup C)$. This problem is equivalent to ask whenever $P \cup C'$ has no answer set, where $C' = \{fail \leftarrow B(c) \mid c \in C\} \cup \{\leftarrow not\ fail\}$. Indeed, assume that each answer set M of P is also an answer set of $P \cup C$. Hence, M models each constraint in C. Therefore, $B(c)$ is never satisfied. Thus, $fail$ cannot be inferred, and it must be false. Hence, the constraint $\leftarrow not\ fail$ is not satisfied, and $P \cup C'$ has no answer set. The other implication is similar. In conclusion, this problem is the complementary of the answer set existence problem (Dantsin *et al.* 2001). Therefore, the assertion is coNP-complete for HCF programs and Π_2^P-complete for disjunctive programs.

`@constraintInAtLeast`. Given a set of constraints C and a positive integer k, `@constraintInAtLeast` asks whenever there exist at least k answer sets of P that satisfy the set of constraints C, that is, if $P \cup C$ has at least k answer sets. Again, since the problem of answer set counting is #P-complete for HCF programs and $\#\cdot$coNP-complete for disjunctive programs (Fichte *et al.* 2016), then, the problem of checking if the number of answer sets is at least k is PP-complete for HCF programs and $\mathsf{C} \cdot$ coNP-complete for disjunctive programs.

[1] For further details on complexity classes related to counting problems, we refer to Hemaspaandra and Vollmer (1995).

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

`@constraintInAtMost`. Given a set of constraints C and a positive integer k, `@constraintInAtMost` asks whenever there exist at most k answer sets of P that satisfy the set of constraints C, that is, if $P \cup C$ has at most k answer sets. Note that this problem is equivalent to ask whenever $P \cup C$ has not at least $k+1$ answer sets, that is the complementary problem of `@constraintInAtLeast`, and it has the same complexity of `@constraintInAtLeast`.

`@constraintInExactly`. Given a set of constraints C and a positive integer k, `@constraintInExactly` asks whenever there exist exactly k answer sets of P that satisfy the set of constraints C, that is, if $P \cup C$ has exactly k answer sets. Hence, this problem is equivalent to ask whenever $P \cup C$ has at least k answer sets and has at most k answer sets. Therefore, it has the same complexity of `@constraintInAtLeast` and `@constraintInAtMost`.

`@bestModelCost`. Given a set of weak constraints W, and two positive integers c and l, `@bestModelCost(c,l)` asks whenever there is an optimal answer set M of $P \cup W$ such that its cost is c at level l. It is easy to see that the computational complexity of this task is equal to that of deciding the existence of an optimal answer set for programs with weak constraints, which is Δ_2^P-complete for HCF programs and Δ_3^P-complete for disjunctive programs (Buccafurri *et al.* 2000).

5 The ASP-WIDE environment

The ASP-WIDE environment implements this paper's unit testing mechanism and the annotation language. While command line tools are efficient to use, the focus was to build an environment containing a code editor with syntax checking, syntax highlighting, and execution/testing capabilities. This integrated development tool offers a convenient environment for writing, executing and testing answer set programs. Since web-based environments, not only for logic programming, but also for conventional languages, are widely used, ASP-WIDE is based mostly on web technologies.

5.1 Architecture and implementation

As many modern web-based applications, ASP-WIDE consists of a front-end, which is built using the Angular framework, and a back-end implemented in Java utilizing the Spring framework. The communication between front-end and back-end is realized with HTTP-Requests transmitting JSON data. The overall architecture is depicted in Figure 3; below we detail the two main components of ASP-WIDE. For a deeper (more technical) description of the implementation we refer the reader to the Master Thesis of Berei (2019).

The front-end of the development environment was built with the Angular framework (see `https://angular.io/`), which is a web technology for building comprehensive web applications. The UI components are partly based on Angular Material components (cfr. `https://material.angular.io/`), while the code editor utilizes the Monaco editor known from Visual Studio Code (see `https://microsoft.github.io/monaco-editor/`

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

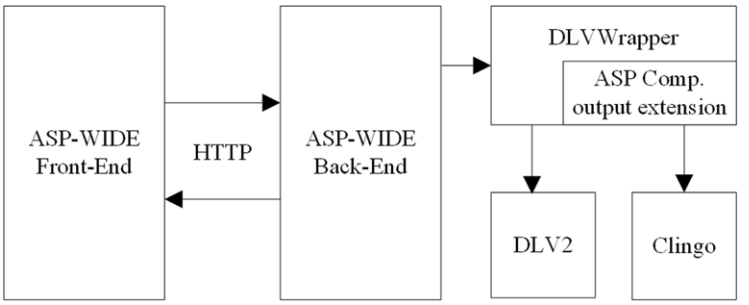


Fig. 3. Architecture of ASP-WIDE.

`index.html`). The Monaco editor comes with several built-in features, like the possibility to define a custom syntax highlighting using Monarch, but also syntax errors/warnings can be displayed efficiently. Thus ASP-WIDE environment is restricted to run inside a browser that renders HTML with CSS and executes JavaScript in background.

The back-end of ASP-WIDE consists of a REST web service implemented in Java using the Spring framework. In particular, web controllers receive web requests and execute the required logic in order to fulfill actions for the front-end. Some of these actions are: (*i*) file management operations; (*ii*) checking the syntax of the program; (*iii*) executing programs and returning answer sets; and (*iv*) executing unit tests and returning test results. Following "Separation of concerns," a well-known design principle, certain aspects of the back-end have been split into three modules, namely *Development*, *Execution,* and *File Management*.

The File Management module deals with typical CRUD operations for files, while the Development and Execution modules implement the *testing engine* that is responsible for syntax checking and execution of programs and tests.

The main steps of the proposed test engine can be summarized as follows: (i) parsing of the ASP-Core-2 input language including an extension for the annotation language of this paper; (ii) interpreting parsed assertions in order to generate the tester program; (iii) execution of ASP systems; and (iv) evaluating test results.

More precisely, the test engine starts by parsing the entire ASP program, together with provided test annotations, by means of a parser generated with JavaCC (see `https://javacc.org/`). In particular, such a parser is obtained from a grammar definition of the input language and provides the possibility to implement a visitor pattern on top of the grammar tree by means of JJTree tool included in JavaCC. During parsing, exceptions are caught by our engine in order to report to the users all useful information (i.e., error line and column) to identify the error in the provided ASP code. If no errors happen at this stage, an object-oriented representation of the parsed program and the parsed annotations is obtained.

In particular, this is encapsulated in the Java class named `TestSuite` reported in Figure 4. By exploiting such a virtual model, rules and blocks of the original program are stored only once and are shared among the different tests, while test-specific data such as assertions, input facts, constraints, and more, are enclosed in the different `Test` objects.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

```
public class TestSuite{
    private HashMap<String, Block> blocks;
    private HashMap<String, String> rules;
    private ArrayList<Test> tests;
    public TestSuite() {
        this.blocks = new HashMap<String, Block>();
        this.rules = new HashMap<String, String>();
        this.tests = new ArrayList<Test>();
    }
    // further functions and definitions
}
```

Fig. 4. `TestSuite` class definition.

```
    private AssertionResult checkAssertion(String code, AssertTrueInAtMost
        assert, SolverType st) {
        AssertionResult ar = new AssertionResult();
        ar.setName(assert.getClass().getSimpleName());
        ar.setExecutedCode(code);
        String[] atoms = assert.getAtoms().split("\\.");
        StringBuilder sb = new StringBuilder(code);
        for(String atom: atoms) {
            sb.append(":- not ");
            sb.append(atom.trim());
            sb.append(".\n");
        }
        ar.setExecutedCode(sb.toString());
        ExecutableFile testFile = new ExecutableFile("checkAssertionTrueAtMost",
            ar.getExecutedCode(), "");
        testFile.setSolverType(st);
        ExecutionResult er = this.executionLogic.executeCode(testFile,
            (assert.getAssertCount()+1));
        ar.setExecutionOutput(er.getResult());
        if(er.getModels() != null && er.getModels().size() <=
            assert.getAssertCount()) {
            ar.setSucceeded(true);
        }
        return ar;
    }
```

Fig. 5. Implementation of the `@trueInAtMost` assertion.

Starting from the obtained `TestSuite` each test is executed separately. In order to better understand the testing workflow, Figure 5 reports the code that implements the execution of a test verifying a `@trueInAtMost` assertion.

The function `checkAssertion` receives three parameters, an ASP program (`code`), an assertion to verify (`assert`), and the ASP system that should be used (`st`). Intuitively, the ASP program is obtained from the generated `TestSuite`, by fetching all the rules indicated (explicitly or indirectly by using block definitions) in the scope attribute of the test while the assertion object stores the answer set count k, and the list of *atoms* that has to be true in at most k answer sets.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Table 5. *Implementation of the assertions. P denotes the scope (i.e., the subprogram to test), T_P the program built to implement the assertion, C a constraint, A a set of atoms, k, c, l are integers*

Assertions	**Tester program T_P**	**Test output**
`@noAnswerSet`	P	Return *fail* if T_P admits answer sets, *pass* otherwise.
`@trueInAll(`A`)`	$P \cup \bigcup_{a \in A}\{\leftarrow\ a\}$	Return *fail* if T_P admits answer sets, *pass* otherwise.
`@trueInAtLeast(`A`,`k`)`	$P \cup \bigcup_{a \in A}\{\leftarrow\ not\ a\}$	Return *pass* as soon as the solver on T_P outputs k answer sets, *fail* otherwise.
`@trueInAtMost(`A`,`k`)`	$P \cup \bigcup_{a \in A}\{\leftarrow\ not\ a\}$	Return *pass* if the solver terminates on T_P outputting at most k answer sets, *fail* otherwise.
`@trueInExactly(`A`,`k`)`	$P \cup \bigcup_{a \in A}\{\leftarrow\ not\ a\}$	Return *pass* if the solver on T_P outputs k answer sets, *fail* otherwise.
`@constraintForAll(`C`)`	$P \cup \{f \leftarrow\ C;\ \leftarrow\ not\ f\}$	Return *pass* if T_P admits no answer set, *fail* otherwise.
`@constraintInAtLeast(`C`,`k`)`	$P \cup \{C\}$	Return *pass* as soon as the solver on T_P outputs k answer sets, *fail* otherwise.
`@constraintInAtMost(`C`,`k`)`	$P \cup \{C\}$	Return *pass* if the solver terminates on T_P outputting at most k answer sets, *fail* otherwise.
`@constraintInExactly(`C`,`k`)`	$P \cup \{C\}$	Return *pass* if the solver on T_P outputs k answer sets, *fail* otherwise.
`@bestModelCost(c,l)`	P	Return *pass* if the optimal answer set of T_P has a cost of c at level l, *fail* otherwise.

According to the transformation described in Table 5, in our example, the tester program is obtained by adding to the input program a constraint for each atom that has to be true (Figure 5, lines 6-11).

Then, the tester program is evaluated by the ASP system `st` (Figure 5, lines 12–15). This evaluation is implemented in the Execution module that is built on top of the library DLVWrapper (Ricca 2003). In particular, we extended the DLVWrapper library to handle any ASP systems supporting the output format of the last ASP Competition (Gebser *et al.* 2017), such as, for example, Clingo (Gebser *et al.* 2019). According to the selected

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

```
protected void startRun throws IOException, DLVInvocationException {

    // some preparation ...

    List<String> input = new ArrayList<String>();
    input.add(DLVWrapper.getInstance().getPath());
    try {
        if (isDLV2){
            addOption("--silent");
        }
        else if(isClingo) {
            addOption("--outf=1");   // competition output format
            addOption("--quiet=0,0"); // output configuration
        }
        else{ // DLV
            addOption("-silent");
        }
    } catch (DLVInvocationException e) {
        e.printStackTrace();
    }
    input.addAll(options);
    input.addAll(inputProgram.getFiles());

    // starting the ASP system in a process ...

}
```

Fig. 6. Customization of DLVWrapper according to the used ASP system.

ASP system, DLVWrapper executes it as a command line tool with its specific arguments. A snapshot of the code that prepares the execution engine is reported in Figure 6.

There are several benefits that come with the DLVWrapper. One of them is that it provides a layer of objects that abstracts the interaction with the solver and simplifies the implementation. Moreover, it supports asynchronous program execution with the possibility to register callback functions (ModelHandlers) for found answer sets, which allows handling large results without saturating the memory of the caller.

The output produced by the ASP system (i.e., the list of answer sets of the tester program) is encapsulated in an object of the class `ExecutionResult` that is further used to check the tested assertion. In our example, if at most k answer sets have been found out of the $k+1$ requested by the execution module, then the assertion succeeded (Figure 5 lines 16–19). Assertion results are then provided as output to the user request and are visualized by the front-end.

5.2 User interface

The user interface of ASP-WIDE is depicted in Figure 7. The design of the environment was inspired by ASPIDE and features four main areas of interaction with the user:

(*i*) The toolbar on the top of the environment;
(*ii*) The workspace or file explorer on the left side;
(*iii*) The code editor in the middle/right area (with open tabs on the top); and
(*iv*) The output area on the bottom, which shows answer sets and test results.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

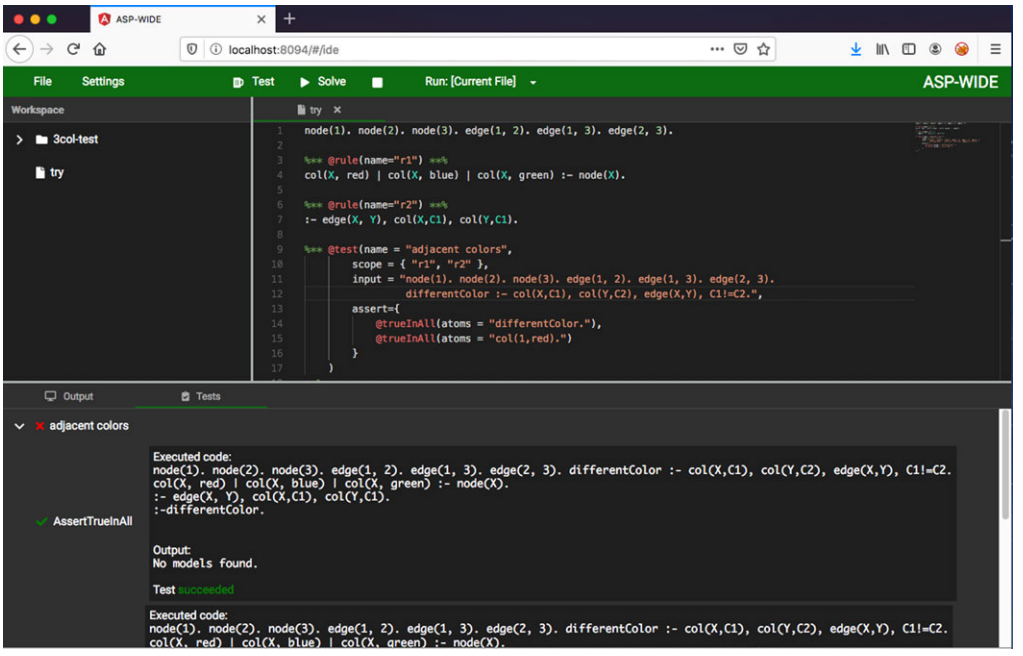


Fig. 7. The ASP-WIDE interface.

The toolbar features usual menus for handling files and settings; two buttons to run files and tests and a drop-down list for handling solver execution configurations (called run configuration as in ASPIDE), respectively. Programs can be organized in projects, which appear and can be operated as folders on the workspace explorer. Acting on the workspace explorer one can open files. The opened file is displayed in the code editor in the middle/right area of the screen. The code editor features syntax highlights and code completion, that is, it suggests how to complete predicate names and variables, to assist program and test case development. Errors and warnings are also immediately displayed, and the modifications are automatically saved, as is customary in modern web-based interfaces. The execution of programs can be controlled by managing run configurations, where one can specify the files to include, the solver to use and the command line parameters. Executing the current file or unit tests (if there are any) is done by clicking on the buttons “Solve” or “Test,” which can be found in the middle of the Toolbar. In case one just wants to run some files together, the interface allows the user to select files from the workspace explorer, right click, and select “Run directly” from a drop-down menu. More involved configurations can be set up acting on the run configuration window. The result of the execution (of tests and programs) is shown in the bottom part of the environment, containing output area. The result of test case execution outlines, for each test, the result (using red color for failed tests, and green color for passed ones), and for each test it is possible to inspect the execution details, and a witness of the result.

5.3 Availability

The ASP-WIDE environment can be installed as a standalone application on any computer with a modern (java script-enabled) web browser and Java 8 installed. ASP-WIDE

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

can be downloaded from: http://www.mat.unical.it/ricca/asptesting/asp-wide_1.0-beta4.zip, and the sources are distributed under GPL license (send an email to ricca@mat.unical.it).

6 Experimental evaluation

In this section we present an experimental evaluation aimed at assessing the ASP-WIDE implementation of the test case execution environment. On the one hand, the experiment aims to demonstrate the practical usability of the ASP-WIDE language by showcasing its ability to effectively identify and capture bugs in real-world use cases. Also, the experiment shows that ASP-WIDE exhibits good performance in test case execution. On the other hand, the experiment highlights the importance of having a complexity-aware implementation of test case execution.

6.1 Development of the test suites

The first step of the experimental campaign was the development of a benchmark covering some concrete use case problems. In particular, we considered three well-known problems used in the literature both to introduce ASP and to test ASP systems in competitions (Gebser *et al.* 2020; 2017; 2012): (HP) Hamiltonian Path, (FF) Fast Food problem, and Quantified Boolean Formulas with two quantifiers (2QBF).

The HP problem involves determining the existence of a finite set of vertices $v_1, \ldots, v_k$ of a directed graph $G = \langle V, E \rangle$, such that: (i) every node $v \in V$ appears exactly once, and (ii) for each $1 \leq i < k$, $(v_i, v_{i+1}) \in E$. HP is a well-known problem in literature belonging to the class of NP-complete problems. HP played a relevant role in the ASP literature because it is one of the canonical examples of non-tight (Erdem and Lifschitz 2003) encoding (i.e., with positive recursive definitions).

FF is an optimization problem introduced in ASP competitions (Gebser *et al.* 2020). The objective of FF is to find a k-subset of n restaurants that serve as depots in such a way the distance between each restaurant and the closest depot is minimized. FF features an encoding that follows the guess&check methodology and makes use of aggregates and, importantly, of weak constraints to model the optimality condition.

2QBF is the problem of deciding the satifiability of a quantified boolean formula with two quantifiers. More in detail, the problem is to check satisfiability of a formula of the form $\exists X \forall Y \phi$, where X and Y are sets of propositional variables, and ϕ is a formula in disjunctive normal form. Recall that a formula ϕ is in disjunctive normal form if it is of the form $C_1 \vee \cdots \vee C_n$ where C_i is a conjunction of propositional literals (i.e., propositional variables or their negation) for all $i = 1, \cdots, n$. 2QBF is Σ_2^P-complete. Note that, 2QBF is the canonical example of the application of the saturation programming technique (Eiter and Gottlob 1995), that is used to model problems belonging to the second level of the Polynomial Hierarchy (PH).

All in all, the experiment covers many of the main constructs of ASP (i.e., recursive definitions, constraints, aggregates) as well as the two main programming methodologies (i.e., guess&check and saturation). Moreover, it considers representatives for both decision and optimization problems belonging to different levels of the PH. The encodings of HP, FF and 2QBF have been introduced in the literature and used also in ASP

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Table 6. *Employed assertions for each domain*

Assertion\Domain	Hamiltonian Path	Fast Food	2QBF
`@constraintForAll`	✓	✓	✓
`@constraintInAtLeast`	-	-	✓
`@trueInAll`	-	-	✓
`@trueInAtLeast`	-	✓	✓
`@trueInExactly`	✓	-	-
`@noAnswerSet`	✓	-	✓
`@bestModelCost`	-	✓	-

Table 7. *Modifications applied to generate mutants*

Modification\Domain	Hamiltonian Path	Fast Food	2QBF
renamePredicates	✓	✓	✓
deleteRule	✓	✓	✓
deleteLiteral	-	✓	✓
addDefaultNegation	✓	-	✓
swapTerms	✓	-	✓
changeAggregates	-	✓	-
changeMathOperators	-	✓	-
swapDefaultNegation	-	✓	-

competitions (Gebser *et al.* 2020). Thus, this benchmark can be pragmatically considered to be made of concrete and representative use cases of ASP programs.

A test suite for the above-mentioned problems has been developed by using the testing language presented in Section 3. We have created unit test specifications for the considered problems: HP, FF, and 2QBF, consisting of 6, 5, and 6 cases, respectively. Table 6 summarizes the assertions used in the test cases for each considered problem. We observe that `@constraintForAll` assertion, checking properties that hold in all answer sets, was used in all the three cases; whereas `@trueInExacly` revealed to be useful only in HP. As expected, `@bestModelCost` was used only in FF, which is the only optimization problem. All the remaining assertions were used in at least two cases. The test case specifications were accompanied by random problem instances, forming a test suite consisting of 88 unit tests for HP, 23 for FF, and 459 for 2QBF.

In order to assess our test suite, we performed a mutation analysis (Oetsch *et al.* 2012). The latter involves making various modifications to the *original* (correct) encoding, resulting in the creation of *mutants*. The purpose is to evaluate the effectiveness of the test suite in detecting potential bugs by executing it on these mutants. More in detail, we utilized the mutation model and tool specifically designed for ASP, which was introduced by Oetsch *et al.* (2012). In our work, we refer to this tool as ASP-Modificator , which provides a pool of possible modifications that can be applied, possibly multiple times, to an input program P, in order to generate different *mutants* of P. Among possible modifications supported by ASP-Modificator, we selected those that can be applied to the considered problem encoding according to their syntactic features. In particular, Table 7 reports the list of modification used for generating mutants for each problem.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

By applying such transformations, we obtained nine altered mutants (i.e., $M_1, \cdots, M_9$) for each problem encoding.

6.2 Experiment setup

In order to ensure the reproducibility of our results, we provide detailed information below regarding the methods compared and the execution environment used in the study.

Compared methods. In the experiment we run two implementations. The first is the test case execution engine of ASP-WIDE, configured with the ASP system CLINGO (Gebser *et al.* 2019). Recall that, ASP-WIDE produces for each test case a tester program (as specified in Table 5), and then analyzes the output of the ASP solver on the tester program to present the results to the user. This behavior has been extracted from the environment and incorporated into a command line tool, which is then executed in isolation within the experimental environment.

Another implementation that was executed is a re-engineering of the algorithms employed in ASPIDE, which we denote as ASPIDE-LIKE. Note that, ASPIDE does not provide direct access to the internal engine, thus it has been faithfully re-implemented. More in detail, in the ASPIDE approach the semantics of the assertions is implemented straightly. In few words, the instance under test is assembled according to the test case specification and fed to an ASP system (i.e., CLINGO in this experiment). In turn, a Java thread processes the answer sets to check whether the assertion to test is satisfied. This approach is not complexity-wise optimal in most cases. For example, checking whether a property holds in all answer sets requires a full enumeration in ASPIDE, whereas it can be done in a single call to an ASP solver in ASP-WIDE (cfr. Table 5). ASPIDE-LIKE implementation is "complexity-wise optimal" only when analyzing the first solution generated by the ASP system is adequate, for example, in the case of `@bestModelCost`.

Hardware and software resources. All the experiments were executed on a machine equipped with Xeon(R) Gold 5118 CPUs, running Ubuntu Linux (kernel 5.4.0-77-generic). Memory and time were limited to 4 gigabyte and 300 s, respectively. The evaluation of ASP programs was performed by running CLINGO (Gebser *et al.* 2019) version 5.4.0.

Benchmarks availability. All the material (encoding, instances, tests, executables, etc.) needed to reproduce the experiments can be downloaded from `https://osf.io/6hxdn/?view_only=2a2067f712ac438cb3b6a9c7aa528331`.

6.3 Results

In the following the results obtained by running ASP-WIDE and ASPIDE-LIKE are analyzed.

Efficacy of the testing methodology. First, we discuss about the efficacy of the approach (i.e., we answer to the question whether our (small) test suite is sufficient to cover all mutants) and check whether it delivers acceptable performance in terms of executions time.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Figure 8 reports an histogram for each considered problem. More specifically, each histogram reports one bar for each mutant. The green (resp. red) color indicates succeeded (resp. failed) tests case executions for each generated mutant. A test succeeds whenever an assertion is satisfied and fails otherwise.

As expected, the bar corresponding to the original (i.e., bug-free) encoding are only colored in green, which means that no assertion fails on a correct program. Moreover, some assertion fails in all the mutants (note that the red color is visible in bars corresponding to mutants). This means that all the bugs introduced in the encodings for the three problems could be detected.

Given the high *worst-case* complexity of checking assertions, and the intrinsic complexity of benchmark problems (recall that HP is NP-complete, FF is NP hard and 2QBF is Σ_2^P-complete), one might wonder whether this result can be obtained in a reasonable time.

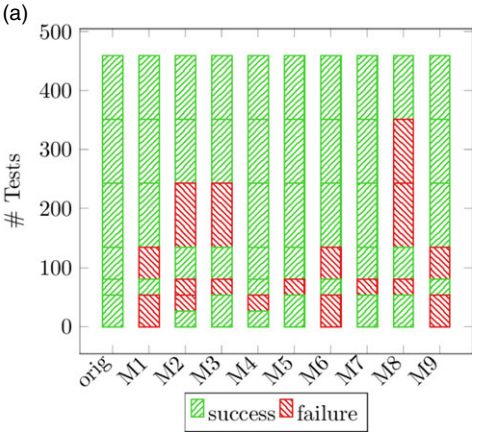
The average execution time employed by ASP-WIDE is reported in Table 8. Test cases in HP, FF, and 2QBF were completed in average 0.003 s, 0.05 s, and 0.026 s, respectively. This average performance is clearly acceptable and reveals that the majority of assertion tests is basically instantaneous. (A more detailed analysis on these results is provided in the next paragraph).

We remark that this result can be seen as a confirmation of the small-scope hypothesis (Oetsch *et al.* 2012). As observed by Oetsch *et al.* (2012), to detect a bug in an ASP program, it is sufficient to "analyze programs after grounding them over a small domain." As a consequence, regardless of the high worst-case complexity associated with assertion testing tasks, unit testing remains effective in identifying bugs due to the reasonable solving time of state-of-the-art ASP systems. On the other hand, it is important to take into account the high worst-case complexity of assertion testing when designing test cases. Indeed, complexity results tell us that utilizing large instances in a test case can render it ineffective and pointless, as waiting for the results may become unreasonably time-consuming. A rule of thumb for devising good unit tests is to concentrate in devising smart test cases that run over small problem instances.

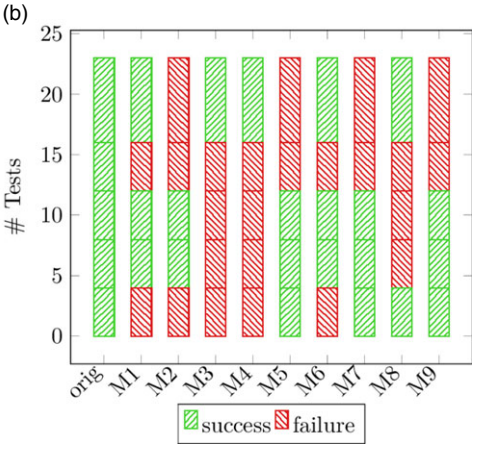
Comparison with ASPIDE. Now, ASP-WIDE and ASPIDE-LIKE performance are compared. This is done to measure the improvements that can be obtained by delivering an implementation, like ASP-WIDE, that carefully considers the complexity of the task at hand. Also an in-depth analysis of the performance of the systems running each different annotation type is performed.

Obtained results are summarized by Figure 9 and Table 8. In particular, Table 8 reports, for each benchmark and testing method, number of solved instances ($\#solved$), average running time ($avg(T)$), and standard deviation $\sigma(T)$ computed over all the test case executions (column $\#$). While ASP-WIDE consistently generates results within the time limit for all test cases, ASPIDE fails to do so. Therefore, average and standard deviation provided in Table 8 take into account the execution time for solved instances and double the time limit (i.e., 600 s) for unsolved instances. This performance measure, commonly known as the PAR-2 score, is widely used in systems comparison (Froleyks *et al.* 2021). From Table 8 it is evident that the ASP-WIDE outperforms by orders of magnitude ASPIDE-LIKE. Nonetheless, the values of the standard deviation suggest that some specific instances are more demanding.

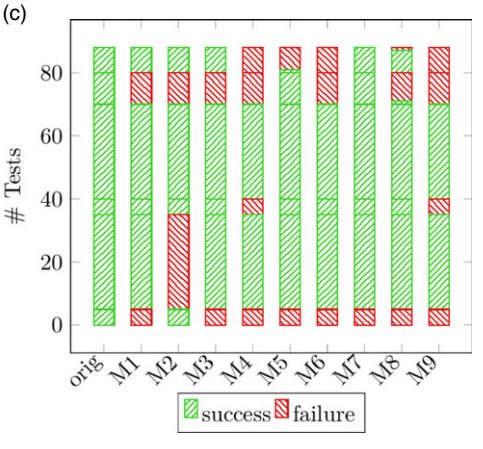
https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press



Unit tests (2QBF)



Unit tests (FF)



Unit tests (HP)

Fig. 8. Successful and failed unit tests for each domain.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Figure 9 reports the aggregate performance of both ASP-WIDE and ASPIDE-LIKE in form of cactus plots, one for each assertion type (executed on all encodings). Recall that a line in a cactus plot contains a point (X, Y) if a given method is able to solve X instances within Y seconds. From Figure 9a–9g it emerges that ASP-WIDE is very efficient in checking all the assertion types. More precisely, ASP-WIDE outperforms ASPIDE-LIKE, which often reveals an exponential-like behavior. Whereas, in the case of `@noAnswerSet` and `@bestModelCost`, the two approaches performs equally since the underlying check is the same (cfr. Figure 9f and 9g). Here, the first answer of the system is sufficient to check the assertions, as the corresponding task has the same computational complexity of checking the existence of an (optimal) answer set.

The overall gap in performance is evident in Figure 9h, which compares the two approaches over all executions. More in detail, we report that ASP-WIDE run 5700 tests within 135 s by using, on average, 2.75MB of memory, while, ASPIDE-LIKE run 4373 tests in 10 h by using, on average 94MB of memory.

7 Related work

The first paper approaching the problem of systematic testing of ASP programs is (Janhunen *et al.* 2010), where a general framework for structure-based testing of answer set programs, encompassing the definition of test coverage notions for ASP programs, has been proposed. In (Janhunen *et al.* 2010) the complexity issues related to coverage problems and the inherent complexity of relevant decision problems were also studied. An experimental comparison of basic strategies for random testing and structure-based testing of ASP programs has been presented (Janhunen *et al.* 2011). Results obtained in this latter indicate that random testing is quite effective in catching errors provided that sufficiently many admissible test inputs are considered. It has been empirically demonstrated (Oetsch *et al.* 2012) that the small-scope hypothesis of traditional testing holds also in the case of ASP programs. That is, many errors can be found by testing a program w.r.t. test inputs considering a small number of objects (i.e., from a small scope). More recently, a new tool for random-based testing of ASP programs, called Harvey, has been described (Greßler *et al.* 2017). In Harvey random testing for ASP has been implemented using ASP itself (i.e., both test-input generation and determining test verdicts is done employing ASP). Harvey achieves uniformity of the test-input selection by using XOR streamlining (Gomes *et al.* 2007) a technique used also in the area of SAT solving (Gomes *et al.* 2006). The methods mentioned up to now focus on the problem of generating automatically test suites for ASP programs that are sufficient to identify defects, *after correct programs have been written*. These tools are thus particularly useful in cases in which one wants to improve an encoding and use a natural (but less efficient) encoding, to check whether a more complicated (but efficient) one is being developed. On the other hand, as outlined in the introduction, the goal of unit testing in software development is to drive the implementation toward working software also when *no previous solution exists*. Indeed, in TDD, test cases are derived from the requirements even before writing the source code (Fraser *et al.* 2003; Beck 2002). Nonetheless, automatic test generation can be combined with unit testing, for example, in case one is evolving a solution to meet some non-functional requirement such as efficient computation.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

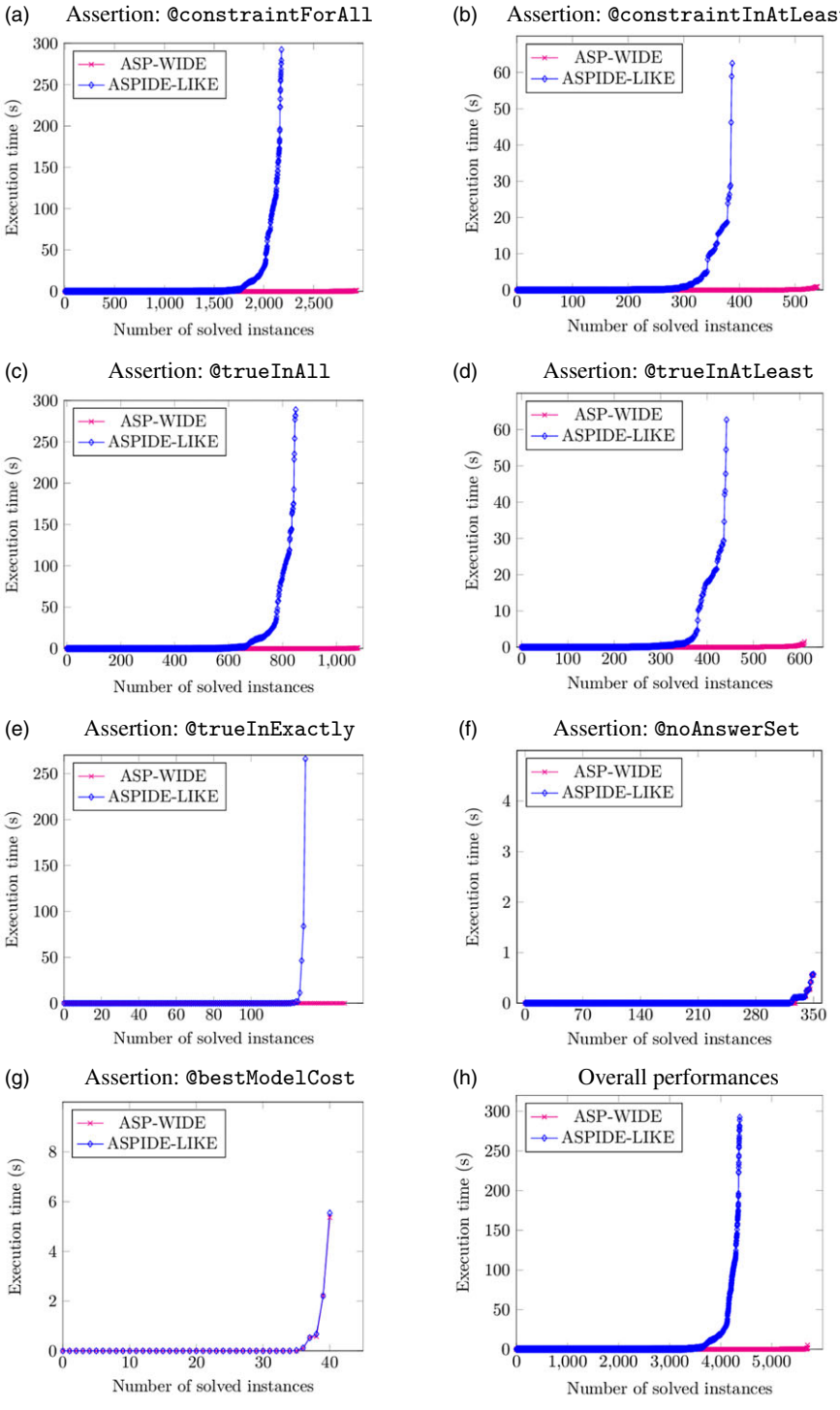


Fig. 9. Assertion-based comparison between ASP-WIDE and ASPIDE-LIKE methods.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Table 8. *Comparison between ASP-WIDE and ASPIDE-LIKE*

Benchmark	Test cases	#	ASP-WIDE			ASPIDE-LIKE		
			#solved	$avg(T)$	$\sigma(T)$	*#solved*	$avg(T)$	$\sigma(T)$
Hamiltonian Path	6	880	880	0.003	0.018	565	216.750	286.749
Fast Food	5	230	230	0.050	0.398	212	50.078	160.661
2QBF	6	4590	4590	0.030	0.102	3596	137.275	244.897

Focusing on unit testing, the first implementation of an ASP-specific solution was presented and included in the comprehensive development tool ASPIDE (Febbraro *et al.* 2011a). This implementation utilizes rule naming inside of ASP comments in combination with a test definition language for specifying test cases. While rule naming can be accomplished in an annotation-like manner, which does not interfere with program executability, the specification of test cases required a separate test file and a dedicated syntax (Febbraro *et al.* 2011a). Since adding meta-information to programs in form of annotations is known from conventional programming languages as C# and Java, a purely annotation-based test case specification for answer set programs is desirable. With (De Vos *et al.* 2012) a language for annotating answer set programs (called LANA) is presented. Although LANA is not solely devoted to testing, it does address test case definition inside of ASP comments. Despite fully relying on annotations, its implementation (called ASPUnit) requires each unit test to be defined in a separate file and was never integrated in a development environment (to the best of our knowledge). Consequently the desire for a lightweight test definition mechanism that is purely annotation-based and does not necessarily require additional files or external tools that are not included in an environment dedicated to assisted programming remained unfulfilled. Moreover, it is important to note that, from a pure syntactic perspective, our assertion language closely resembles the one of ASPIDE, which is also based on Java-like conventions, and uses similar names for the assertions. On the other hand, our language syntax-wise is significantly different from LANA, which follows its own syntactic conventions and can provide the same assertions.

The annotation language presented in this paper, albeit inspired by existing proposals, presents a new syntax that differs from both ASPIDE and LANA proposals and recalls the well-known annotation style of Java. Since one of our design goals was to keep it simple while considering all the most important features, our language does not support (by design) some of the ASPIDE-specific options (such as automatic extraction of program modules and run configuration management), and some of the LANA-specific options (such as pre-/post-conditions and signatures). We discarded those that are not strictly related to program testing (e.g., signatures), can be simulated (e.g., pre-/post-conditions), are implementation-specific (run configuration management), and have a not-so-obvious semantics (automatic expansion of program modules). Indeed, automatic expansion of program modules in ASPIDE allows the automatic extension of a block under test with the rules from the original program up to the point that a modularity condition is satisfied, such as the splitting condition (Lifschitz and Turner 1994) or the more precise conditions of (Janhunen *et al.* 2009). In our experience this feature

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

augments the program under test in a way that is not obvious to the programmer, thus we decided to discard this feature. Alternative ways of verifying the correctness of programs with input and output have been recently proposed (Fandinno *et al.* 2020), that could be considered for integration in our framework.

We remark that our approach allows (as first suggested by De Vos *et al.* (2012)) to inline tests with code. This choice brings advantages, for example, makes it easy to pack and distribute ASP programs, that are often contained in a single file, with tests. However, the programmer can decide to keep tests and code in separate files, which might be convenient in some cases (e.g., the ASP code is divided in several files); thus our approach can deliver the maximum flexibility in the organization of a project.

Finally, we observe that our unit testing language has been conceived for ASP-Core-2, nonetheless, it can be applied -almost as it is- to any extension of ASP, such as the richer languages supported by Clingo (Gebser *et al.* 2019) and DLV 2.0 (Alviano *et al.* 2017), DLVHEX (Eiter *et al.* 2018), ASPMT (Bartholomew and Lee 2019; Shen and Lierler 2018), CASP (Mellarkod *et al.* 2008; Balduccini and Lierler 2017; Banbara *et al.* 2017), and SPARC (Balai *et al.* 2013). Indeed, annotations (in comments) do not interfere with the specifications. Moreover, our approach could be complemented with techniques for rushing and strolling among answer sets (Fichte *et al.* 2022).

8 Conclusion

Unit testing frameworks are nowadays considered best practice in all modern software development processes. The development of ASP applications can be accelerated by resorting to unit testing frameworks, as it happens for all known programming languages. In this paper we revisit unit testing in ASP by proposing a new unit test language that unifies the strengths of previous approaches. The new language allows the development of test cases inline with ASP code, keeps the style of expressing test case conditions from ASPIDE and is annotation-based as LANA. Moreover, it features a refreshed syntax that is nearer to the JUnit framework and should look more familiar to developers that are accustomed to XUnit style languages. Importantly, the new unit testing language is implemented in a novel web-based development environment for ASP, which supports TDD of ASP programs. Another contribution of the paper is a complete identification of the computational complexity of the tasks associated to unit testing of ASP programs. To the best of our knowledge, this overview was never done in the literature and contains both known and novel results.

Despite the high worstcase complexity of testing ASP programs, thanks to the small-scope hypothesis, one can define effective test suites with good performance in practice. Moreover, the experiment reported in the paper reveals the importance of devising a complexity-aware implementation, as ASP-WIDE outperforms the more naive approach of ASPIDE (Febbraro *et al.* 2011b).

As far as future work is concerned, we are improving the ASP-WIDE environment by implementing additional features, such as quick fixes, code templates, support for debuggers that are available in more mature IDEs for ASP (Febbraro *et al.* 2011a; Busoniu *et al.* 2013); moreover we are evaluating the possibility of resorting to a dedicated answer set counting system (Fichte *et al.* 2016) to implement test case conditions that are PP-complete and $\mathsf{C} \cdot \mathsf{coNP}$-complete.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

References

Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P. and Zangari, J. 2017. The ASP system DLV2. In *Logic Programming and Nonmonotonic Reasoning – 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, M. Balduccini and T. Janhunen, Eds. Lecture Notes in Computer Science, vol. 10377. Springer, 215–221.

Amendola, G., Berei, T. and Ricca, F. 2021. Testing in ASP: revisited language and programming environment. In *Logics in Artificial Intelligence – 17th European Conference, JELIA 2021, Virtual Event, May 17–20, 2021, Proceedings*, W. Faber, G. Friedrich, M. Gebser, and M. Morak, Eds. Lecture Notes in Computer Science, vol. 12678. Springer, 362–376.

Balai, E., Gelfond, M., and Zhang, Y. 2013. Towards answer set programming with sorts. In *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 135–147.

Balduccini, M. and Lierler, Y. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *Theory and Practice of Logic Programming 17,* 4, 462–515.

Banbara, M., Kaufmann, B., Ostrowski, M. and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming 17,* 4, 408–461.

Bartholomew, M. and Lee, J. 2019. First-order stable model semantics with intensional functions. *Artificial Intelligence 273*, 56–93.

Beck. 2002. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12,* 1–2, 53–87.

Berei, T. 2019. Annotation-based testing for answer set programming. *Master Thesis - University of Calabria*, 1–65. URL: https://www.mat.unical.it/ricca/downloads/Annotation-based_Testing_for_ASP_Berei_Tobias.pdf.

Bezerra, C. and Freitas, F. 2017. Verifying description logic ontologies based on competency questions and unit testing. In *Proceedings of the IX Seminar on Ontology Research in Brazil and I Doctoral and Masters Consortium on Ontologies, Brasília, Brazil, August 28th-30th, 2017*, M. Abel, S. R. Fiorini, and C. Pessanha, Eds. CEUR Workshop Proceedings, vol. 1908. CEUR-WS.org, 159–164.

Bogaerts, B., Erdem, E., Fodor, P., Formisano, A., Ianni, G., Inclezan, D., Vidal, G., Villanueva, A., Vos, M. D. and Yang, F., Eds. 2019. *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20–25, 2019*. EPTCS, vol. 306.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Com. ACM 54,* 12, 92–103.

Buccafurri, F., Leone, N. and Rullo, P. 2000. Enhancing disjunctive datalog by constraints. *TKDE 12,* 5, 845–860.

Busoniu, P., Oetsch, J., Pührer, J., Skocovsky, P. and Tompits, H. 2013. Sealion: An Eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming 13,* 4–5, 657–673.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krenwallner, T., Leone, N., Maratea, M., Ricca, F. and Torsten, S. 2020. Asp-core-2 input language format. *Theory and Practice of Logic Programming 20,* 2, 294–309.

Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys 33,* 3, 374–425.

De Vos, M., Kisa, D. G., Oetsch, J., Pührer, J. and Tompits, H. 2012. Annotating answer-set programs in LANA. *Theory and Practice of Logic Programming 12,* 4–5, 619–637.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Eiter, T., Germano, S., Ianni, G., Kaminski, T., Redl, C., Schüller, P. and Weinzierl, A. 2018. The DLVHEX system. *Künstliche Intelligenz 32,* 2–3, 187–189.

Eiter, T. and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence 15,* 3-4, 289–323.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine 37,* 3, 53–68.

Erdem, E. and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming 3,* 4–5, 499–518.

Erdogmus, H., Morisio, M. and Torchiano, M. 2005. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering 31,* 3, 226–237.

Fandinno, J., Lifschitz, V., Lühne, P. and Schaub, T. 2020. Verifying tight logic programs with anthem and vampire. *Theory and Practice of Logic Programming 20,* 5, 735–750.

Febbraro, O., Leone, N., Reale, K. and Ricca, F. 2011a. Unit testing in ASPIDE. In *Applications of Declarative Programming and Knowledge Management – 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28–30, 2011, Revised Selected Papers*, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 7773. Springer, 345–364.

Febbraro, O., Leone, N., Reale, K. and Ricca, F. 2011b. Unit testing in ASPIDE. *CoRR abs/1108.5434*.

Febbraro, O., Reale, K. and Ricca, F. 2011. ASPIDE: integrated development environment for answer set programming. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16–19, 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 317–330.

Fichte, J. K., Gaggl, S. A. and Rusovac, D. 2022. Rushing and strolling among answer sets - navigation made easy. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 – March 1, 2022*. AAAI Press, 5651–5659.

Fichte, J. K., Hecher, M., Morak, M. and Woltran, S. 2016. Counting answer sets via dynamic programming. *CoRR abs/1612.07601*.

Fraser, S., Beck, K. L., Caputo, B., Mackinnon, T., Newkirk, J. and Poole, C. 2003. Test driven development (TDD). In *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 25–29, 2003 Proceedings*, M. Marchesi and G. Succi, Eds. Lecture Notes in Computer Science, vol. 2675. Springer, 459–462.

Froleyks, N., Heule, M., Iser, M., Järvisalo, M. and Suda, M. 2021. SAT competition 2020. *Artificial Intelligence 301*, 103572.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming 19,* 1, 27–82.

Gebser, M., Maratea, M. and Ricca, F. 2017. The sixth answer set programming competition. *Journal of Artificial Intelligence Research 60*, 41–95.

Gebser, M., Maratea, M. and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory and Practice of Logic Programming 20,* 2, 176–204.

Gelfond, M. and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9,* 3/4, 365–386.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press

Gomes, C. P., Hoffmann, J., Sabharwal, A. and Selman, B. 2007. Short xors for model counting: From theory to practice. In *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28–31, 2007, Proceedings*, J. Marques-Silva and K. A. Sakallah, Eds. Lecture Notes in Computer Science, vol. 4501. Springer, 100–106.

Gomes, C. P., Sabharwal, A. and Selman, B. 2006. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4–7, 2006*, B. Schölkopf, J. C. Platt, and T. Hofmann, Eds. MIT Press, 481–488.

Gressler, A., Oetsch, J. and Tompits, H. 2017. Harvey : A system for random testing in ASP. In *Logic Programming and Nonmonotonic Reasoning – 14th International Conference, LPNMR 2017, Espoo, Finland, July 3–6, 2017, Proceedings*, M. Balduccini and T. Janhunen, Eds. Lecture Notes in Computer Science, vol. 10377. Springer, 229–235.

Hemaspaandra, L. A. and Vollmer, H. 1995. The satanic notations: counting classes beyond #p and other definitional adventures. *SIGACT News 26,* 1, 2–13.

Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J. and Tompits, H. 2010. On testing answer-set programs. In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, H. Coelho, R. Studer, and M. J. Wooldridge, Eds. Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press, 951–956.

Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J. and Tompits, H. 2011. Random vs. structure-based testing of answer-set programs: An experimental comparison. In *Logic Programming and Nonmonotonic Reasoning – 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16–19, 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 242–247.

Janhunen, T., Oikarinen, E., Tompits, H. and Woltran, S. 2009. Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research 35*, 813–857.

Lazaar, N., Gotlieb, A., and Lebbah, Y. 2010. On testing constraint programs. In *Principles and Practice of Constraint Programming – CP 2010 – 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, D. Cohen, Ed. Lecture Notes in Computer Science, vol. 6308. Springer, 330–344.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log. 7,* 3, 499–562.

Lierler, Y., Maratea, M. and Ricca, F. 2016. Systems, engineering environments, and competitions. *AI Magazine 37,* 3, 45–52.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13-18, 1994*, P. V. Hentenryck, Ed. MIT Press, 23–37.

Madeyski, L. 2010. *Test-Driven Development – An Empirical Evaluation of Agile Practice.* Springer.

Marek, V. W. and Truszczynski, M. 1991. Autoepistemic logic. *Journal of ACM 38,* 3, 588–619.

Mellarkod, V. S., Gelfond, M. and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1–4, 251–287.

Oetsch, J., Prischink, M., Pührer, J., Schwengerer, M. and Tompits, H. 2012. On the small-scope hypothesis for testing answer-set programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*, G. Brewka, T. Eiter, and S. A. McIlraith, Eds. AAAI Press.

Papadimitriou, C. H. 2007. *Computational complexity.* Academic Internet Publ.

Ricca, F. 2003. A java wrapper for DLV. In *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 2nd Intl. ASP'03 Workshop, Messina, Italy, September 26-28, 2003*, M. D. Vos and A. Provetti, Eds. CEUR Workshop Proceedings, vol. 78. CEUR-WS.org.

Shen, D. and Lierler, Y. 2018. SMT-based constraint answer set solver EZSMT+ for non-tight programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October – 2 November 2018*, M. Thielscher, F. Toni, and F. Wolter, Eds. AAAI Press, 67–71.

Sommerville, I. 2007. *Software Engineering, 8th Edition.* International Computer Science Series. Addison-Wesley.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing 8,* 3, 410–421.

https://doi.org/10.1017/S1471068424000103 Published online by Cambridge University Press