



# Annotation-based Testing for Answer Set Programming

Master's Thesis

Candidate

**Tobias Berei BSc**

Supervisors: Prof. Francesco Ricca, UNICAL  
FH-Prof. DI Johann Heinzlreiter, FH Hagenberg

June 2019

# Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 21, 2019

Tobias Berei BSc

# Abstract

Answer set programming is a form of declarative programming for solving search problems. Being an outcome of nonmonotonic reasoning in knowledge representation, it is a popular field of research and also used in real-world applications. An answer set program encodes a search problem in form of rules. Afterwards, an answer set solving system performs the search for valid solutions by computing the stable models (answer sets that correspond to problem solutions). As in conventional programming languages, the development process is exposed to errors, which may lead to unexpected behaviour during execution. To detect potential errors and assure a correctly functioning program, (unit) tests can be utilized. In contrast to other programming languages and technologies, the methodologies (and tooling) for testing answer set programs are sparse. Therefore, the objective of this thesis is to propose a new annotation-based language for defining test cases. Furthermore, since answer set programming is declarative, the execution of test cases can be improved with regards to previous execution approaches. Consequently, this thesis combines a new annotation language for testing answer set programs with an enhanced test execution mechanism in a prototypical implementation of a development environment. As a result, the annotation language appears to be as expressive as previously known languages, but more practical in its usage. Due to the fact, that all annotations are enclosed inside language comments, they do not interfere with program executability. Thus, it does not require a separate test definition file, while being able to express similar structures as with previous approaches. Additionally, also the test execution mechanism is designed as a lightweight, but reliable process. Instead of executing the program (respectively rules) under test and checking the output, it utilizes program rewriting in order to validate the conditions and assertions of the test cases. The evaluation showed, that this procedure is faster than previous approaches. By integrating both, the annotation language and the execution mechanism, in a development environment with basic functionalities, this thesis presents the contents theoretically and demonstrates the feasibility of implementing the same contents in the light of the development environment ASP-WIDE.

# Kurzfassung

Answer Set Programming ist eine Form der deklarativen Programmierung zur Lösung von Suchproblemen. Als Nebenerscheinung des nichtmonotonen Schließens (nonmonotonic reasoning) in der Wissensrepräsentation (knowledge representation), ist es ein beliebtes Forschungsfeld und findet auch in realen Applikationen Anwendung. Ein Answer-Set-Programm beschreibt ein Suchproblem in Form von Regeln. Anschließend reduziert ein System zur Lösung von Answer-Set-Programmen das Suchproblem auf valide Lösungen (im Englischen “stable models” oder “answer sets” genannt). Der Entwicklungsprozess ist hierbei, wie auch in konventionellen Programmiersprachen, der Einführung von Fehlern ausgesetzt, welche zu unvorhergesehenem Fehlverhalten führen können. Um mögliche Fehler zu erkennen und ein korrekt funktionierendes Programm zu gewährleisten, können (Unit-) Tests herangezogen werden. Im Gegensatz zu anderen Programmiersprachen und Technologien, existieren nur wenige Methoden (und Werkzeuge) zum Testen von Answer-Set-Programmen. Daher ist das Ziel dieser Arbeit, eine neue Annotations-basierte Sprache zum Testen von Answer-Set-Programmen zu entwerfen. Da Answer Set Programming einem deklarativem Paradigma folgt, kann die Ausführung im Vergleich zu bereits bestehenden Ansätzen verbessert werden. Folglich vereint diese Arbeit eine neue Annotations-basierte Sprache zum Testen von Answer-Set-Programmen mit einem verbesserten Ausführungsmechanismus in einer prototypischen Implementierung einer Entwicklungsumgebung. Die resultierende Sprache ist dabei ähnlich ausdrucksstark wie bestehende Notationsformen, jedoch zweckmäßiger in der Anwendung. Aufgrund der Tatsache, dass alle Annotationen in Sprachkommentare eingebettet sind, beeinflussen sie die Ausführbarkeit des Programms nicht. Während mit der Annotationssprache ähnliche Konstrukte ausgedrückt werden können wie mit anderen Ansätzen, erfordert sie keine separate Test-Datei. Zusätzlich ist auch die Test-Ausführung als leichtgewichtiger Prozess entworfen. Anstatt das zu testende Programm (bzw. die Regeln) auszuführen und das Resultat zu prüfen, wird das eigentliche Programm modifiziert ausgeführt, um die Testresultate festzustellen. Die Evaluation zeigt, dass dieser Ansatz schneller ist als zuvor bekannte Ausführungsmechanismen. Da sowohl die Annotations-basierte Sprache als auch der Ausführungsmechanismus in der Entwicklungsumgebung ASP-WIDE integriert ist, präsentiert diese Arbeit nicht nur theoretische Inhalte, sondern zeigt auch deren Machbarkeit in Form eines Entwicklungswerkzeugs.

# Contents

<b>Declaration</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Kurzfassung</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem presentation . . . . .	1
1.2 Thesis objective . . . . .	2
<b>2 Preliminaries</b>	<b>3</b>
2.1 Answer Set Programming . . . . .	3
2.1.1 The language of ASP . . . . .	3
2.1.2 Programming methodology . . . . .	5
2.2 Answer Set Programming Systems . . . . .	6
2.2.1 Grounding and Solving . . . . .	6
2.2.2 Clingo . . . . .	7
2.2.3 DLV2 . . . . .	7
2.3 Integrated Development Environments for ASP . . . . .	7
2.3.1 ASPIDE . . . . .	8
2.3.2 SeaLion . . . . .	8
<b>3 Related Work</b>	<b>10</b>
3.1 Theoretical work . . . . .	10
3.2 Practical work . . . . .	12
3.2.1 Unit Testing in ASPIDE . . . . .	12
3.2.2 Language for Annotating Answer Set Programs (LANA) . . . . .	15
<b>4 Annotation-based Testing for ASP</b>	<b>20</b>
4.1 Valuation of previous work . . . . .	20
4.2 Solution approach . . . . .	21
4.3 Annotation Language . . . . .	21
4.3.1 Base Annotations . . . . .	22
4.3.2 Assertion Annotations . . . . .	24
4.3.3 Usage . . . . .	24
4.4 Execution mechanism . . . . .	27
4.4.1 Testing engine . . . . .	27
4.4.2 Test execution . . . . .	28

<b>5</b>	<b>Implementation</b>	<b>34</b>
5.1	Editors for Logic Programming . . . . .	34
5.2	Architecture and Design . . . . .	35
5.3	Technologies . . . . .	35
5.4	Backend . . . . .	37
5.4.1	Annotation language parser . . . . .	37
5.4.2	DLVWrapper usage . . . . .	39
5.4.3	Testing engine realization . . . . .	40
5.5	Frontend . . . . .	42
5.5.1	Code editor . . . . .	42
5.5.2	Syntax highlighting . . . . .	43
5.5.3	Executing and testing answer set programs . . . . .	44
5.6	The ASP-WIDE user interface . . . . .	46
<b>6</b>	<b>Use Case and Evaluation</b>	<b>48</b>
6.1	Use Case . . . . .	48
6.1.1	Hamiltonian path problem instance . . . . .	48
6.1.2	Defining test cases . . . . .	50
6.1.3	Executing test cases . . . . .	52
6.2	Evaluation . . . . .	53
6.2.1	Language features . . . . .	53
6.2.2	Test execution mechanism . . . . .	54
<b>7</b>	<b>Conclusion and prospects</b>	<b>58</b>
<b>A</b>	<b>Annotation language</b>	<b>60</b>
A.1	EBNF grammar specification . . . . .	60
A.2	Lexical tokens . . . . .	62
	<b>References</b>	<b>63</b>
	Literature . . . . .	63
	Online sources . . . . .	65

# Chapter 1

## Introduction

Answer set programming (ASP) is a form of logic programming for knowledge representation and reasoning. In particular it can be used for solving complex search problems. Being a relatively new form of programming (first applications are reported in 1998), the methodology has gone through an extensive development process [Lif08]. Not only various ASP systems for executing answer set programs have been developed, but also multiple IDEs and techniques for debugging/testing have been proposed. Both theoretical, as well as practical contributions have been made over the past 20 years, making ASP a fast developing field of research and also an appealing technology for real-world applications.

Foremost ASP remains a programming paradigm that requires a development and testing phase, as it is the case in most conventional programming languages. For speeding-up the development of error resistant programs, modern software development processes make use of unit tests, often as part of a Test-Driven Development approach [Fra+03]. This approach puts the formulation of tests to the beginning of all software developments. Following implementations have to comply (pass) the given tests in order to achieve an improvement [Bec02]. Consequently, the possibility to execute tests allows to find implementation errors or problems early in the development process. Furthermore, the formulation of unit tests helps to isolate the incorrect behaviour more easily. As Erdogmus, Morisio, and Torchiano [EMT05] conclude, writing tests before code increases the productivity of developers (based on an experiment with undergraduate students). Nevertheless, no matter if tests are formulated as part of a Test-Driven Development approach or for validating the functioning of already written code, it contributes to building correct and robust programs that are free of unexpected behaviour. This is also applicable for answer set programming, which is why defining test cases for ASP is reasonable.

### 1.1 Problem presentation

Since testing in conventional programming languages can be used for detecting errors and increasing productivity, it can also be a useful approach for ASP. Indeed, several contributions in the context of (unit) testing have been proposed. While some follow a purely theoretical intention, others propose ready to use tooling for realizing unit tests.

With regards to theoretical contributions, sound formal definitions of unit testing concepts exist. Moreover, further papers address the usefulness of tests, as well as code coverage metrics in ASP. In contrast to practically oriented contributions, they are of minor value for ASP developers, as they often lack an implementation. On the other hand, only few prac-

tical contributions in the context of testing for ASP exist. While some utilize a complex test definition language, others produce unnecessary overhead and are therefore suboptimal solutions in terms of practicality. Moreover, the possibilities to express certain conditions or assertions inside a test case differ strongly. Also the test execution mechanisms use a very basic approach, that can be improved to achieve faster execution times. All in all the field of testing in ASP is limited, inefficient and often complex to use. Thus, an easy to use and lightweight solution for writing tests including an efficient execution mechanism is missing.

## 1.2 Thesis objective

The aim of this thesis is to provide a new language for testing answer set programs. As annotations are known to be an efficient way of adding metadata to a program, the goal is to define a lightweight annotation-based language. Foremost the language must be simple to use and to understand in order to achieve a high acceptance among ASP developers. Additionally, it should provide as much expressiveness as required for specifying extensive test cases. Besides an annotation language for testing, this thesis also targets the development of an efficient execution mechanism. This mechanism should be at least more performant than executing the program under test and checking the output. Since theoretical contributions are impractical for most ASP developers, both, the annotation language and the execution mechanism, should be integrated into a ready to use tool/environment for defining and executing test cases for ASP.



# Chapter 2

## Preliminaries

In order to fully understand the contents of this thesis, a general understanding of specific preliminary topics is required. This Chapter aims to give a basic introduction to the topics of answer set programming, answer set programming systems and integrated development environments for answer set programming. The following sections do not give a complete introduction to the stated topics, but they can be considered as a primer in order to follow the contents of this work.

### 2.1 Answer Set Programming

Answer set programming (ASP) is a declarative programming paradigm oriented towards search problems. As an outgrowth of non-monotonic reasoning in knowledge representation, it has its roots in logic programming. An answer set program consists of Prolog-style rules which can be used to encode a search problem (and corresponding knowledge). Based on this problem encoding, an answer set solver performs the search for valid answer sets (also referred to as stable models) [Lif08].

An early example of answer set programming has been proposed in 1998 by Soinen and Niemelä [SN98], therefore the ASP methodology is about 20 years old. Since then it evolved and has been used in research and several commercial use cases [BET11; LR15]. With the development of different ASP systems, the features, respectively the syntax of the programming language diverged. In order to unify different notations, an agreement on the core answer set programming language has been made, which is called “ASP-Core-2 Input Language Format” and is considered in this thesis<sup>1</sup>.

The following subsections give an introduction to selected concepts of the syntax, semantics and the programming methodology in answer set programs. For an extensive introduction to answer set programming, take a look at Lifschitz [Lif08], Leone and Ricca [LR15], Bonatti et al. [Bon+10] or consider Baral [Bar03] for a more comprehensive introduction.

#### 2.1.1 The language of ASP

In answer set programs identifiers starting with uppercase letters denote logical variables, whereas identifiers starting with lowercase letters indicate the use of constants. The expression  $p(t_1, \dots, t_n)$  defines an atom, where  $p$  is a predicate of arity  $n$  and  $t_1, \dots, t_n$  are terms. A term

---

<sup>1</sup>Check <https://www.mat.unical.it/aspcomp2013/ASPStandardization> for further information on ASP standardization (accessed 2019-05-02).

can either be a variable or a constant. Furthermore, an atom can be negated by prepending *not*, which is then referred to as a negative literal. Consequently, an atom which is not negated is a positive literal [Bon+10].

An answer set program consists of a set of rules, which are expressions of the form *head* :- *body*, where the body is a conjunction of positive or negative literals and the head is a disjunction of atoms. Following Alviano et al. [Alv+11] these answer set rules can be read as: “If *body* is true, then *head* is true”. Besides a default answer set rule with head and body, either of them can be omitted to obtain two special kinds of rules. A rule without a body models an unconditional truth and is therefore called fact. In this case the connecting sign :- can be skipped for simplicity. On the contrary a rule without a head is called integrity constraint or strong constraint and models a condition that must be false in any possible answer set [Alv+11].

Considering these definitions a rule can be defined as

$$a_1 \mid \dots \mid a_n \text{ :- } b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m.$$

where  $a_1, \dots, a_n, b_1, \dots, b_m$  are literals with  $n \geq 0$  and  $m \geq k \geq 0$  as shown by Bonatti et al. [Bon+10].

Rule safety is a requirement for an answer set program, which is achieved if all rules of the program are safe. A rule is safe if each variable of the rule also appears at least once in a positive literal of the body. Moreover, a term/atom/rule/program is called “ground”, if it contains no variable [LR15].

**Listing 2.1:** 3-colorability problem.

```

1 % model the input (knowledge)
2 node(1). node(2). node(3).
3 edge(1, 2). edge(1, 3). edge(2, 3).
4
5 % generate all potential solutions
6 col(X, red) | col(X, blue) | col(X, green) :- node(X).
7
8 % eliminate invalid solutions
9 :- edge(X, Y), col(X,C), col(Y,C).
```

Listing 2.1 shows a simple example of an answer set program. This program encodes an instance of the 3-colorability problem, which aims to find an assignment of three colors (e.g. red, blue and green) to the nodes of a graph such that connected nodes always have different colors. This instance models three nodes (1, 2, 3) of which each is connected with the other nodes by edges. Lines 2 and 3 encode the input for the search problem by using facts. This input can be seen as given knowledge. The rule in Line 6 generates all potential solutions by generating all possible color assignments for each node. Notice that the variable  $X$  in the atom *node* can take all defined constant values of this atom (1, 2 and 3). Additionally this rule is safe as the variable  $X$  appears in at least one positive literal of the body, namely *node*. As not all color assignments are valid, the integrity constraint in Line 9 eliminates all invalid solutions by prohibiting that two connected nodes can have the same color. When executing this simple answer set program with the ASP system DLV2, it will output all possible answer sets (see Listing 2.2). For this example there are six answer sets, each fulfilling the integrity constraint. Notice that Listing 2.2 contains all answer sets in an abbreviated form (input facts are truncated).

**Listing 2.2:** Output of DLV2 for the 3-colorability problem instance.

```

{ node(1), ... , edge(2,3), col(1,blue), col(3,green), col(2,red) }
{ node(1), ... , edge(2,3), col(1,blue), col(2,green), col(3,red) }
{ node(1), ... , edge(2,3), col(1,green), col(3,red), col(2,blue) }
{ node(1), ... , edge(2,3), col(3,green), col(1,red), col(2,blue) }
{ node(1), ... , edge(2,3), col(3,blue), col(1,red), col(2,green) }
{ node(1), ... , edge(2,3), col(2,red), col(3,blue), col(1,green) }

```

Answer set programs can also encode optimization constructs by the use of weak constraints. A weak constraint is an expression of the form:

$$:\sim b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m. [w@l]$$

where  $w$  and  $l$  are the weight and level of the constraint. The weight can be considered as the cost of violating the condition in the body of the constraint, while the levels can be used to define a priority among constraints [LR15].

Considering Listing 2.1, the program can be extended to model an optimization problem, where node 1 should preferably be red. This can be achieved by appending the Lines 2-4 of Listing 2.3 to the original program.

**Listing 2.3:** Weak constraints for 3-colorability problem.

```

1 % node 1 should preferably be red
2 :- col(1, red). [1@1]
3 :- col(1, green). [2@1]
4 :- col(1, blue). [2@1]

```

These constraints model the demand for node 1 to be red by having a lower cost on the corresponding weak constraint. As the color of the other nodes is not important, they can have any (but higher) cost than color red. Furthermore, all constraints are defined on the same level, as there is no priority among them. By evaluating the extended program the answer set solver will produce answer sets with color red for node 1 for the optimized solution as shown in Listing 2.4. Notice that the answer sets in Line 5 and 6 are optimal, which is indicated by the keyword OPTIMUM in Line 7. Consequently, two optimal answer sets exist for this example.

**Listing 2.4:** Output of DLV2 for the extended 3-colorability problem instance.

```

{ ... , col(2,red), col(1,blue), col(3,green) }
COST 2@1

{ ... , col(1,red), col(3,green), col(2,blue) }
COST 1@1

{ ... , col(1,red), col(3,green), col(2,blue) }
{ ... , col(1,red), col(3,blue), col(2,green) }
OPTIMUM

```

While this Subsection gives an introduction to selected language elements required for this thesis, Calimeri et al. [Cal+14] give a full description of the core answer set programming language features, which contains notations for queries, choice rules, aggregate functions, etc.

### 2.1.2 Programming methodology

Answer set programs very often follow the “generate and test” strategy, which means that in a first step a superset of the set of solutions is described (all “potential solutions”). Afterwards

the integrity constraints eliminate all invalid “potential solutions” in order to obtain the valid answer sets for the search problem [Lif08].

Listing 2.1 shows the “generate and test” organization in a simple example of an answer set program. As already stated, the rule in Line 6 generates all potential solutions for this search problem. In a final step all solution candidates are tested for a valid color assignment, which is achieved by the integrity constraint in Line 9.

When encoding more complex search problems, the program may contain the definition of several subsequent auxiliary predicates/atoms which are used in the constraints. According to Lifschitz [Lif08], these rules constitute a third aspect of the programming methodology in ASP, namely the “define” part. Therefore the “generate and test” organization can be considered to be a “generate, define, test” organization for more complex answer set programs.

## 2.2 Answer Set Programming Systems

Answer set programming systems are efficient programs, that take answer set programs as input and produce valid solutions (answer sets or stable models) for the encoded problem. ASP systems as Clingo<sup>2</sup>, DLV<sup>3</sup> and DLV2<sup>4</sup> usually incorporate two components for evaluating answer sets, namely the grounder and the solver, which will be explained later. With regards to answer set solvers, Clasp and WASP are commonly known in the research community due to their continuous contribution to answer set programming competitions [Alv+13; Cal+16; GMR15; GMR17]. Furthermore, they are integrated into the ASP systems Clingo, respectively DLV2, which provide a user-friendly command line interface for running answer set programs. Consequently, Clingo and DLV2 have been chosen to be practically relevant for this thesis. Therefore the following subsections give an overview of the ASP solving process and the inner components of Clingo and DLV2.

### 2.2.1 Grounding and Solving

The usual problem-solving work flow in ASP involves modeling, grounding and solving. While during the modeling phase the problem is expressed in a logic program, an answer set programming system can perform the latter, namely grounding and solving. According to Kaufmann et al. [Kau+16] a grounder systematically replaces all variables in the program by variable-free terms and the solver takes the resulting propositional program and computes its answer sets.

The grounders of Clingo and DLV2 perform their computation by generating a ground program that does not contain any variable while having the same answer sets as the original program. This first step is usually called “grounding” or “instantiation” while the result is the “ground instantiation” of the original program. As grounding may be computationally expensive, it has a big impact on the performance of the whole system. This is a consequence of the fact, that the output of the grounder is the input for the utilized ASP solver. Consequently, a naive approach for grounding, for instance replacing the variables with all constants appearing in the program, is undesirable from a computational point of view [Kau+16].

Kaufmann et al. [Kau+16] as well as Lifschitz [Lif08] state, that the inner functioning of the solving process has its roots in satisfiability testing (SAT). Satisfiability testing is a

<sup>2</sup>Visit <https://potassco.org/clingo/> for information on Clingo (accessed 2019-05-02).

<sup>3</sup>Visit <http://www.dlvsystem.com/> for information on DLV (accessed 2019-05-02).

<sup>4</sup>Visit <https://www.mat.unical.it/DLV2/> for information on DLV2 (accessed 2019-05-02).

process, that given an input formula in propositional logic, either produces a satisfying assignment for the formula or proves that the formula is unsatisfiable [Gom+08]. Since the grounding process produces a propositional program as output, a solver implementation inspired by satisfiability testing can efficiently evaluate the valid models. Nowadays, solvers are based on a variation of the DPLL procedure (Davis-Putnam-Logemann-Loveland), which is CDCL (Conflict-Driven Clause Learning) [GKS12].

### 2.2.2 Clingo

Clingo is an answer set programming system developed by the University of Potsdam in Germany<sup>5</sup>. This ASP system combines the grounder Gringo with the solver Clasp in order to form a solving system. The system is currently at version 5, while the adoption of the ASP-Core-2 input language format came with version 4 [Geb+14]. Due to several contributions to Answer Set Programming Competitions the solver Clasp is well-known among researchers and the ASP community.

### 2.2.3 DLV2

DLV2 is developed at the University of Calabria and updates the ASP system DLV<sup>6</sup>. It combines the grounder I-DLV with the answer set solver WASP. In contrast to DLV, the DLV2 implementation is fully-compliant with the ASP-Core-2 input language format. According to Alviano et al. [Alv+17] DLV2 turns out to be “dramatically more efficient than DLV and comparable with Clingo” in terms of system performance. This statement was made after comparing the systems with benchmarks from answer set programming competitions. In addition to this conclusion the answer set solver WASP also contributed to various ASP competitions and is therefore widely known among researchers in the field of non-monotonic reasoning.

## 2.3 Integrated Development Environments for ASP

With the wide usage spectrum of logic-based paradigms in research projects as well as industrial applications, the need for a user-friendly development environment arose. The combination of a code editor with a convenient execution environment is one key feature of an integrated development environment (IDE) for logic-based programming, as demonstrated by various implementations [GCP17]. Some of these are IDP Web-IDE<sup>7</sup>, LoIDE<sup>8</sup>, SeaLion<sup>9</sup> and ASPIDE<sup>10</sup>. While IDP Web-IDE and LoIDE are web-based development environments, SeaLion and ASPIDE do not rely on web technologies. SeaLion is designed as a plugin for the Eclipse IDE and ASPIDE is a standalone tool with a comprehensive set of features. Despite the strong variation of features, all IDEs support core functionalities such as a code editor with syntax checking and an execution environment. This thesis will consider ASPIDE and SeaLion as IDEs of practical relevance. Furthermore, it will mostly refer to ASPIDE, as a comprehensive IDE for answer set programming. The following subsections will describe ASPIDE

---

<sup>5</sup>Visit <https://potassco.org/clingo/> for more information on Clingo (accessed 2019-05-02).

<sup>6</sup>Visit <https://www.mat.unical.it/DLV2/> for more information on DLV2 (accessed 2019-05-02).

<sup>7</sup>Visit <http://adams.cs.kuleuven.be/idp> for more information on IDP Web-IDE (accessed 2019-05-02).

<sup>8</sup>Visit <https://github.com/demacs-unical/loide> for more information on LoIDE (accessed 2019-05-02).

<sup>9</sup>Visit <http://www.sealion.at/> for more information on SeaLion (accessed 2019-05-02).

<sup>10</sup>Visit <https://www.mat.unical.it/ricca/aspide/> for more information on ASPIDE (accessed 2019-05-02).

and SeaLion more detailed and especially focus on testing capabilities of these development environments.

### 2.3.1 ASPIDE

ASPIDE is a comprehensive IDE for ASP, which integrates an editing tool with syntax highlighting, syntax correction, auto-completion, code templates, quick fixes and refactoring support. Moreover, it provides graphical tools for program composition, debugging, profiling, database access, solver execution configuration and output handling. The integrated development environment was built using the ASP system DLV for executing programs. While Febbraro, Reale, and Ricca [FRR11] describe the comprehensive features of the IDE and provide a feature-wise comparison with other development environments, Febbraro et al. [Feb+11] also present the capabilities of ASPIDE with regards to unit testing.

ASPIDE is an example of an IDE featuring testing support for ASP. In particular it incorporates a new testing language for specifying and running unit tests, inspired by the JUnit framework (Java) [Feb+11]. Figure 2.1 shows the start screen of ASPIDE. Further discussion of the unit testing capabilities of ASPIDE is contained in Chapter 3.

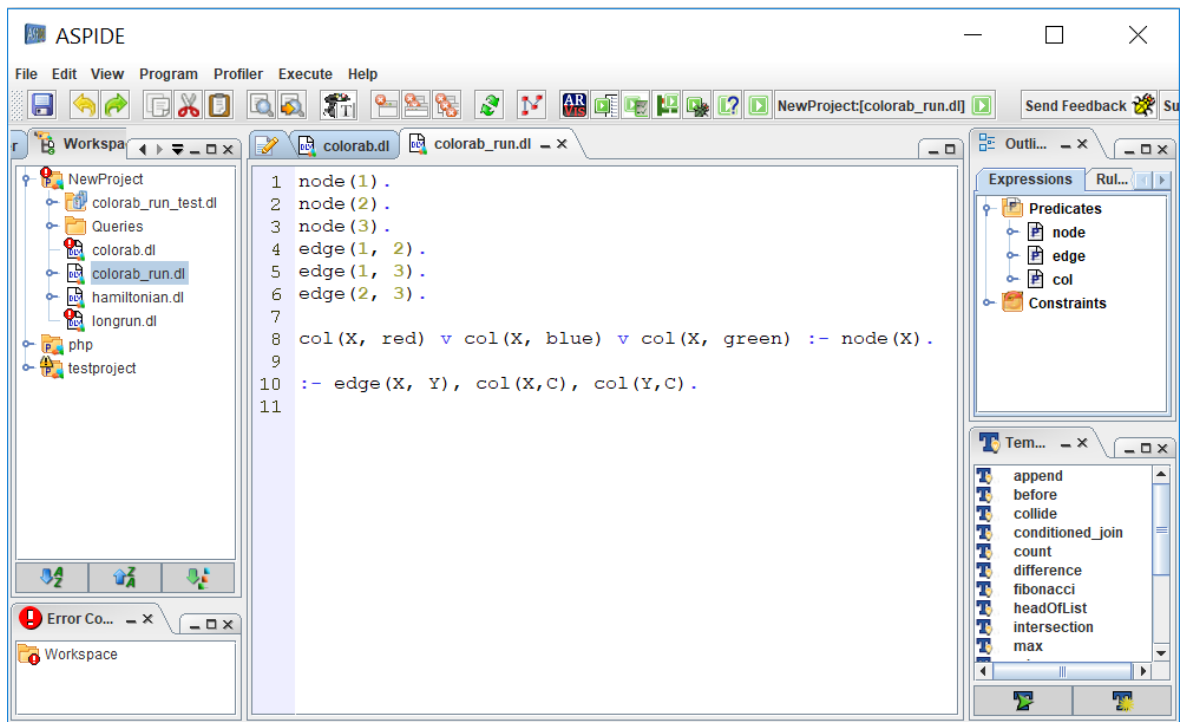


Figure 2.1: Screenshot of the integrated development environment ASPIDE.

### 2.3.2 SeaLion

The integrated development environment SeaLion was originally developed as part of an ongoing research project on the methods and methodologies for developing answer set programs. It is designed as a plugin for the IDE Eclipse which is widely known, especially as a development environment for Java. The goal was not to only target an audience of ASP

experts, but also software developers new to ASP. SeaLion supports the languages of DLV and Gringo (of the ASP system Clasp) to a large extent. Based on that, the editor provides syntax highlighting, syntax checking, error reporting, error highlighting and automatic program outlines. Furthermore, the IDE supports run/solve configurations for specifying command line arguments and multiple input files. Additionally, a key feature of SeaLion is the visualization and the visual editing of interpretations, which allows to graphically manipulate the interpretations under consideration for the program [OPT11]. Figure 2.2 shows a screenshot of the Eclipse plugin.

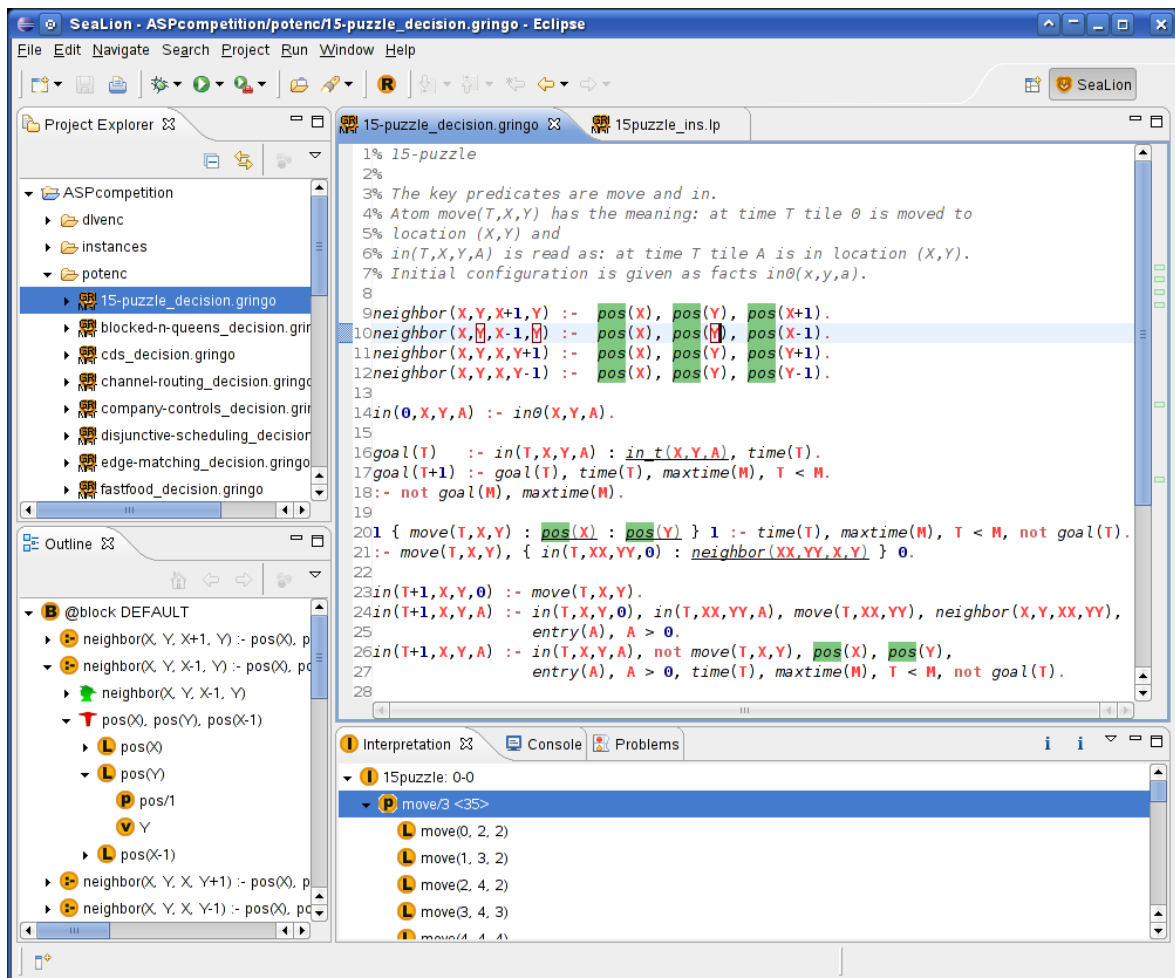


Figure 2.2: Screenshot of the Eclipse plugin SeaLion. (Image source: [OPT11])

Another feature of SeaLion is the possibility to generate documentation for answer set programs using an integrated tool called ASPDoc. This documentation generator is based on LANA (Language for ANnotating Answer set programs), which was proposed by De Vos et al. [De +12]. Alongside with LANA and ASPDoc another tool called ASPUnit was presented, which implements an annotation-based unit testing mechanism. While ASPDoc is fully integrated in SeaLion, the integration of ASPUnit is intended according to Oetsch, Pührer, and Tompits [OPT11]. Due to their practical relevance, LANA and ASPUnit will be discussed in more detail in Chapter 3.

## Chapter 3

# Related Work

When dealing with annotation-based testing for answer set programming previous work has to be considered as the foundation of testing in ASP. Compared to conventional programming languages like Java and C# comparably little effort has been done with regards to testing. In spite of ASP being a relatively new field of programming, there is theoretical, as well as practical work in the field of testing that has to be noticed. The following sections give an overview of related work in the context of testing in answer set programming. Initially, theoretical papers that lay the foundation of any testing in ASP are outlined. Secondly, practical work in form of implementations show the feasibility of realizing testing mechanisms in ASP.

### 3.1 Theoretical work

Due to the development of efficient solver technology, ASP has become an important host language for solving reasoning problems in the field of artificial intelligence like planning, diagnosis or information integration. Nonetheless, the evolution of the methods to support the development of answer-set programs, for example unit testing, have not been targeted sufficiently. Janhunen et al. [Jan+10] lay down the foundations for testing answer set programs by defining/formalizing terms known from classical unit testing, for example test case and test suite, in the context of ASP. Based on these formalizations, some key aspects are adopted and presented hereinafter, due to their relevance for this thesis.

According to Myers and Sandler [MS04] a test case for a program  $P$  consists of a description of the correct output of  $P$ , given some input of  $P$ . In ASP several outputs, as well as no output, are possible for an input to a program. This is a consequence of the fact, that answer set programs are non-deterministic. Hence, the definition of a decision mechanism, which can be seen as a test oracle, is required to determine the correct outputs for any input. For this case Janhunen et al. [Jan+10] define a so called “specification” for a program  $P$ , which is a mapping from the sets of possible inputs of  $P$  to collections of sets of outputs (of  $P$ ). Following this, the correct outputs (if any) for  $P$  are determined by applying the specification (mapping) on the input of  $P$ .

Based on these findings, given a program  $P$  and a specification  $\sigma$  for  $P$ , a test case  $T$  for  $P$  is a pair  $\langle inp(T), out(T) \rangle$  where  $inp(T)$  is an actual input for  $P$  and  $out(T)$  is the output of  $P$  (based on  $inp(T)$ ) that fulfills  $out(T) = \sigma(inp(T))$ . Subsequently,  $P$  passes  $T$  if  $P[inp(T)] = out(T)$ , respectively  $P$  fails  $T$  otherwise. Furthermore, a test suite  $S$  for a program  $P$  and a specification  $\sigma$  for  $P$  is a collection of test cases for  $P$  and  $\sigma$  [Jan+10].



**Listing 3.1:** Hamiltonian path problem.

```

1 % facts
2 vtx(1). vtx(2). vtx(3). vtx(4). vtx(5).
3 edge(1, 2). edge(2, 3). edge(3, 4). edge(4, 5).
4 edge(3, 2). edge(1, 3). edge(2, 4). edge(5, 1).
5 start(1).
6
7 % each edge either belongs to hamiltonian-path or not (generation of possible solutions)
8 inPath(X, Y) v outPath(X, Y) :- edge(X, Y).
9
10 % the starting vertex is reached for sure
11 reached(X) :- start(X).
12
13 % if we reached Y and edge(Y, X) is in the path, we also reached X (traversing)
14 reached(X) :- inPath(Y, X), reached(Y).
15
16 % there must be no vertex that is not reached
17 :- vtx(X), not reached(X).
18 % the start node cannot be the target vertex of an edge
19 :- start(X), inPath(_, X).
20 % each vertex in the path must have at most one incoming and one outgoing edge
21 :- vtx(X), #count{Y : inPath(X, Y)} > 1.
22 :- vtx(X), #count{Y : inPath(Y, X)} > 1.

```

Listing 3.1 shows an instance of the hamiltonian path problem, which is NP-complete and tries to find a solution for the following search problem: Given a finite directed graph  $G = (V, E)$  and a vertex  $a$ , is there a path in  $G$  starting from  $a$  and passing through each vertex in  $V$  exactly once? The facts in the lines 2-5 specify the input for the current instance, while the rules contained in the lines 8-22 define the problem solving itself. First all possible solutions are generated (Line 8). Afterwards, the generated solutions are traversed based on the defined starting vertex (Lines 11 and 14). The integrity constraints in the lines 17, 19 and 21-22 assure that all vertices are reached, the start node is not targeted by any edge and each vertex has at most one incoming and one outgoing edge.

An easy way for humans to calculate the answer sets of this problem, is to draw the graph and manually check if there are hamiltonian paths (for simplicity reasons, this method is assumed to be always complete for this example). For this instance two answer sets, namely the paths  $(1, 2, 3, 4, 5)$  and  $(1, 3, 2, 4, 5)$ , can be found. As this is the complete solution, the aim is to test whether the program constructs the same answer sets. Therefore we define a specification  $\sigma$  that maps the input of the lines 2-5 to the two hamiltonian paths evaluated by the complete method. Now a test case can be defined, which contains the specification and the program under test. Table 3.1 shows all required elements for the definition of a test case according to Janhunen et al. [Jan+10] based on the current hamiltonian path problem instance.

Since  $\sigma$  is defined to map the input of the test case to the valid output, the program  $P$  passes  $T$  if

$$P[inp(T)] = \sigma(inp(T)) = out(T) ,$$

respectively fails  $T$  otherwise. While Janhunen et al. [Jan+10] also discuss different test coverage notions, analyzes their computational complexity and lays down basic techniques for test automation in ASP, the formalizations presented above are of major importance for this thesis. They are seen as the foundations of testing in answer set programming.

In addition to the definition of the foundations, there are further related works in the

Test case $T$
<pre> inp(T) := 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).          start(1).  out(T) := {(1, 2, 3, 4, 5), (1, 3, 2, 4, 5)}  P := reached(X) :- start(X).      reached(X) :- inPath(Y, X), reached(Y).      :- vtx(X), not reached(X).      :- start(X), inPath(_, X).      :- vtx(X), #count{Y : inPath(X, Y)} &gt; 1.      :- vtx(X), #count{Y : inPath(Y, X)} &gt; 1.  σ(inp(T)) := out(T) </pre>

**Table 3.1:** Test case definition for the hamiltonian path problem instance.

context of testing in ASP. Nevertheless, they are of minor importance for this thesis. Oetsch et al. [Oet+12] deal with the so called small-scope hypothesis when testing answer set programs. It evaluates to which extent this hypothesis applies to ASP. The small-scope hypothesis states that a high proportion of errors can be found by testing a program for all test inputs within some small scope [Oet+12]. Furthermore, Janhunen et al. [Jan+11] compare random- vs. structure-based testing of answer set programs. The results indicate that the random-based approach is ineffective, while structure-based techniques catch faults with a high rate more consistently [Jan+11].

## 3.2 Practical work

When considering practical work related to this thesis, little previous effort has been made to provide a testing tool or framework for ASP. Mainly two related works exist, which are of major importance, namely De Vos et al. [De +12] and Febbraro et al. [Feb+11]. Both deal with unit testing in answer set programming and define a formalism as well as a ready-to-use implementation for executing test cases. While De Vos et al. [De +12] follow a purely annotation-based approach for defining tests, Febbraro et al. [Feb+11] make use of annotations partially. Instead tests are defined in a test specification language. The following subsections give a more detailed introduction to both approaches and provide examples for understanding how unit tests can be defined and executed.

### 3.2.1 Unit Testing in ASPIDE

As already stated in Chapter 2, ASPIDE is a comprehensive development environment for ASP. While incorporating an extensive set of features [FRR11], one major aspect with relevance for this thesis is its support for unit tests. In particular Febbraro et al. [Feb+11] propose a new language for specifying and running unit tests on answer set programs. The intention is to define a test suite containing several test cases in a file that is based on a

specific testing language. The development environment is able to read these specifications, execute the test suite and present the outcomes of the tests in the user interface. Assertions made in the test specification are verified by calling an ASP solver and checking its output [Feb+11]. In particular, ASPIDE allows three test execution modes:

- Execution of selected rules
- Execution of a split program
- Execution of the whole program

While the execution of selected rules or the whole program seem natural, the execution of a split program relates to testing the splitting set containing the atoms of the selected rules. According to Lifschitz and Turner [LT94], a logic program is in many cases dividable into two parts, so that one of them, the “bottom” part, does not refer to the predicates defined in the “top” part. Consequently, the “bottom” rules can then be used for the evaluation of the predicates that they define and the computed values can be exploited to simplify the “top” definitions. Hence, testing splitting sets corresponds to testing the interface between certain parts/aspects of the program.

In addition, ASPIDE supports rule naming, which can be useful for referencing specific rules in the test case definition. Rule naming can be utilized by adding the following annotation right above the rule:

```
% @name=myrule
```

As this annotation is written inside a single line comment, it does not interfere with program executability. Nevertheless, ASPIDE is able to parse the annotation and reference the annotated rule by its name.

To use unit testing, a test file has to be defined, which might contain a single test or a test suite including multiple test cases. Each test case can include one or more assertions on the execution result. Also invocation settings can be configured in a way, such that the whole test suite is self-contained in a single test file. For configuring the settings and defining tests, a test language is used, whose syntax is presented in Listing 3.2.

**Listing 3.2:** Testing language of ASPIDE (Adapted from Febbraro et al. [Feb+11]).

```

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"); ] | [ excludeInputFile("file"); ] ]*
9         [
10            [ filter | pfilter | nfilter ]
11            [
12                [ (predicateName [ ,predicateName ]* ) ]
13                | [SELECTED RULES]
14            ] ;
15        ]?
16        [ selectRule(ruleName); ]*
17        [
18            [
19                assertName( [ intnumber, ]? [ [ "atoms" ] | [ "constraint" ] ] ); ]
20                | [ assertBestModelCost(intcost [ , intlevel ]? ); ]

```

```

21         ]*
22     ]
23 }
24 ]*
25 [
26     [ assertName( [ intnumber, ]? [ [ "atoms" ] | [ "constraint" ] ]; ] ]
27     | [ assertBestModelCost(intcost [, intlevel ]? ); ]
28 ]*

```

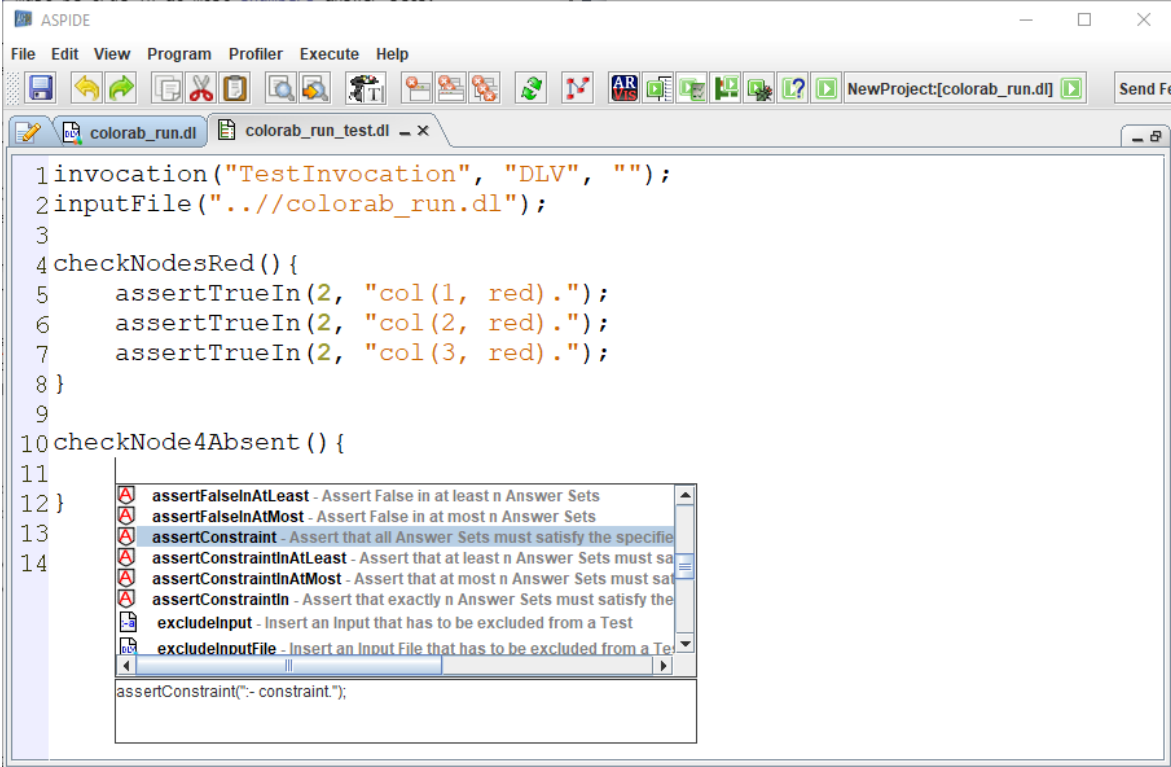
Strings written in bold face represent non-terminal symbols, while token specifications are omitted in this representation for simplicity reasons. Some global invocation settings, that apply to all tests specified in the same file, can be set as seen in the first line. Afterwards, the tester can specify one or more global inputs by writing some *input* and *inputFile* statements. While *input* allows the definition of rules or facts inside a string, *inputFile* relates to a file containing ASP inputs. Line 4 demonstrates how to define a test case with a specific name and stating whether it should be executed on selected rules, a splitting set or the whole program. The tester can also specify additional solver options (line 6) and inputs (line 7) for the test case. Furthermore, global inputs given from the test suite can be excluded with *excludeInput* or *excludeInputFile* as defined in line 8. With the optional statements *filter*, *pfilter*, *nfilter* in line 10, certain output predicates from the test results can be filtered. The statement *selectRule* in line 16 allows to select the rules under test by utilizing rule naming as explained before. Subsequently, the expected output of a test case is expressed with assertion statements. These assertions can be local to the current test (lines 19-20) or global (lines 26-27). Febbraro et al. [Feb+11] define several assertion types to be used:

- **assertTrue**("atomList")/**assertFalse**("atomList")  
Asserts that all atoms of *atomList* must be true/false in any answer sets.
- **assertCautiouslyTrue**("atomList")/**assertCautiouslyFalse**("atomList")  
Equivalent to *assertTrue* and *assertFalse*.
- **assertBravelyTrue**("atomList")/**assertBravelyFalse**("atomList")  
Asserts that all atoms of *atomList* must be true/false in at least one answer set.
- **assertTrueIn**(number, "atomList")/**assertFalseIn**(number, "atomList")  
Asserts that all atoms of *atomList* must be true/false in exactly *number* answer sets.
- **assertTrueInAtLeast**(number, "atomList")/  
**assertFalseInAtLeast**(number, "atomList")  
Asserts that all atoms of *atomList* must be true/false in at least *number* answer sets.
- **assertTrueInAtMost**(number, "atomList")/  
**assertFalseInAtMost**(number, "atomList")  
Asserts that all atoms of *atomList* must be true/false in at most *number* answer sets.
- **assertConstraint**(":-constraint.")  
Asserts that all answer sets must satisfy the given constraint.
- **assertConstraintIn**(number, ":-constraint.")  
Asserts that exactly *number* answer sets must satisfy the given constraint.
- **assertConstraintInAtLeast**(number, ":-constraint.")  
Asserts that at least *number* answer sets must satisfy the given constraint.
- **assertConstraintInAtMost**(number, ":-constraint.")  
Asserts that at most *number* answer sets must satisfy the given constraint.
- **assertBestModelCost**(**intcost**)/**assertBestModelCost**(**intcost**, **intlevel**)  
When executing programs with weak constraints, they assert, that the cost of the best

model with level *intLevel* must be exactly *intCost*.

The string *atomList* specifies a list of atoms to be used with the assertions. These atoms can be ground or non-ground. If they are non-ground the assertion is true if some ground instance of it matches the answer sets (some or all, depending on the assertion).

According to the testing language and the available assertions, a test suite can be composed inside the development environment. The benefits of having an integrated development environment for this task are features like syntax highlighting, auto-completion and a UI-based presentation of the test results. An example of a test suite defined for the 3-colorability problem instance presented in Chapter 2, can be seen in Figure 3.1.



```

1 invocation("TestInvocation", "DLV", "");
2 inputFile("../colorab_run.dl");
3
4 checkNodesRed() {
5     assertTrue(2, "col(1, red).");
6     assertTrue(2, "col(2, red).");
7     assertTrue(2, "col(3, red).");
8 }
9
10 checkNode4Absent() {
11
12 }
13
14

```

assertFalseInAtLeast - Assert False in at least n Answer Sets  
assertFalseInAtMost - Assert False in at most n Answer Sets  
assertConstraint - Assert that all Answer Sets must satisfy the specific  
assertConstraintInAtLeast - Assert that at least n Answer Sets must sa  
assertConstraintInAtMost - Assert that at most n Answer Sets must sa  
assertConstraintIn - Assert that exactly n Answer Sets must satisfy the  
excludeInput - Insert an Input that has to be excluded from a Test  
excludeInputFile - Insert an Input File that has to be excluded from a Te  
assertConstraint(":- constraint");

Figure 3.1: Screenshot of a test suite definition in ASPIDE.

The auto-completion feature of ASPIDE is combined with proper suggestions of suitable keywords or assertions inside the test case. After executing the test suite, the IDE provides an overview of the test results for each test case and assertion. Also the evaluated answer sets are presented in a way, that the tester can understand the cause of the failed test and fix potential errors in the program. While this is a simple example of a test suite, Febraro et al. [Feb+11] show a test suite definition with a more comprehensive example.

### 3.2.2 Language for Annotating Answer Set Programs (LANA)

De Vos et al. [De +12] propose a method to augment answer set programs with additional meta-information formulated using a dedicated annotation language. This language for annotating answer set programs (called LANA) can be used to group rules into blocks, as well as to specify language signatures, types, pre- and postconditions and unit tests for such blocks.

Since all annotations are enclosed in ASP comments, they are invisible to an ASP solver and therefore do not interfere with the program execution. Nevertheless, special tools are able to interpret the annotations for generating documentation, defining unit tests, verification purposes, syntax checking or code completion features. In particular LANA is presented alongside two utilities:

- ASPDoc: used for generating a HTML documentation for an annotated answer set program
- ASPUnit: used for running and monitoring unit tests on blocks (in annotated answer set programs)

Both tools are provided as stand-alone JAR files to be executed on the command prompt with annotated answer set programs as arguments. Furthermore, ASPDoc also operates as part of the integrated development environment SeaLion, which was already presented in Chapter 2.3.2. As this thesis targets unit testing for ASP, hereinafter only the elements relevant for testing, including ASPUnit and the corresponding annotations, will be presented according to De Vos et al. [De +12].

Although the intention of LANA is to write annotations in the same file as the answer set program, ASPUnit requires them to be written in a separate test file (as explained later). All keywords of the annotation language LANA start with the symbol “@”. One basic feature is to group the rules of an answer set program together by defining blocks. A block can be defined with the keyword `@block`, which is shown in Listing 3.3.

**Listing 3.3:** Defining a block in LANA.

```

1 %** @block Facts { *%
2
3 node(1). node(2). node(3).
4 edge(1, 2). edge(1, 3). edge(2, 3).
5
6 %** } *%
7
8 %** @block RulesToTest { *%
9
10 col(X, red) | col(X, blue) | col(X, green) :- node(X).
11
12 :- edge(X, Y), col(X,C), col(Y,C).
13
14 %** } *%

```

While the facts in the lines 3-4 could be part of the block *RulesToTest*, they are not enclosed in it, as they belong to a specific problem instance in this case. Instead the aim is to separate the input of the search problem (instance specific rules) from the rules that perform the problem solving (generate all possible solutions and validate against the integrity constraint). As the block *RulesToTest* can be referenced in several test cases, different input facts for multiple problem instances can be provided. This way the rules under test can be validated against more complex problem instances.

In LANA blocks may be nested but they must not overlap. Like all annotations, the annotations in Listing 3.3 demonstrates how they are written inside of multi-line ASP comments. According to the ASP-Core-2 input language format, a multi-line comment starts with `%*` and ends with `*/`, therefore an additional asterisk “\*” is added to distinguish regular comments from annotations, as seen in Line 4 and Line 10.

Consider a unit test which should check whether node 1 is of color red in at least two

answer sets. According to the answer sets of the program in Listing 3.3, this simple test should pass. In LANA a test case can be defined with the keyword `@testcase` and has to be followed by a name, as demonstrated in Listing 3.4. Additionally, a description of the purpose of the unit test may be given afterwards. The keyword `@scope` is followed by the names of the blocks, that should be tested. In particular the rules of the specified blocks are joined with the subsequent rule that defines `node1Red`. Line 5 expresses that the atom `node1Red` should be true in at least 2 answer sets.

**Listing 3.4:** Defining a test case in LANA.

```

1 %**
2   @testcase CheckNode1Red
3   node 1 must be red in least 2 answer sets
4   @scope Facts, RulesToTest
5   @testatoms node1Red @trueinatleast 2
6 *%
7 node1Red :- col(1, red).

```

Several other test conditions can be used in test case definitions with LANA, for example instead of `@trueinatleast`, a tester can use `@trueinatmost`, `@trueinall`, `@falseinatleast`, etc. Also he could use `@testhasanswerset` or `@testnoanswerset` instead of `@testatoms` to express that at least one or no answer set is expected.

When trying to build a test case, that checks whether `node(1)` is true in exactly two answer sets, LANA is bound to its set of test conditions and modes. Therefore two test cases have to be defined, namely a test case with mode `@trueinatleast 2` and a test case with mode `@trueinatmost 2`, to express “true in exactly two”. For a complete syntactical definition of LANA with regards to the `@testcase` annotation, refer to Table 3.2.

Element	Definition	Informal Description
test case	<code>“@testcase” name</code> <code>[description] “@scope” blocks</code> <code>testcond [ASP code]</code>	A test case for the <i>blocks</i> from blocks passes if the blocks joined with <i>ASP code</i> satisfy all specified test conditions.
testcond	<code>“@testhasanswerset”</code>   <code>“@testnoanswerset”</code>   <code>“@testatoms” atmList mode</code>	A test condition holds if one, resp., no, answer set is found, or all ground atoms in <i>atmList</i> are entailed according to <i>mode</i> .
mode	<code>“@trueinall”</code>   <code>“@trueinatleast” n</code>   <code>“@trueinatmost” n</code>   <code>“@falseinall”</code>   <code>“@falseinatleast” n</code>   <code>“@falseinatmost” n</code>	Mode of entailment of a test condition, all, at most <i>n</i> , or at least <i>n</i> answer-sets.

**Table 3.2:** Test case specification in LANA. (Adapted from De Vos et al. [De +12])

Despite its limited set of test conditions and modes, LANA is a powerful annotation-based language for defining unit tests. For a comprehensive validation of blocks, De Vos et al. [De +12] also propose preconditions and postconditions. As they validate the inputs,

respectively the outputs of a block, they can rather be seen as a contract (than as a unit test).

Alongside with LANA, De Vos et al. [De +12] also provide an implementation in form of a command-line tool called ASPUnit. This tool is provided as an executable JAR file. For execution, ASPUnit requires each unit test to be stored in a separate file. Additionally, the tool takes a test-suite specification file as input, which is a file containing relevant information regarding the location of the answer set program, the files containing test cases and the ASP solver that is used to execute them. While the syntax of the test-suite specification file is related to the annotation language, the content is not enclosed in ASP comments. An example of a specification file might look as shown in Listing 3.5.

**Listing 3.5:** Test suite specification example. (Adapted from De Vos et al. [De +12])

```

1 @testsuite name
2 description
3 @program ASPfiles
4 @programdir pathToASPfile
5 @test testCaseFile1
6 @test testCaseFile2
7 ...
8 @testdir pathToTestFiles
9 @solvertype ASPsolver
10 @solver solverFile
11 @grunder grunderFile

```

A test suite specification start with `@testsuite` followed by a name. A short description may be given afterwards. `@program` is followed by a list of files that contain the program under test, while they are located relative to the path specified after `@programdir`. Subsequently, the names of the files containing the test cases have to be specified with the keyword `@test`. The base path to the test files is defined after `@testdir`. The keyword `@solvertype` specifies the type of solver, that is used, which is one of DLV, Clasp or Clingo. The absolute path to the solver is defined after `@solver`, while a separate grunder can be used with `@grunder` [De +12].

After the test cases are defined and the test suite specification is configured correctly, ASPUnit can be invoked as follows:

```
java -jar aspunit.jar mytestsuite
```

ASPUnit will execute all unit tests on the specified answer set program using the solver settings in the test suite specification. The results are printed as a report, which contains the information regarding the success or failure for each test case. If needed, also a counterexample, as well as a short description, can be included in the report when the test case failed. A test report might look as shown in Listing 3.6, which is adapted from an example presented by De Vos et al. [De +12].

**Listing 3.6:** A report of ASPUnit for a test suite (Adapted from De Vos et al. [De +12]).

```

Test Case ShipTopLeftCorner: Successful

Test Case NoDiagonalShips : Successful

Test Case TouchingShips : Failed
    two ships must not touch each other

```



```
Failed Test : @falseinall forbiddenShip
Counterexample:
Answer set:
  c(1).c(10).c(2).c(3).c(4).c(5).
  c(6).c(7).c(8).c(9).forbiddenShip.
  r(1).r(10).r(2).r(3).r(4).
  r(5).r(6).r(7).r(8).r(9).
  ship(1, 1, 1, 2).ship(1, 2, 1, 4).
```

## Chapter 4

# Annotation-based Testing for ASP

After dealing with preliminary topics in Chapter 2, the related work was presented in Chapter 3. Based on the revision of previous work in the context of this thesis, a valuation of the methods and technologies is made hereinafter. This is important to evaluate advantages as well as drawbacks of specific concepts. Beneficial key concepts of certain approaches may have an influence on this thesis, which is determined in the following Section. Afterwards, the solution approach of this thesis is presented. This includes the specification of an annotation-based language for defining unit tests and the execution mechanism. Both of them are provided alongside several examples to demonstrate the intention of this solution approach.

### 4.1 Valuation of previous work

With the introduction of a general framework for unit testing in ASP, Janhunen et al. [Jan+10] have undoubtedly laid down the basic foundations. The theoretical formalization of a test case, test suite and the conditions for passing, respectively failing, tests is accord with its meaning in conventional programming languages and testing frameworks. Additionally, different coverage metrics for answer set programs are presented. Since most definitions are provided alongside meaningful examples, the findings are well founded and sound. Consequently, this work can be considered as common ground when implementing a testing framework for answer set programming. Alternative interpretations of the included concepts have not been found, which is why they are fully adopted for this thesis.

Considering unit testing in ASP, Febbraro et al. [Feb+11] have presented ASPIDE which is a development environment capable of unit testing as a fully integrated feature. This includes a feature-rich support for developers, as well as a comfortable execution environment. While utilizing annotations inside the answer set program, unit testing with ASPIDE still requires at least one additional test file including the test suite specification. This has to be written in a specific testing language, which is parsed and executed only by the IDE. Due to the extensiveness of the language, manifold test cases can be defined to cover a variety of test scenarios. Also referencing multiple ASP files for testing is possible, which allows the realization of test suites in large scale applications and complex problem instances. The idea of using rule naming inside of ASP files is reasonable and therefore also targeted with this thesis. Additionally, the variety of assertion types in test case definitions is useful when implementing unit tests and therefore also adopted. On the contrary, the overhead of having at least one additional test file could be avoided. With ASPIDE following a suboptimal naive

approach for test execution, which is evaluating the answer sets of the program and checking the output of the ASP system, a performance increase in this area can also be targeted.

LANA is an annotation language, whose aim is not purely testing. Despite the possibility to define unit tests (ASPUnit), it also aims to provide pre- and postconditions for blocks or to add metadata for generating documentation (ASPDoc). Hence, the assertion types for test case specifications are not as complete as in ASPIDE. While LANA can be written inside an answer set program, without the need for an additional file, the command-line tool ASPUnit requires each test case to be written in a separate file. Moreover, the annotations extend the original ASP file, which can result in worse readability of the program. With regards to the test assertion types, LANA does not add new functionality to this thesis. Nonetheless, it should not require, but enable the tester to write tests in a separate file (as ASPUnit does), which is desired with this thesis. Since the test annotations appear to be compact, some properties may also be adopted.

## 4.2 Solution approach

As this thesis aims not to provide just a theoretical specification of an annotation-based language, but also a practical implementation, all statements and goals are formulated considering their feasibility in terms of implementation. After evaluating previous work, the aspiration is to combine the best of both ASPIDE and LANA, to form a new annotation-based language for unit testing in conjunction with an efficient execution mechanism. Thus, key aspects of both are (cherry) picked, refined and redeveloped to achieve this goal.

To provide a satisfying developer experience when writing unit tests in ASP, the definition of an easy to use and pleasing annotation language is essential. Similarly, the execution should be realized in an efficient way, to provide acceptable response times better than a naive approach (evaluating all answer sets and checking the output). Accordingly, the developments of this thesis are split into two parts. The annotation language aims to provide a comfortable way of implementing unit tests, while long familiarization phases due to a complex syntax have to be avoided. This could be achieved by sticking to annotation syntaxes known from conventional programming languages as Java or C#. Moreover, the possibility to express manifold test cases must be provided. The execution mechanism, on the other hand, should be able to read the annotations and execute them adequately. In ASP the execution performance for specific problem instances can be increased by rearranging rules and constraints. This way an efficient execution, which is better than the naive approach can be realized. After developing an annotation language and the corresponding execution mechanism (in the following sections) also an implementation as part of a web-based development environment is provided to demonstrate the feasibility (proof of concept) in Chapter 5.

## 4.3 Annotation Language

The syntax of the annotation language is inspired by the annotation style of Java. Being a widely used programming language, long familiarization phases can be avoided. Its annotation style gives the possibility to express complex (meta-)data while staying compact at the same time. Moreover, it provides structures for defining lists as values/properties of the annotation, which is useful for defining unit tests in ASP. Listing 4.1 shows an example of a complex

annotation in the context of the Java Persistence API<sup>1</sup>. Here annotations usually start with the symbol “@”, as in ASPIDE or LANA. Additionally, they can be nested and its values can hold lists (of any type).

**Listing 4.1:** Adopted example of annotations from the Java Persistence API.

```

1 @SqlResultSetMapping(
2   name="OrderResults",
3   entities={
4     @EntityResult(
5       entityClass=Order.class,
6       fields={
7         @FieldResult(name="id",      column="order_id"),
8         @FieldResult(name="quantity", column="order_quantity"),
9         @FieldResult(name="item",    column="order_item")
10      }
11    ),
12    @EntityResult(
13      entityClass=Item.class,
14      fields={
15        @FieldResult(name="id",      column="item_id"),
16        @FieldResult(name="name",    column="item_name")
17      }
18    )
19  }
20 )
21 @Entity
22 public class Order {
23   @Id
24   protected int id;
25   protected long quantity;
26   @ManyToOne
27   protected Item item;
28   ...
29 }

```

Two concepts of previous works that have been evaluated are rule naming (ASPIDE) and the aggregation of rules in form of blocks (LANA). These concepts are adopted for the new annotation language and enhanced to provide more practicality. Alongside with these structures, a new syntax for defining unit test cases is introduced. As they should be ignored by any ASP solving system, they must be enclosed in multi-line comments inside the answer set program according to the ASP-Core-2 input language format. In particular, to distinguish annotations from regular comments, all annotations of the language must be enclosed between **%\*\*** and **\*\*%**. All in all, the annotation language consists of two parts, the base annotations and the assertion annotations, which are explained in the following subsections. A full grammar specification of the annotation language can be found as an appendix to this thesis (Chapter A).

### 4.3.1 Base Annotations

The **@rule**, **@block** and **@test** annotations constitute the base annotations of the language. They can be written anywhere in the answer set program, except the **@rule** annotation. It must be followed by the rule it should reference. Listing 4.2 shows a definition of the

<sup>1</sup>Example adopted from <https://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-096251.html> (accessed 2019-05-02)

`@rule` annotation, that is mainly used for rule naming. In this case the subsequent rule is named “myRule”. Moreover, this rule is also assigned to a block “someBlock”, which should be defined elsewhere. Nonetheless, this assignment is optional.

**Listing 4.2:** Definition of the `@rule` annotation.

```
1 @rule(
2     name = "myRule",
3     block = "someBlock"
4 )
```

The `@block` annotation can be written somewhere in the answer set program. Contrary to LANA, blocks do not enclose a set of rules, instead rules are either assigned to it, or they reference the covered rules. To assign certain rules to an already defined block, the property “block” of the `@rule` annotation can be utilized, as shown above. Nevertheless, it can be more pleasing for a tester to reference the rules, that the block covers, directly within the definition as shown in Listing 4.3. Here a block “someBlock” is defined and the rules “rule1”, “rule2” and “rule3” are covered by it. The definition of the list “rules” is optional.

**Listing 4.3:** Definition of the `@block` annotation.

```
1 @block(
2     name = "someBlock" ,
3     rules = { "rule1", "rule2", "rule3" }
4 )
```

With regards to the `@test` annotation, more theory is involved. Listing 4.4 shows an example utilizing all properties of the annotation. First, the annotation can be written somewhere inside the answer set program, but also in a separate file. A name can be specified by the “name” property. The rules, respectively blocks to test can be selected with the property “scope”, which is a list of names, similar to LANA. Nonetheless, instead of only referencing blocks, also rule names can be used here.

Each test case definition acquires a pool of visible rules and blocks, which can be referenced by their names (to be used in the test). This pool contains rules and blocks of the current file, if and only if the property “programFiles” is not defined. Otherwise only the rules and blocks of the files in “programFiles” are available for the test. In order to separate the rules that perform the problem solving (instance independent rules) from the rules that define the input for the problem (instance dependent rules), the input can be specified individually using the properties “input” and “inputFile”. While “input” is intended for just a few additional facts/rules, the files in “inputFiles” can contain extensive data for complex problem instances. Both properties are optional, in case no additional input is needed for the test. As also presented in ASPIDE and LANA, assertions can be made with the property “assert”, which is a list of assertion to be fulfilled to pass the test case. The placeholder “assertionList” written in bold in line 7 of Listing 4.4 holds a comma separated list of assertion annotations.

**Listing 4.4:** Definition of the `@test` annotation.

```
1 @test(
2     name = "check valid color assignment",
3     scope = { "rule1", "rule2", "inputFactsBlock" },
4     programFiles = { "3-colorability.dl" },
5     input = "node(4). edge(3, 4).",
6     inputFiles = { "additional-facts-3-colorab.dl" },
7     assert = { assertionList }
8 )
```

### 4.3.2 Assertion Annotations

Assertion annotations constitute the second part of the annotations. They can only be used inside the “assert” property of the `@test` annotation, to define specific assertions on the test instance. The assertion types are mostly adapted from ASPIDE, while some types were added or modified. Table 4.1 contains all annotations including a description. With regards to the first four annotations (`@trueInAll`, `@trueInAtLeast`, `@trueInAtMost` and `@trueInExactly`) also the counterparts `@falseInAll`, `@falseInAtLeast`, `@falseInAtMost` and `@falseInExactly` are defined. Moreover, the use of the assertion `@bestModelCost` is intended when testing a program with weak constraints.

### 4.3.3 Usage

When combining the base annotations with the assertion annotations, complex unit test cases for answer set programs can be defined. Considering the basic 3-colorability problem instance from Chapter 2, unit tests can be defined in the same program file as shown in Listing 4.5. This instance produces 6 answer sets, in detail the color red is assigned to each node in two answer sets, as already shown. Therefore, the assertions `@trueInExactly`, `@trueInAtLeast` and `@trueInAtMost` can be demonstrated with this example.

**Listing 4.5:** Example test cases for a basic 3-colorability problem instance.

```

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

```

Lines 4 and 6 contain `@rule` annotations to assign names to the rules. Since the aim is to test only rules that perform the problem solving (instance independent rules), no names are assigned to the input facts in the lines 1 and 2. This way we can reference the rules we want to test in the scope of the test case (property “scope” in line 12) and potentially define a different set of inputs (property “input”). Nevertheless, for simplicity the same input facts are kept in line 13. As for this test instance the color red is assigned to each node exactly 2 times, all assertions of the lines 15-17 will pass, which will cause the whole unit test to pass. The annotation block enclosed between `%%` and `%%` (lines 9-20) may also contain further tests besides “checkNodesRed”.

Table 4.1: The assertion annotations

Assertion	Description
<code>@noAnswerSet</code>	The test must have no answer set.
<code>@trueInAll(   atoms = "aspAtoms" )</code>	The atoms specified in <i>aspAtoms</i> must be true in all answer sets.
<code>@trueInAtLeast(   number = num,   atoms = "aspAtoms" )</code>	The atoms specified in <i>aspAtoms</i> must be true in at least <i>num</i> answer sets.
<code>@trueInAtMost(   number = num,   atoms = "aspAtoms" )</code>	The atoms specified in <i>aspAtoms</i> must be true in at most <i>num</i> answer sets.
<code>@trueInExactly(   number = num,   atoms = "aspAtoms" )</code>	The atoms specified in <i>aspAtoms</i> must be true in exactly <i>num</i> answer sets.
<code>@constraintForAll(   constraint = "aspConstraint" )</code>	The constraint specified in <i>aspConstraint</i> must be fulfilled in all answer sets.
<code>@constraintInAtLeast(   number = num,   constraint = "aspConstraint" )</code>	The constraint specified in <i>aspConstraint</i> must be fulfilled in at least <i>num</i> answer sets.
<code>@constraintInAtMost(   number = num,   constraint = "aspConstraint" )</code>	The constraint specified in <i>aspConstraint</i> must be fulfilled in at most <i>num</i> answer sets.
<code>@constraintInExactly(   number = num,   constraint = "aspConstraint" )</code>	The constraint specified in <i>aspConstraint</i> must be fulfilled in exactly <i>num</i> answer sets.
<code>@bestModelCost(   cost = costValue,   level = levelValue )</code>	The best model has to meet the cost of <i>costValue</i> at level <i>levelValue</i> (in case of executing weak constraints).

In the light of an example containing weak constraints also the use of the annotation `@bestModelCost` can be demonstrated. Accordingly, the example in Listing 4.6 incorporates weak constraints with the goal to find an answer set where firstly the color of node 1 is red (lowest weight at highest level) and secondly node 2 has the optimal color (according to the weak constraints on level 1), which is green with weight 2 in this case.

**Listing 4.6:** Example test cases for an extended 3-colorability problem instance.

```

1 node(1). node(2). node(3).
2 edge(1, 2). edge(1, 3). edge(2, 3).
3
4 %** @block(name="rulesToTest") **%
5
6 %** @rule(name="r1", block="rulesToTest") **%
7 col(X, red) | col(X, blue) | col(X, green) :- node(X).
8 %** @rule(name="r2", block="rulesToTest") **%
9 :- edge(X, Y), col(X,C), col(Y,C).
10
11 %** @rule(name="r3", block="rulesToTest") **%
12 :- col(1, red). [1 @ 2]
13 %** @rule(name="r4", block="rulesToTest") **%
14 :- col(1, green). [2 @ 2]
15 %** @rule(name="r5", block="rulesToTest") **%
16 :- col(1, blue). [3 @ 2]
17
18 %** @rule(name="r6", block="rulesToTest") **%
19 :- col(2, red). [1 @ 1]
20 %** @rule(name="r7", block="rulesToTest") **%
21 :- col(2, green). [2 @ 1]
22 %** @rule(name="r8", block="rulesToTest") **%
23 :- col(2, blue). [3 @ 1]
24
25 %**
26 @test(
27     name = "checkBestModelCost",
28     scope = { "rulesToTest" },
29     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
30     assert = {
31         @bestModelCost(cost = 1, level = 2),
32         @bestModelCost(cost = 2, level = 1)
33     }
34 )
35 %**

```

In contrast to the example with the basic 3-colorability problem instance (Listing 4.5), in Listing 4.6 the use of a block is advisable. The block “rulesToTest” covers 8 rules, which would have to be referenced explicitly in the property “scope” in line 28 otherwise. Instead only the block is referenced and the assertion annotations are specified in the lines 31 and 32. The first assertion specifies, that the weight at level 2 (higher priority) is 1. Secondly, line 32 asserts, that the weight at level 1 (lower priority) should be 2. Since the optimal answer set (best model) for this problem instance will contain the ground atoms `col(1, red)` and `col(2, green)`, both assertions, respectively the test case will pass.

While in the previous examples the test case definitions were written as part of the program file, they can also be written in a separate testing file. Consequently, also test suites can be realized by writing a set of test cases into test files and referencing the original program files with the property “programFiles” of `@test`. Similarly complex problem instances, for



example in form of a set of input facts, can be provided in a separate ASP file and referenced with the property “inputFiles”.

## 4.4 Execution mechanism

After developing an annotation-based language for unit testing in Section 4.3, the execution is the second essential part of this thesis. While a naive approach for executing unit tests, namely to evaluate all answer sets and checking the output (as done in ASPIDE), is straight forward to implement, more efficient executions in terms of the number of evaluated answer sets can be achieved. This increase in efficiency can be often accomplished by modifying the program under test and limiting the number of answer sets to be evaluated. Still, all modifications and limitations have to be carried out in a way, such that the outcome of a test case, in particular the assertions, can be determined with certainty. Hence, a special process for test execution was developed to fulfill the stated requirements.

### 4.4.1 Testing engine

The test execution mechanism of this thesis is divided into four phases and utilizes an answer set solving system. The phases consist of:

1. Reading annotations present in the program
2. Creating an intermediate program based on the test cases and assertions
3. Executing intermediate program on solving system
4. Evaluating outcome according to the test case definition

All four steps can be imagined to be carried out by a tool that has access to an ASP system (like Clingo or DLV2) for evaluating answer sets. For simplicity this tool is also referred to as “testing engine” hereinafter.

Firstly, the testing engine is able to read the annotations according to the syntax of the language in order to handle different types of assertions and other given parameters. In particular, this part can be seen as a parsing unit, that constructs an internal model of a test suite containing test cases, each of them possibly containing several assertions. This internal model can be used in the next step.

The generation of the intermediate program is the most important part in the process. It guarantees the efficiency of the execution. In detail, specific answer set rules are added to the original program, depending on the assertion type to validate. Thus, a rewriting of the original program may happen for each assertion type in a test case, as each type has a different purpose (demonstrated in Subsection 4.4.2). All in all, the aim of the intermediate program is to be able to limit the number of evaluated answer sets and still determine the outcome of the assertion. A benefit of intermediate programs is, that the ASP system performs most of the test execution. Since the intermediate program is written in a certain way, for most of the assertions the number of evaluated answer sets can be limited. Additionally, except the assertion type `@bestModelCost`, the actual answer sets produced by the solver do not need to be touched to evaluate the outcome, instead only the number of resulting models is significant.

As already stated, due to the need of a program rewriting, an intermediate program translation is required for each assertion type available. In conclusion, the testing engine then reads the test definition, creates an intermediate program for the given assertion type, executes it on the ASP system and evaluates the outcome in a way, that is more efficient than

the naive approach. A more detailed explanation of the process is given in Subsection 4.4.2 with an example for most assertion types.

#### 4.4.2 Test execution

Alongside with the definition of the process performed by the testing engine in Subsection 4.4.1, this Subsection provides the intermediate program translations for each assertion type of the annotation language. Instead of specifying them in a formal way, each translation is demonstrated with a unit test example of the known 3-colorability problem instance from Section 2. Thus, the ASP program in Listing 4.7 is reused for each test case definition hereinafter.

**Listing 4.7:** 3-colorability problem instance for test demonstration.

```

1 %** @block(name="rulesToTest") **%
2
3 %** @rule(name="r1", block="rulesToTest") **%
4 col(X, red) | col(X, blue) | col(X, green) :- node(X).
5
6 %** @rule(name="r2", block="rulesToTest") **%
7 :- edge(X, Y), col(X,C), col(Y,C).

```

#### @trueInAll

The assertion @trueInAll expresses, that certain atoms should be true in all answer sets. An example of a test case using this assertion is shown in Listing 4.8.

**Listing 4.8:** Test case example for @trueInAll.

```

1 %**
2 @test(
3     name = "checkTrueInAll",
4     scope = { "rulesToTest" },
5     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6     assert={
7         @trueInAll(atoms = "col(1, red).")
8     }
9 )
10 **%

```

For translating this assertion into an intermediate program, an integrity constraint containing the asserted atoms is added to the program under test. The resulting intermediate program is shown in Listing 4.9.

**Listing 4.9:** Intermediate program translation for Listing 4.8.

```

1 node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).
2 col(X, red) | col(X, blue) | col(X, green) :- node(X).
3 :- edge(X, Y), col(X,C), col(Y,C).
4 :- col(1, red).

```

By adding the integrity constraint `:- col(1, red).` the program aims to find an answer set that does not fulfill the assertion. The execution will determine if there is a model, while the solver is configured to search for only one answer set. Based on that, the test fails, respectively passes if there are no answer sets. For this example the test will fail, as there are answer sets containing the atom `col(1, red).`

**@trueInAtLeast**

When asserting that certain atoms should be true in at least  $n$  answer sets, the `@trueInAtLeast` assertion can be used. Listing 4.10 shows an example of a test case using this assertion.

**Listing 4.10:** Test case example for `@trueInAtLeast`.

```

1 %**
2   @test(
3     name = "checkTrueInAtLeast",
4     scope = { "rulesToTest" },
5     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6     assert={
7       @trueInAtLeast(number = 2, atoms = "col(1, red).")
8     }
9   )
10 **%
```

The translation of this assertion into an intermediate program will add an integrity constraint containing the asserted atoms to the program under test. Listing 4.11 shows the rewritten program for the given test case which is used for test execution.

**Listing 4.11:** Intermediate program translation for Listing 4.10.

```

1 node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).
2 col(X, red) | col(X, blue) | col(X, green) :- node(X).
3 :- edge(X, Y), col(X,C), col(Y,C).
4 :- not col(1, red).
```

In this case the integrity constraint `:- not col(1, red).` assures, that the program aims to find answer sets that contain the atom `col(1, red)`. The execution will determine if there are at least  $n$  models, while the solver is configured to search for  $n$  answer sets. Depending on the number of found answer sets, the test fails, respectively passes if there are  $n$  answer sets. In this case, the test will pass, as there are exactly 2 answer sets containing `col(1, red)`.

**@trueInAtMost**

This assertion can be used to assure that certain atoms are true in at least  $n$  answer sets. Accordingly Listing 4.12 shows an example of a test case using `@trueInAtMost`.

**Listing 4.12:** Test case example for `@trueInAtMost`.

```

1 %**
2   @test(
3     name = "checkTrueInAtMost",
4     scope = { "rulesToTest" },
5     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6     assert={
7       @trueInAtMost(number = 1, atoms = "col(1, red).")
8     }
9   )
10 **%
```

The intermediate program for this example will contain the program under test and an additional constraint with the asserted atoms. In particular, the intermediate program translation is equivalent to `@trueInAtLest`, which was already shown in Listing 4.11.

Nonetheless, the execution differs from `@trueInAtLeast`. In this case the execution will determine if there are more than  $n$  models, while the solver is configured to search for  $n + 1$  answer sets. Based on the results, the test fails, respectively passes if there are  $n$  or less answer sets. For this test case, the test will fail, as there are exactly 2 answer sets containing `col(1, red)`.

### **@trueInExactly**

To assert that certain atoms must be true in exactly  $n$  answer sets, the assertion `@trueInExactly` can be utilized. Consequently, Listing 4.13 demonstrates its usage in a test case.

**Listing 4.13:** Test case example for `@trueInExactly`.

```

1 %**
2   @test(
3     name = "checkTrueInExactly",
4     scope = { "rulesToTest" },
5     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6     assert={
7       @trueInExactly(number = 2, atoms = "col(1, red).")
8     }
9   )
10 **%
```

Similar to `@trueInAtLeast` and `@trueInAtMost`, this assertion produces the same intermediate program as seen in Listing 4.11. An integrity constraint containing the asserted atoms are added to the program under test.

Again, the aim is to only find answer sets containing the atom `col(1, red)`. Here the execution will evaluate if there are exactly  $n$  answer sets, while the solver is configured to search for  $n + 1$  models. Based on that, the test fails if there are less or more than  $n$ , respectively passes if there are exactly  $n$  answer sets. In this example, the test will pass, as there are exactly 2 answer sets containing `col(1, red)`.

### **@falseInAll, @falseInAtLeast, @falseInAtMost and @falseInExactly**

With regards to the annotations `@falseInAll`, `@falseInAtLeast`, `@falseInAtMost` and `@falseInExactly`, no explicit examples are presented. Being counterparts of the annotations `@trueInAll`, `@trueInAtLeast`, `@trueInAtMost` and `@trueInExactly`, their intermediate programs and evaluation can be derived, based on the previous subsections.

### **@constraintForAll**

The assertion `@constraintForAll` can be used to express that a constraint must be satisfied in all answer sets. A test case example using `@constraintForAll` can be seen in Listing 4.14.

**Listing 4.14:** Test case example for `@constraintForAll`.

```

1 %**
2   @test(
3     name = "checkConstraintForAll",
4     scope = { "rulesToTest" },
5     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6     assert={
7       @constraintForAll(constraint = ":- col(1, red).")
8     }
9   )
10 **%
```

```

8     }
9     )
10  **%

```

The intermediate program translation will contain some additional auxiliary atom, for example *assertionHelperAtom*, and a constraint as seen in Listing 4.15.

**Listing 4.15:** Intermediate program translation for Listing 4.14.

```

1 node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).
2 col(X, red) | col(X, blue) | col(X, green) :- node(X).
3 :- edge(X, Y), col(X,C), col(Y,C).
4 assertionHelperAtom :- col(1, red)
5 :- not assertionHelperAtom.

```

By adding these rules, the program aims to find an answer set that does not satisfy the constraint. Therefore, the execution will determine if there are models, while the solver is configured to search for one answer set. Based on the result, the test fails, respectively passes if there is no answer set. For this example it will fail, as there are answer sets containing `col(1, red)`.

### @constraintInAtLeast

When asserting that a constraint must be satisfied in at least  $n$  answer sets, the assertion `@constraintInAtLeast` can be utilized. Listing 4.16 shows an example of how to use it in a test case.

**Listing 4.16:** Test case example for `@constraintInAtLeast`.

```

1  %**
2    @test(
3      name = "checkConstraintInAtLeast",
4      scope = { "rulesToTest" },
5      input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6      assert={
7        @constraintInAtLeast(number = 1, constraint = ":- col(1, red).")
8      }
9    )
10  **%

```

For this assertion, the intermediate program translation will contain the program to test and the specified constraint as is (Listing 4.17).

**Listing 4.17:** Intermediate program translation for Listing 4.16.

```

1 node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).
2 col(X, red) | col(X, blue) | col(X, green) :- node(X).
3 :- edge(X, Y), col(X,C), col(Y,C).
4 :- col(1, red).

```

This way the program only searches for answer sets, that fulfill the specified constraint. Consequently, the execution will determine if there are at least  $n$  models, while the solver is configured to search for  $n$  answer sets. By checking the number of models, the test fails, respectively passes if there are  $n$  answer sets. For this instance, the test will pass, as there are 2 answer sets containing `col(1, red)`.

**@constraintInAtMost**

The annotation `@constraintInAtMost` can be utilized to assert, that a constraint must be satisfied in at most  $n$  answer sets. The usage is shown with a test case in Listing 4.18.

**Listing 4.18:** Test case example for `@constraintInAtMost`.

```

1 %**
2   @test(
3     name = "checkConstraintInAtMost",
4     scope = { "rulesToTest" },
5     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6     assert={
7       @constraintInAtMost(number = 3, constraint = ":- col(1, red).")
8     }
9   )
10 **%
```

With this assertion, the intermediate program is equivalent to `@constraintInAtLeast` as presented in Listing 4.17. This program aims to only find answer sets that fulfill the constraint.

Nevertheless, the execution differs, as with `@constraintInAtMost` the execution checks if there are  $n$  models, while the solver is configured to search for  $n + 1$  answer sets. Based on that, the test fails, respectively passes if there are  $n$  or less answer sets. In this example, the test passes because there are 2 answer sets containing `col(1, red)`.

**@constraintInExactly**

To assert, that a certain constraint must be satisfied in exactly  $n$  answer sets, the annotation `@constraintInExactly` can be used, as shown in the following Listing 4.19.

**Listing 4.19:** Test case example for `@constraintInExactly`.

```

1 %**
2   @test(
3     name = "checkConstraintInExactly",
4     scope = { "rulesToTest" },
5     input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6     assert={
7       @constraintInExactly(number = 1, constraint = ":- col(1, red).")
8     }
9   )
10 **%
```

Although the intermediate program and its target is equivalent to `@constraintInAtLeast` and `@constraintInAtMost`, the evaluation differs. The execution will determine if there are exactly  $n$  answer sets, while the solver is configured to search for  $n + 1$  models. The assertion passes if there are exactly  $n$  answer sets, respectively fails otherwise. In this case, the test fails because there are 2 answer sets containing `col(1, red)`.

**@bestModelCost**

`@bestModelCost` asserts a certain weight  $w$  at level  $l$ . It can be used in answer set programs incorporating weak constraints. Therefore, the basic 3-colorability problem instance is not sufficient here. Instead consider the extended 3-colorability problem instance and the test case of Listing 4.20.

**Listing 4.20:** Test case example for `@bestModelCost`.

```

1 %** @block(name = "rulesToTest") **%
2
3 %** @rule(name="r1", block="rulesToTest") **%
4 col(X, red) | col(X, blue) | col(X, green) :- node(X).
5 %** @rule(name="r2", block="rulesToTest") **%
6 :- edge(X, Y), col(X,C), col(Y,C).
7
8 %** @rule(name="r3", block="rulesToTest") **%
9 :- col(1, green). [3 @ 1]
10 %** @rule(name="r4", block="rulesToTest") **%
11 :- col(1, blue). [2 @ 1]
12 %** @rule(name="r5", block="rulesToTest") **%
13 :- col(1, red). [1 @ 1]
14
15 %**
16     @test(
17         name = "checkBestModelCost",
18         scope = { "rulesToTest" },
19         input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
20         assert = {
21             @bestModelCost(cost = 1, level = 1)
22         }
23     )
24 %**

```

For the execution of this assertion, no intermediate program translation is performed. The testing engine executes the program under test and checks the optimal answer set afterwards. The weight at level  $l$  has to match the weight specified in the assertion, namely  $w$ . This way the test passes, respectively fails otherwise. For this example, the test will pass, as the optimal answer set contains `col(1, red)` with weight 1 at level 1.

### **@noAnswerSet**

When asserting that an answer set program has no answer set, the annotation `@noAnswerSet` can be used. It can be written inside the property “assert” like any other annotation, as demonstrated in Listing 4.21.

**Listing 4.21:** Test case example for `@noAnswerSet`.

```

1 %**
2     @test(
3         name = "checkHasNoAnswerSet",
4         scope = { "rulesToTest" },
5         input = "node(1). node(2). node(3). edge(1, 2). edge(1, 3). edge(2, 3).",
6         assert = {
7             @noAnswerSet
8         }
9     )
10 %**

```

An intermediate program translation is not required for this assertion and the solver is configured to evaluate one answer set. If no answer set is found, the test passes, respectively fails otherwise. For the given example, the test will fail, as the program has several answer sets.

## Chapter 5

# Implementation

The implementation is an important aspect of this thesis, showing the feasibility of implementing the annotation-based language for unit testing in ASP. The intention is to provide a ready to use tool for unit testing, that also acts as a proof of concept for the presented work. While tooling in ASP is often implemented in form of a command line utility, for example ASPDoc or ASPUnit (LANA), the goal is to develop a testing environment with a user interface called “ASP-WIDE”, which stands for “Answer Set Programming - Web-based Integrated Development Environment”. The idea is to provide a tool, which is comfortable to use, in order to increase the acceptance among ASP developers. Nevertheless, the integration of a comprehensive set of features is not intended, as the realization of a feature-rich development environment is not in the scope of this thesis. Instead, only the inclusion of basic features, which are required for ASP development and unit testing according to this thesis, is planned.

In the following sections the architecture and design of the implementation is presented. Afterwards, appropriate technologies are selected, for realizing the testing environment in the light of future enhancements. This should guarantee a future-proof implementation, that can be extended, for example in form of an open source project. Also implementation details in form of source code are presented according to the architecture. Especially the realization of the annotation language and the testing mechanism are highlighted. Finally, the resulting development environment ASP-WIDE is presented showing its user interface and functionality.

### 5.1 Editors for Logic Programming

For realizing ASP-WIDE, existing development environments have been analyzed first. Although this thesis considers ASPIDE as a comprehensive IDE for ASP, different editors for logic programming at a smaller scale have been released. Germano, Calimeri, and Palermi [GCP17] cite IDP Web-IDE<sup>1</sup>, SWISH<sup>2</sup>, LogiQL REQPL<sup>3</sup>, PDDL Editor<sup>4</sup>, Clingo in the Browser<sup>5</sup>, dlhex Online Demo<sup>6</sup> and more for being web-based editors for logic programming.

---

<sup>1</sup>URL: <http://adams.cs.kuleuven.be/idp/server.html> (accessed 2019-04-29)

<sup>2</sup>URL: <https://swish.swi-prolog.org/> (accessed 2019-04-29)

<sup>3</sup>URL: <https://repl.logicblox.com/?> (accessed 2019-04-29)

<sup>4</sup>URL: <http://editor.planning.domains/> (accessed 2019-04-29)

<sup>5</sup>URL: <http://potassco.sourceforge.net/clingo.html> (accessed 2019-04-29)

<sup>6</sup>URL: <http://www.kr.tuwien.ac.at/research/systems/dlhex/demo.php> (accessed 2019-04-29)



Moreover, they also present a new web-based IDE called LoIDE<sup>7</sup>. Following their findings, a trend towards web-based solutions may be observed. Furthermore, a common set of features among all editors and IDEs can be recognized. Mostly all of them provide a code editor with syntax highlighting and the possibility to execute answer set programs. On the contrary not all of them provide the possibility to manage multiple files, for example in a file explorer. Nevertheless, the current file can be validated and errors are shown either in the line they occur or in a separate output window. Also answer sets are presented in an output window. While ASPIDE includes various output representation modes (console, table, visualizator, etc.), simpler editors stick to a console output similar to the output of an ASP system. With LoIDE, the developer is also able to select the ASP system he wants to use for executing the current program.

## 5.2 Architecture and Design

After evaluating existing editors and IDEs for logic programming (Section 5.1), a common set of minimal features can be fixed, that must be part of a development environment for ASP. Together with the features for implementing annotation-based testing, the desired set of features for ASP-WIDE includes:

- **Workspace management:** A file explorer enables the user to do basic file operations like creating and deleting ASP files.
- **Code editor:** The code editor for ASP files is an essential part offering syntax highlighting and error markers.
- **Execution/Testing environment:** This offers the possibility to execute both answer set programs and unit tests inside the development environment.
- **Output window:** The output of the executed answer set program, as well as the outcome of the executed tests must be displayed in a suitable way.

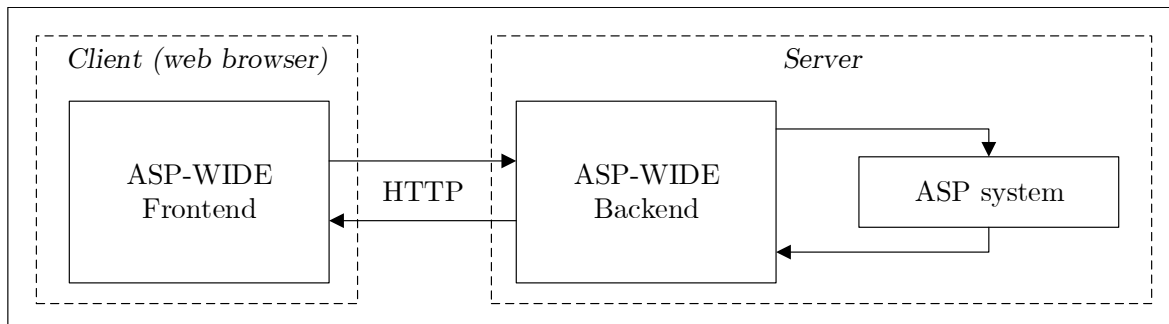
Since web technologies provide proper ground for implementing such tooling and several editors have been built with them, also this thesis is realized as a web-based application. Therefore, the architecture of ASP-WIDE is split into two parts, which are often found among web applications. One part is downloaded by the user, who wants to interact with the application. In particular HTML, CSS and JavaScript resources are loaded and rendered in the user's web browser (client), which is called "frontend". The "backend" is usually located on a web-server, that performs specific server-side operations. Typically frontend and backend communicate over HTTP messages sending JSON data. Figure 5.1 illustrates the overall architecture of ASP-WIDE including frontend, backend and an ASP system. The use of an ASP system is mandatory, as the program execution and also test execution depend on it. Nonetheless, only the backend has direct access to it.

## 5.3 Technologies

Once the base architecture of ASP-WIDE was fixed in Section 5.2, the technologies can be selected. All technologies are selected with respect to possible further developments, that go beyond the scope of this thesis, for example in form of an ongoing open source project.

Starting with the Backend, a technology capable of handling HTTP requests from multiple clients and executing specific server-side logic, for example calling an ASP system, is

<sup>7</sup>URL: <https://github.com/DeMaCS-UNICAL/LoIDE> (accessed 2019-04-29)



**Figure 5.1:** The base architecture of ASP-WIDE.

required. Therefore, using Java in conjunction with the Spring framework<sup>8</sup>, as well as .NET Core<sup>9</sup> are considered. Both provide platform independence, a self-contained deployment mode and functionality to easily implement HTTP endpoints for web requests. Additionally, the interaction with an ASP system should be possible. For this task, a tool called DLVWrapper<sup>10</sup> provides optimal functionality. It does not only provide a layer of abstraction for the interaction with the ASP system, but also supports asynchronous program execution and callback functions (so called “ModelHandlers”) for found answer sets. At its current state, DLVWrapper supports interaction with the ASP systems DLV and DLV2. For ASP-WIDE the integration of DLV2 and Clingo is intended. Hence, adaptations to DLVWrapper have to be made in order to be able to interact with Clingo. Nevertheless, the use of this wrapper is reasonable and since it is entirely written in Java, also the Backend of ASP-WIDE is chosen to be realized in Java with the Spring framework. In particular, Spring Boot is used to obtain a self-contained deployment mode that already includes a web server instance and can be started on any operating system running Java. In terms of data storage, no database is used, because the ASP files of the current user are stored on the underlying file system that the Backend runs on. Considering data security, this is a suboptimal solution, but as this implementation is used as a proof of concept, adding authentication and data security aspects is not required.

With regards to the frontend, the Angular<sup>11</sup> framework is used. It supports the development of web applications for desktop and mobile devices. Moreover, it is a popular technology for realizing single page applications while having a big community and extensive documentation. JSON data is utilized for the communication with the Backend. Additionally, the feature-rich Monaco editor is adopted, which is known from Visual Studio Code<sup>12</sup>. It offers a well documented API, for instance for setting custom error/warning markers in the code or defining a syntax highlighting for a custom language.

After all technologies for realizing ASP-WIDE have been selected, a complete overview of the architecture, including the used technologies, is shown in Figure 5.2.

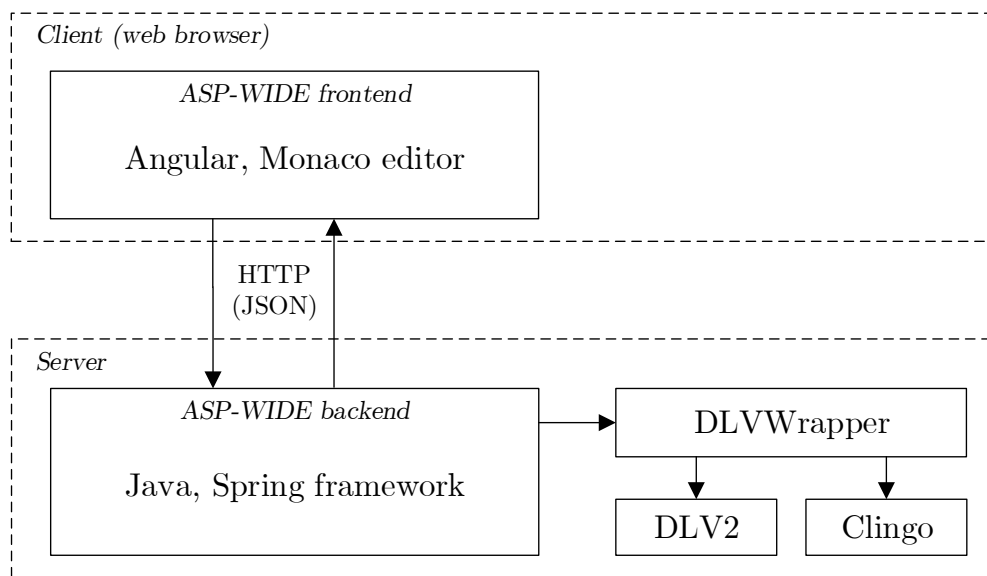
<sup>8</sup>Visit <https://spring.io/> for more information (accessed 2019-05-02).

<sup>9</sup>Visit <https://docs.microsoft.com/en-us/dotnet/core/> for more information (accessed 2019-05-02).

<sup>10</sup>Visit <http://www.dlvsystem.com/dlvwrapper/> for more information (accessed 2019-05-02).

<sup>11</sup>Visit <https://angular.io/> for more information (accessed 2019-05-02).

<sup>12</sup>Visit <https://microsoft.github.io/monaco-editor/index.html> for more information (accessed 2019-05-02).



**Figure 5.2:** The complete architecture of ASP-WIDE.

## 5.4 Backend

According to Section 5.2 the backend was realized with Java and the Spring framework. Additionally, DLVWrapper was used for communication with the ASP systems for executing answer set programs. In particular the solvers Clingo and DLV2 were integrated in the Backend. In the following subsections the most important parts of the implementation, namely

- the implementation of the annotation language parser,
- the usage of the ASP systems with DLVWrapper, and
- the testing engine

are presented. Although the implementation involved more aspects like the definition of the web endpoints for the HTTP requests, they are omitted in the following subsections.

### 5.4.1 Annotation language parser

The parser for the answer set program and the annotation language is a core component of the ASP-WIDE Backend. On the one hand, the parser is used for validating a user-defined answer set program according to the ASP-Core-2 input language format, on the other hand, also the annotation language of this thesis can be validated. When the developer is writing an answer set program in the Frontend, he receives immediate feedback in form of errors or warnings in the code. Most syntax errors can be provided by validating the code against a parser that models the ASP-Core-2 input language format. In case of ASP-WIDE, the parsing errors, which are syntax errors, are directly passed to the Frontend with specific details. The details include the lines of code in which the error appears, as well as the start- and end-index in the line. More complex validations, for example the check for rule safety, can be achieved by custom validators, which can also pass warnings to the Frontend.

The parser was implemented using JavaCC<sup>13</sup>, which is a parser generator for Java. It is a tool, that reads grammar specification and converts it to a Java program, which is able to recognize matches according to the grammar. Besides modeling the ASP-Core-2 input language format, the parser also models the annotation language of this thesis. Therefore, the parser is extended by the relevant grammar definitions for the annotations. Moreover, another tool called JJTree (which is included in JavaCC) was used, which is capable of building grammar trees. In particular, custom validators were implemented using the tree builder.

Listing 5.1 shows a part of the grammar specification for JavaCC. A statement according to ASP-Core-2 may either be a comment, an integrity constraint, a rule (including facts), a weak constraint or an optimization construct. Additionally, line 4 contains a non-terminal symbol `annotation()`, which enables the parser to also validate the annotations of this thesis. All possible annotations are included as successor of the non-terminal symbol `annotation()`. After generating the parser with JavaCC based on the grammar specification, each non-terminal symbol can be converted into a Java class if the JavaCC option `VISITOR = true` is configured. This option enables the use of the visitor pattern with JJTree and JavaCC, for example for the implementation of custom validators. The visitor pattern is also utilized for implementing the testing engine as explained in Section 4.4.1.

**Listing 5.1:** Grammar specification of “statement” according to ASP-Core-2 with annotation support.

```

1 void statement() #Statement :
2 {}
3 {
4     ( annotation() )
5     | ( comment() )
6     | ( < CONS > ( body() )? < DOT > ) // integrity constraints
7     | ( head() ( < CONS > ( body() )? )? < DOT > ) // rules (and facts)
8     | ( < WCONS > ( body() )? < DOT >
9         < SQUARE_OPEN > weightAtLevel() < SQUARE_CLOSE > ) // weak constraints
10    | ( optimize() < DOT > ) // optimization constructs
11 }
```

In Listing 5.2 a simple validation of an answer set program is performed. The code is passed as parameter to the function `checkCodeValidity`. If the program is not empty, the parser is used to validate the code. When a `ParseException` is detected, a new object of the type `CodeError` is created and afterwards returned by the function. Considering that this validation would always return the first syntax error and not validate the remaining code, a special mechanism is required for validating the whole program. Because of the fact, that the order of the rules in an answer set program is irrelevant for execution, each rule can be validated on its own. Although this is not an optimal solution, it is sufficient for this implementation. This way all errors of the program can be gathered in an aggregate of errors and returned to the Frontend as validation result.

**Listing 5.2:** Validating an answer set program with the generated ASP-Core-2 parser.

```

1 private List<CodeError> checkCodeValidity(String code) {
2     List<CodeError> errors = new ArrayList<CodeError>();
3
4     if(code == null || code.isEmpty()) {
5         CodeError ce = new CodeError();
```

<sup>13</sup>Visit <https://javacc.org/> for more information (accessed 2019-05-02).

```

6     ce.name = "ERROR: no code specified for validation.";
7     ce.description = "ERROR: no code specified for validation.";
8     errors.add(ce);
9     return errors;
10  }
11
12  try {
13      AspCore2wAnnotationsParser parser = new AspCore2wAnnotationsParser(
14          new ByteArrayInputStream( code.getBytes() ));
15      parser.program();
16  }
17  catch(ParseException ex) {
18      ex.printStackTrace();
19
20      CodeError ce = new CodeError();
21      ce.line = ex.currentToken.beginLine;
22      ce.startIndex = ex.currentToken.next.beginColumn;
23      ce.endIndex = ex.currentToken.next.endColumn;
24
25      String tmpErrMsg = ex.getMessage();
26      tmpErrMsg = tmpErrMsg.replaceFirst("line [0-9]+", "line "+ce.line);
27      tmpErrMsg = tmpErrMsg.replaceFirst("column [0-9]+", "column "+ce.startIndex);
28      ce.name = tmpErrMsg;
29
30      ce.description = ex.getStackTrace().toString();
31      errors.add(ce);
32  }
33  catch(TokenMgrError err) {
34      // error handling
35  }
36
37  return errors;
38 }

```

#### 5.4.2 DLVWrapper usage

The library DLVWrapper is used for communication between the ASP-WIDE Backend and the ASP systems. It was originally developed as a wrapper for the ASP system DLV, to make it usable from Java applications [Ric03]. Later on, it was extended to support interaction with DLV2. Since the goal of ASP-WIDE is to support DLV2, as well as Clingo for solving answer set programs, DLVWrapper needs further extensions.

As Clingo is a popular ASP system and has been used in answer set programming competitions, it supports the ASP-Core-2 input language format. For comparing different solving systems at the ASP competitions, also an output format had to be defined (to represent the found models). Hence, Clingo can be configured to print answer sets in the competition output format, which can be utilized the communication with DLVWrapper. The wrapper is modified to be able to call Clingo as a command-line tool with its specific arguments. After all, the default behavior of Clingo is to print only one answer set, which is why arguments have to be provided to print more (or all) answer sets. Furthermore, also the competition output format of the ASP system has to be requested with command line arguments. Accordingly, DLVWrapper has to provide system-specific arguments, based on the solver type which is used.

**Listing 5.3:** Customization of DLVWrapper according to the used ASP system.

```

1 protected void startRun() throws IOException, DLVInvocationException {
2     ... // some preparations
3
4     List<String> input = new ArrayList<String>();
5     input.add(DLVWrapper.getInstance().getPath());
6     try {
7         if (isDLV2){
8             addOption("--silent");
9         }
10        else if(isClingo) {
11            addOption("--outf=1"); // competition output format
12            addOption("--quiet=0,0"); // output configuration
13        }
14        else{ // DLV
15            addOption("-silent");
16        }
17    } catch (DLVInvocationException e) {
18        e.printStackTrace();
19    }
20    input.addAll(options);
21    input.addAll(inputProgram.GetFiles());
22
23    ... // starting the ASP system in a process
24 }

```

Listing 5.3 shows how the command line arguments are set, based on the underlying ASP system. The lines 7-15 demonstrate how the wrapper adds arguments (called “options”) to the invocation of the solving system. After the options are added, also the files to be executed are added to the invocation in line 21. Then the system is started in a new process.

A major benefit of DLVWrapper, besides the abstraction of the ASP system, is the asynchronous execution of answer set programs. Thus, callback handlers for found models, so called “ModelHandlers” must be implemented, to react appropriately if a model is found. Considering that Clingo prints found models on the command line, the output stream of the system has to be used for the recognition of found answer sets. As the system is configured to print the models in the competition output format, a parser had to be written for interpreting the output. Thus, a parser generated with JavaCC reads the output stream of Clingo and recognizes found models. Because of using the competition output format at this point, any solver supporting it, can be easily connected to DLVWrapper. Nevertheless, the foremost intention was the integration of Clingo. As a result, the extended DLVWrapper is capable of interaction with DLV, DLV2 and Clingo.

### 5.4.3 Testing engine realization

Another important aspect of the Backend is the realization of the testing engine. While the parser validates answer set programs with annotations and the DLVWrapper manages the interaction with a solving system, a mechanism is needed to interpret the annotations and perform the following steps:

1. Creation of an intermediate program for test execution
2. Execution of the intermediate program on the ASP system with appropriate parameters
3. Valuation of the test results

To create an intermediate program for a test case, the testing engine needs access to the

properties of the test specification (ASP rules, input, assertions, etc.). For this access, the parser of Subsection 5.4.1 is utilized. As already stated, the generated parser supports the visitor pattern, which comes with `JJTree` and can be used to build a virtual model of the test cases and their properties. An instance of the “`TestSuite`” class includes all blocks, rules and tests of an answer set program, as shown in Listing 5.4. Since blocks and rules can be reused across multiple test cases, they are not included in the “`Test`”-objects of the list in line 4. On the contrary, inputs in form of answer set rules are part of the test case specification and therefore included in the “`Test`”-objects.

**Listing 5.4:** The virtual (in-memory) model of a program using annotations for unit testing.

```

1 public class TestSuite {
2     private HashMap<String, Block> blocks;
3     private HashMap<String, String> rules;
4     private ArrayList<Test> tests;
5
6     public TestSuite() {
7         this.blocks = new HashMap<String, Block>();
8         this.rules = new HashMap<String, String>();
9         this.tests = new ArrayList<Test>();
10    }
11
12    // further function definitions ...
13 }
```

By using this virtual model, an intermediate program can be generated according to the test cases and their assertions. Rules and blocks are selected according to the test case specification and translation is performed according to the assertion. Afterwards the final result is the intermediate program, that needs to be executed on an ASP system. In particular, it needs to be executed with specific parameters to search for a fixed number of answer sets according to the definitions in Subsection 4.4.2. Listing 5.5 shows the implementation of the `@trueInAtMost` assertion. While lines 8-14 demonstrate how an integrity constraint is added to the intermediate program, lines 19-20 contains the execution of the intermediate program with the desired number of evaluated models. In the background `DLVWrapper` is used to run the ASP system with appropriate arguments. Finally, line 23 checks if the assertion passed or failed, based on the number of resulting answer sets.

**Listing 5.5:** The implementation of the `@trueInAtMost` assertion.

```

1 private AssertionResult checkAssertion(String code,
2     AssertTrueInAtMost as, SolverType st) {
3     AssertionResult ar = new AssertionResult();
4     ar.setName(as.getClass().getSimpleName());
5     ar.setExecutedCode(code);
6
7     String[] atoms = as.getAtoms().split("\\.");
8
9     StringBuilder sb = new StringBuilder(code);
10    for(String atom: atoms) {
11        sb.append(":- not ");
12        sb.append(atom.trim());
13        sb.append(".\n");
14    }
15    ar.setExecutedCode(sb.toString());
16
17    ExecutableFile testFile =
```

```
18     new ExecutableFile("checkAssertionTrueAtMost", ar.getExecutedCode(), "");
19     testFile.setSolverType(st);
20     ExecutionResult er = this.executionLogic
21         .executeCode(testFile, (as.getAssertCount()+1));
22     ar.setExecutionOutput(er.getResult());
23
24     if(er.getModels() != null && er.getModels().size() <= as.getAssertCount()) {
25         ar.setSucceeded(true);
26     }
27     return ar;
28 }
```

## 5.5 Frontend

For realizing the Frontend of ASP-WIDE, the Angular framework was used. Angular is a framework for building web-based applications for desktop and mobile devices. Moreover, it utilizes TypeScript as a typed superset of JavaScript for building applications. When building complex applications, also a variety of packages can be included in the project. These can be packages including specific components or styles. While the Frontend incorporates several important aspects, the following subsections describe the most important components with respect to this thesis, which are:

- Code editing in ASP-WIDE
- Syntax highlighting
- Executing/testing answer set programs

Finally, also the user interface showing all regions and functionality is presented.

### 5.5.1 Code editor

One of the most essential components when building a development environment is the code editor. Since the Monaco editor is used in the editor Visual Studio Code and developed as an open source project, it is commonly known among developers. Thanks to an existing package for angular projects (called “ngx-monaco-editor”), it can be easily integrated into ASP-WIDE by using the node package manager (npm). Additionally, the Monaco editor comes with several built-in features as auto completion, syntax colorization and an extensive API for further configurations.

After adding Monaco to the angular project with the command

```
npm install ngx-monaco-editor --save
```

and some basic configurations, it can be used in an angular component. In particular, the editor is added in the HTML template of the component “IdeComponent” with the code shown in Listing 5.6. The editor is added to the template file in line 9. Furthermore, the options in line 10 can be used to define configuration options for the editor, for instance the language that is currently used with the editor. Line 11 assigns a model, namely the content of the opened file, and the event handler in line 12 handles change events when content of the file is modified. A dedicated initialization function, which is called on the initialization of the Monaco editor, can be specified as shown in line 13.



**Listing 5.6:** Integration of the Monaco editor in ASP-WIDE.

```

1 ...
2 <ng-container *ngFor="let item of this.tabItems">
3   <mat-tab>
4     <ng-template mat-tab-label class="ide-tab-label">
5       <i class="fas fa-file"></i>
6       {{item.name}}
7       <i (click)="event_onClickCloseTab(item)" class="fas fa-times"></i>
8     </ng-template>
9     <ngx-monaco-editor class="ide-monaco-editor"
10      [options]="editorOptions"
11      [(ngModel)]=item.content"
12      (ngModelChange)="this.event_onContentModified($event)"
13      (onInit)="onInit($event)">
14   </ngx-monaco-editor>
15 </mat-tab>
16 </ng-container>
17 ...

```

### 5.5.2 Syntax highlighting

Due to the usage of the Monaco editor in the Frontend, also a custom syntax colorization/highlighting for the annotation language can be realized. So called “colorizers” can be implemented with Monarch<sup>14</sup>, which is integrated into the editor. With Monarch declarative JSON-based syntax highlighters can be implemented and easily used in the editor.

Since Monaco does not come with a syntax highlighter for answer set programming (or Prolog) a custom syntax highlighter had to be written for the syntax of ASP, as well as the annotation language. Therefore, a pre-implemented highlighter was modified to achieve the required syntax colorization.

**Listing 5.7:** Parts of the colorizer written for ASP and the annotation language.

```

1 {
2   keywords: [
3     '#count', '#sum', '#avg', 'not'
4   ],
5   ...
6   tokenizer: {
7     root: [
8       [/@\s*[a-zA-Z_\$][\w\$\$]*/, 'annotation'],
9       ...
10    ],
11    comment: [
12      [/[^\s*/]+/, 'comment'],
13      [/\s*/, 'comment', '@push'], // nested comment
14      [/\s*/, 'comment', '@pop'],
15      [/[^\s*/]+/, 'comment']
16    ],
17    ...
18  }
19 }

```

Listing 5.7 shows some parts of the JSON-based definition of the colorizer. In the lines 2-4 some keywords of the language are defined. Additionally the lines 7-18 contain so called

<sup>14</sup>Visit <https://microsoft.github.io/monaco-editor/monarch.html> for more information on Monarch (accessed 2019-05-09).

“tokenizers”, which define how the lexical analysis takes place and how the input is divided into tokens. For instance, line 8 defines what an annotation consists of by using a regular expression.

After implementing the custom syntax highlighter by using the Monarch side-by-side editor<sup>15</sup>, it can be added to the Monaco editor instance of ASP-WIDE. This happens by specifying the used language in the configuration, which is loaded on editor start-up. An appropriate place for putting the initialization logic is the module “AppModule” of the Angular project.

**Listing 5.8:** Setting the language “aspCore2Lang” on Monaco initialization.

```

1 export function myMonacoOnLoad(){
2   (<any>window).monaco.languages.register({ id: 'aspCore2Lang' });
3   (<any>window).monaco.languages.setMonarchTokensProvider('aspCore2Lang', aspCore2Lang);
4 }
5
6 const monacoConfig: NgxMonacoEditorConfig = {
7   defaultOptions: { scrollBeyondLastLine: false },
8   onMonacoLoad: myMonacoOnLoad
9 };
10
11 @NgModule({
12   declarations: [
13     AppComponent,
14     IdeComponent,
15     ...
16   ],
17   imports: [
18     ...
19     MonacoEditorModule.forRoot(monacoConfig),
20   ],
21   ...
22 })
23 export class AppModule { }
```

Listing 5.8 contains the initialization logic of the Monaco editor. The function “myMonacoOnLoad” in lines 1-4 enables the Monaco editor to use the custom syntax highlighter called “aspCore2Lang”. Afterwards this function needs to be called when initializing the editor instance, which is achieved with the configuration “NgxMonacoEditorConfig” in lines 6-9. Finally, the configuration is passed to the corresponding module in line 19.

### 5.5.3 Executing and testing answer set programs

Of course the Frontend of ASP-WIDE needs to communicate with the Backend in order to be able to execute answer set programs and tests. As already stated, this communication is based on sending JSON data over HTTP requests. Thanks to the “HttpClient”, that came with Angular version 4.3, JSON requests can be implemented easily. Nevertheless, before sending HTTP requests, the data structure needs to be defined. In order to communicate flawlessly, both Front- and Backend share the same data structures.

<sup>15</sup>Check <https://microsoft.github.io/monaco-editor/monarch.html> for the Monarch side-by-side editor (accessed 2019-05-10).

**Listing 5.9:** Base data structures for executing an answer set program.

```

1 export interface IAbstractLogicItem{
2   name: string,
3   path: string,
4   uuid: string,
5   hash: string
6 }
7
8 export interface ILogicFile extends IAbstractLogicItem{
9   content: string
10 }
11
12 export interface IExecutableFile extends ILogicFile {
13   solverType: string
14 }

```

The data structures used by the Frontend for sending HTTP requests that execute an answer set program are shown in Listing 5.9. The interface “IAbstractLogicItem” represents an object that can be either a file or a folder, as ASP-WIDE also supports the management of folders. These logic items may have a name, a path, a UUID and a hash, which can be used for checking file integrity. A derived type “ILogicFile” contains the content of the file to execute, while “IExecutableFile” also specifies the desired solver (DLV2 or Clingo).

By the use of these data structures, a HTTP request can be composed in order to retrieve the execution results of the specified program. After the specified file is executed on the desired solver, a HTTP response according to the type “IExecutionResult” of Listing 5.10 is sent by the server. It is derived from “IValidationResult”, which holds validation errors or warnings if the program could not be executed. Otherwise “IExecutionResult” holds the resulting answer sets as simple string, which is sufficient for ASP-WIDE.

**Listing 5.10:** Base data structures for receiving execution results of an answer set program.

```

1 export interface IValidatableFile extends ILogicFile{}
2
3 export interface IValidationResult extends IValidatableFile{
4   errors: Array<ICodeError>,
5   warnings: Array<ICodeWarning>
6 }
7
8 export interface IExecutionResult extends IValidationResult{
9   result: string
10 }

```

After having specified both sending and receiving types for a HTTP request, the web-service call can be implemented inside an injectable service in the Angular project. While the use of a generator for creating client-side SDKs for calling the API (for example Swagger Codegen<sup>16</sup> or NSwag<sup>17</sup>) is possible, it is not required for ASP-WIDE. Due to the availability of the “HttpClientModule” and the explicit definition of HTTP request and response data structures, a client SDK for the server-side API endpoints can be defined easily as shown in Listing 5.11. Effectively the lines 20-26 contain the full implementation of the web-service call to the specified API endpoint using the request and response data structures. As this request can also fail, all responses that contain error codes are handled separately as shown in line 24.

<sup>16</sup>Visit <https://swagger.io/tools/swagger-codegen/> for more information on Swagger (accessed 2019-05-10).

<sup>17</sup>Visit <https://github.com/RicoSuter/NSwag> for more information on NSwag (accessed 2019-05-10).

**Listing 5.11:** Implementation of the API call for executing an answer set program.

```

1 @Injectable({
2   providedIn: 'root'
3 })
4 export class RestClientService {
5
6   private baseUrl: string;
7   private executeUrl = '/execution/execute';
8   ...
9
10  private httpOptions: any = {
11    headers: new HttpHeaders({
12      'Content-Type': 'application/json'
13    })
14  };
15
16  constructor(private http: HttpClient) {
17    this.baseUrl = environment.restBaseUrl;
18  }
19
20  public postExecute(file: IExecutableFile) : Observable<IExecutionResult> {
21    return this.http
22      .post<IExecutionResult>(this.baseUrl+this.executeUrl, file, this.httpOptions)
23      .pipe(
24        catchError(this.handleError<any>('postExecute'))
25      );
26  }
27  ...
28 }

```

## 5.6 The ASP-WIDE user interface

The development environment ASP-WIDE represents the outcome of the implementation phase of this thesis. It incorporates the annotation language, as well as the execution mechanism and several other features required for developing answer set programs. These include syntax highlighting (for ASP code and annotations), file management and an execution/testing environment. It is developed as a standalone tool, which can be started on any machine running Java. Consisting of a Frontend and a Backend, the most essential part for ASP developers is the user interface. Therefore, the user interface is presented and explained hereinafter.

Figure 5.3 shows a screenshot of the user interface of ASP-WIDE. The areas of the interface are divided into four parts, which are commonly found in most IDEs, namely

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

The toolbar features usual menus for handling files and settings. Moreover, it also contains buttons to execute tests, run programs, stop program execution and select the desired run configuration (from left to right). The ability to manage different run configurations is a feature, which was added during the development, but was not intended originally. Run configurations are accessible by selecting multiple files in the explorer, doing a right-click and selecting "Create Run Configuration..." in the context menu or by clicking on the drop-down

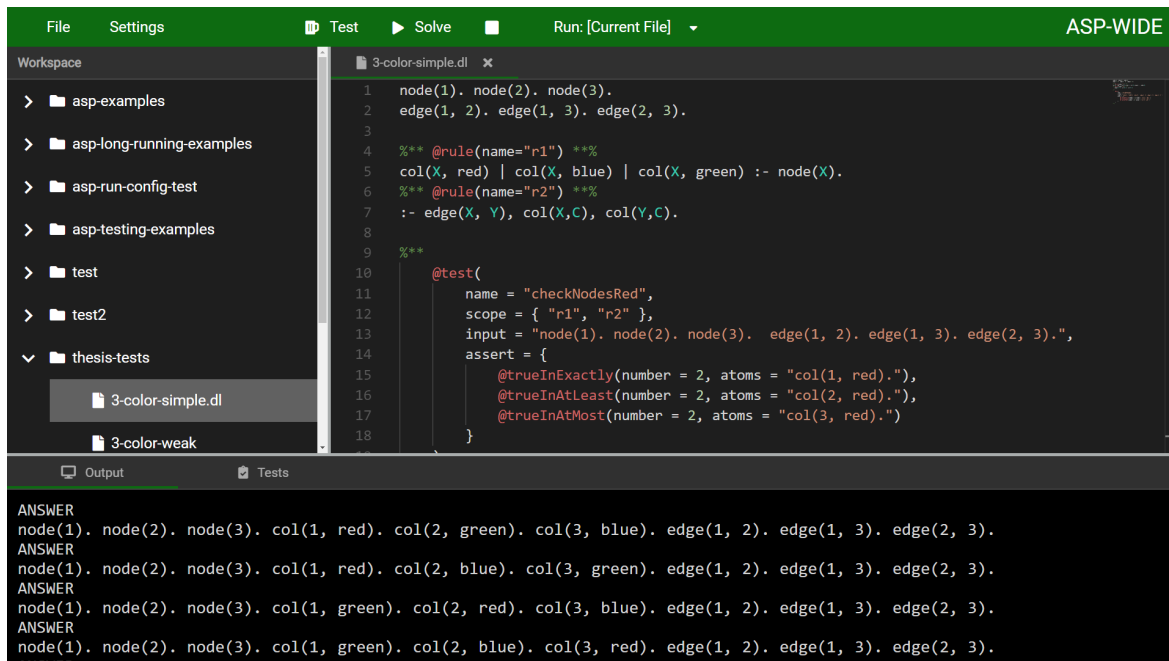


Figure 5.3: The user interface of ASP-WIDE.

menu in the toolbar and selecting “Open Run Configurations...”. With the use of this feature, multiple files can be executed at once. Furthermore, different ASP systems can be selected (if available on the system) and the number of evaluated answer sets can be configured.

On the left, the workspace allows the developer to add folders/files and organize ASP files in a project-like manner. All necessary operations can be performed by using the context menu, which appears after a right click, or via the “File” menu in the toolbar. Additionally, files can be opened in the code editor with a double click. The code editor is able to handle multiple opened files with the tab control on the top. Any changes to the files are automatically saved. Subsequently, it is also validated and errors are shown automatically afterwards.

If the file(s) can be validated without errors, the answer sets can be evaluated by clicking on the button “Solve” or tests can be executed by clicking “Test” in the toolbar. The results are presented in the bottom area in the corresponding tabs. While answer sets are shown in a console-like manner, test results are presented in a more appealing way, as demonstrated in Section 6.1.

## Chapter 6

# Use Case and Evaluation

While Chapter 4 described a new annotation-based testing language and mechanism for ASP, Chapter 5 showed its realization as part of a development environment for ASP, namely ASP-WIDE. This Chapter aims to evaluate the possibilities when using the annotation language as part of ASP-WIDE. Therefore, the usage of the testing language (in ASP-WIDE) is presented with a use case first. This demonstrates the level of expressiveness and usefulness, that can be reached with the new annotation language. Afterwards, the annotation language is compared to the set of features provided by related works. Finally, a feature and performance comparison of different approaches, respectively related works, is a major result of this Chapter.

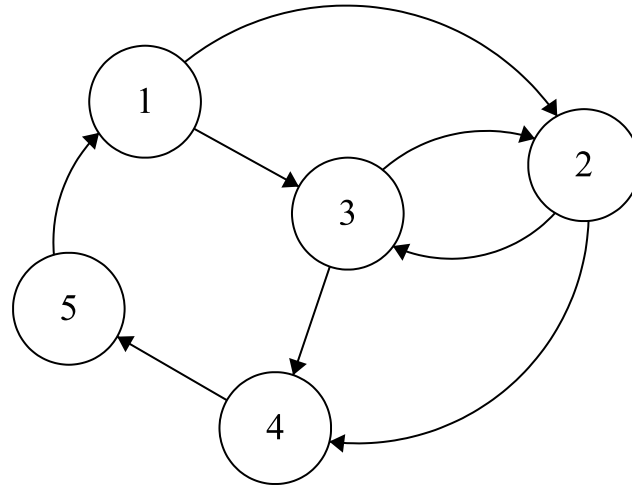
### 6.1 Use Case

The usage of the annotation language is demonstrated alongside a search problem, which is well-known in the context of answer set programming and was already described in Section 3.1, namely the Hamiltonian path problem. After a specific problem instance is described hereinafter, an answer set program (which is capable of solving the problem) and the answer sets of the problem instance are presented. Based on this information, unit tests are formulated by using the annotation language of this thesis. Additionally, the execution of the unit tests in ASP-WIDE including the test results complete the example.

#### 6.1.1 Hamiltonian path problem instance

The well-known Hamiltonian path problem is a search problem, that can be solved with answer set programming. As already mentioned in Section 3.1, the problem can be described as follows: Given a finite directed graph  $G = (V, E)$  and a node  $a$ , is there a path in  $G$  starting from  $a$  and passing through each node in  $V$  exactly once?

Considering the graph in Figure 6.1, which contains five nodes and eight edges, a human can easily identify two Hamiltonian paths starting from node 1, namely  $(1, 2, 3, 4, 5)$  and  $(1, 3, 2, 4, 5)$ . However, an ASP system is able to evaluate the solutions of the search problem much faster, even with more complex problem instances. Therefore, an answer set program modeling this problem instance is shown in Listing 6.1. While the lines 1-3 represent the nodes and edges, line 4 specifies the starting node, namely node 1. Afterwards, the set of all possible solutions is generated with line 7, by expressing that each edge does either belong to a Hamiltonian path or not. Then all possible paths, respectively nodes, are traversed (line 13), while starting with node 1 (line 10). Of course, in a valid Hamiltonian path all nodes must



**Figure 6.1:** A finite directed graph for the Hamiltonian path problem.

be reached, which is assured with the integrity constraint in line 16. Moreover, the starting node cannot be the target of any edge (line 18). The integrity constraints in the lines 20-21 guarantee, that each node has at most one incoming and one outgoing edge. This program models the Hamiltonian path problem instance of Figure 6.1 and provides the already stated Hamiltonian paths (1, 2, 3, 4, 5) and (1, 3, 2, 4, 5) when executed on an ASP system.

**Listing 6.1:** Answer set program modeling an instance of the Hamiltonian path problem.

```

1 vtx(1). vtx(2). vtx(3). vtx(4). vtx(5).
2 edge(1, 2). edge(2, 3). edge(3, 4). edge(4, 5).
3 edge(3, 2). edge(1, 3). edge(2, 4). edge(5, 1).
4 start(1).
5
6 % each edge either belongs to hamiltonian path or not
7 inPath(X, Y) | outPath(X, Y) :- edge(X, Y).
8
9 % start from the specified node
10 reached(X) :- start(X).
11
12 % traverse the path
13 reached(X) :- inPath(Y, X), reached(Y).
14
15 % all nodes must be reached
16 :- vtx(X), not reached(X).
17 % the start node cannot be the target node of an edge
18 :- start(X), inPath(_, X).
19 % each vertex in the path must have at most one incoming and one outgoing edge
20 :- vtx(X), #count{Y : inPath(X, Y)} > 1.
21 :- vtx(X), #count{Y : inPath(Y, X)} > 1.

```

### 6.1.2 Defining test cases

Before defining test cases, a selection of the rules to be tested must happen, since Listing 6.1 contains instance specific rules (input knowledge) in the lines 1-4, as well as problem specific rules in the lines 7-21. When defining unit tests, the aim might be to test only problem specific rules in order to be able to provide different input knowledge for more complex test cases or corner cases. Therefore, the rules under test considering the program in Listing 6.1 consist of the seven answer set rules in the lines 7-21. All developments hereinafter are performed in the ASP-WIDE development environment.

The first step of defining test cases for this example is rule naming and the definition of a block. Hence, the original program is enriched with annotations to assign names to rules and assign all of them to a single block called “hamiltonianRules”, as shown in Listing 6.2. For simplicity the input knowledge (facts) are removed, since they are going to be provided in the test case definitions. Nonetheless, this is not mandatory. Afterwards the developer must think of appropriate test cases in order to validate the problem specific rules under test.

**Listing 6.2:** Block definition and rule naming for defining test cases.

```

1 %** @block(name="hamiltonianRules") **%
2
3 %** @rule(name="r1", block="hamiltonianRules") **%
4 inPath(X, Y) | outPath(X, Y) :- edge(X, Y).
5
6 %** @rule(name="r2", block="hamiltonianRules") **%
7 reached(X) :- start(X).
8
9 %** @rule(name="r3", block="hamiltonianRules") **%
10 reached(X) :- inPath(Y, X), reached(Y).
11
12 %** @rule(name="r4", block="hamiltonianRules") **%
13 :- vtx(X), not reached(X).
14 %** @rule(name="r5", block="hamiltonianRules") **%
15 :- start(X), inPath(_, X).
16 %** @rule(name="r6", block="hamiltonianRules") **%
17 :- vtx(X), #count{Y : inPath(X, Y)} > 1.
18 %** @rule(name="r7", block="hamiltonianRules") **%
19 :- vtx(X), #count{Y : inPath(Y, X)} > 1.

```

The first test, that is realized must assure that each node is contained in the path, in order to be a valid Hamiltonian path. The atom `@reached(...)` can be utilized for this test. Thus, the test case in Listing 6.3 is appended to the program in Listing 6.2. Also input knowledge is provided with the property “input”, which models the same problem instance as presented before. To assure that each node is reached, the test case asserts that all nodes are reached in all valid answer sets (which are Hamiltonian paths) by the use of `@trueInAll(...)`.

**Listing 6.3:** Test case that checks if all nodes are reached.

```

1 %**
2   @test(name = "allNodesReached",
3     scope = { "hamiltonianRules" },
4     input = "vtx(1). vtx(2). vtx(3). vtx(4). vtx(5).
5       edge(1, 2). edge(2, 3). edge(3, 4). edge(4, 5).
6       edge(3, 2). edge(1, 3). edge(2, 4). edge(5, 1). start(1).",
7     assert = {
8       @trueInAll(atoms = "reached(1). reached(2).
9         reached(3). reached(4). reached(5).")

```



```

10     }
11     )
12 **%

```

The second test to be implemented should assure, that the starting node is not targeted by any edge, as this is a requirement of the Hamiltonian path. Accordingly as second test case is appended to the program, which is shown in Listing 6.4. In particular the assertion `@falseInAll(...)` in line 8 expresses, that all permutations of ground atoms targeting the starting node 1 are not part of answer sets (Hamiltonian paths).

**Listing 6.4:** Test case that assures that the starting node is not targeted.

```

1 %**
2   @test(name = "startingNodeNotTargeted",
3         scope = { "hamiltonianRules" },
4         input = "vtx(1). vtx(2). vtx(3). vtx(4). vtx(5).
5                 edge(1, 2). edge(2, 3). edge(3, 4). edge(4, 5).
6                 edge(3, 2). edge(1, 3). edge(2, 4). edge(5, 1). start(1).",
7         assert = {
8             @falseInAll(atoms = "inPath(2, 1). inPath(3, 1).
9                             inPath(4, 1). inPath(5, 1).")
10        }
11    )
12 **%

```

A third test case to be implemented should check, whether the produced answer sets correspond to the the real answer sets of the problem instance. Since they were already stated, a test case specification as shown in Listing 6.5 can be appended to the program as a final test. Here the assertions `@trueInExactly(number = 1, ...)` require the answer sets to exactly contain the valid answer sets of the problem instance. Nonetheless, this test case does not guarantee that only these two answer sets exist.

**Listing 6.5:** Test case to validate the answer sets of the program.

```

1 %**
2   @test(name = "checkAnswerSets",
3         scope = { "hamiltonianRules" },
4         input = "vtx(1). vtx(2). vtx(3). vtx(4). vtx(5).
5                 edge(1, 2). edge(2, 3). edge(3, 4). edge(4, 5).
6                 edge(3, 2). edge(1, 3). edge(2, 4). edge(5, 1). start(1).",
7         assert = {
8             @trueInExactly(number = 1, atoms = "inPath(1, 3). inPath(3, 2).
9                             inPath(2, 4). inPath(4, 5)."),
10            @trueInExactly(number = 1, atoms = "inPath(1, 2). inPath(2, 3).
11                             inPath(3, 4). inPath(4, 5).")
12        }
13    )
14 **%

```

Since all developments of this Subsection are performed in the ASP-WIDE development environment, syntax errors are instantly reported by the IDE. They need to be corrected to be compliant with the ASP-Core-2 input language format and the syntax of the annotation language. Otherwise, the execution of the program, respectively the test cases, is not possible. Figure 6.2 shows a screenshot of the development environment and the complete program including the test cases.

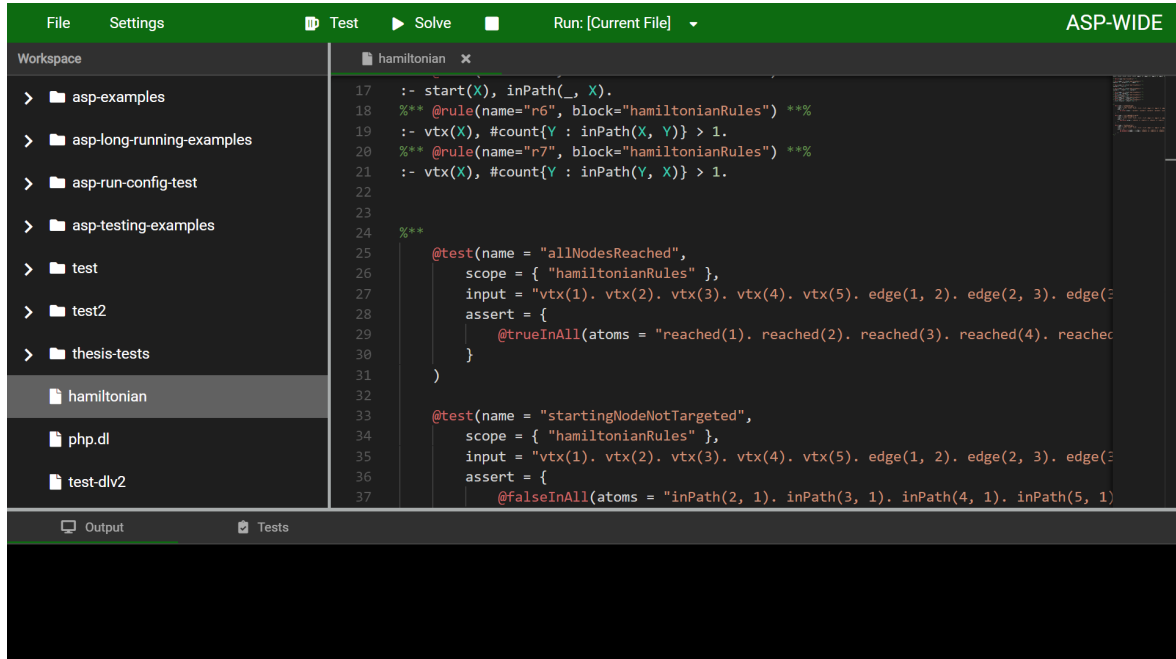


Figure 6.2: A screenshot of ASP-WIDE containing the program including test cases.

### 6.1.3 Executing test cases

The test cases defined in Subsection 6.1.2 can be executed using ASP-WIDE. The execution can be started with the button “Test” in the toolbar on the top. Afterwards the results of the unit tests are presented in the bottom area in the tab “Tests”. The screenshot in Figure 6.3 shows the outcomes of the test cases. Also test execution details, containing the intermediate program and the output of its execution, are provided alongside the test results. According to the screenshot, all tests succeed, which means that all tested aspects of the program behave as expected.

For demonstrating a failing test, a typical implementation mistake is introduced in the answer set program. For instance a developer might forget to add a rule modeling, that the starting node is not targeted by any edge. Therefore, the rule

$$\text{:- start}(X), \text{inPath}(\_, X).$$

is removed, which guarantees that the starting node is not targeted by any edge. This should result in answer sets, which are not valid Hamiltonian paths for the given problem instance. After executing the tests again, the outcomes of two tests change. As expected, the tests “startingNodeNotTargeted” and “checkAnswerSets” fail, because their assertions are not fulfilled by the answer sets of the modified program. The screenshot in Figure 6.4 shows all test outcomes, as well as the test execution details for the test “startingNodeNotTargeted”. These details clearly show, that an answer set exists, in which the starting node is targeted by an edge (it contains the ground atom `inPath(5, 1)`).

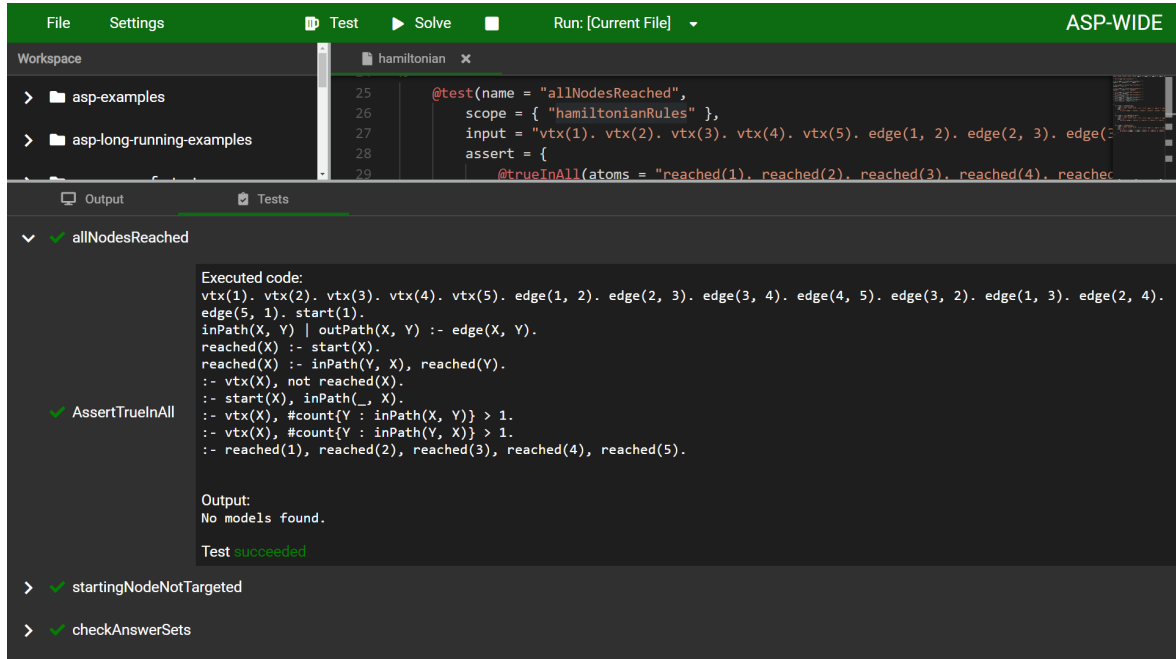


Figure 6.3: A screenshot of ASP-WIDE containing the results of the test cases.

## 6.2 Evaluation

After showing the usage and demonstrating the expressiveness of the annotation language in ASP-WIDE, an evaluation is going to be the content of this section. Instead of evaluating ASP-WIDE, this evaluation focuses solely on the annotation language, its functionality and execution mechanism, as ASP-WIDE was not designed as a feature rich development environment for competing with similar IDEs. ASP-WIDE was intended as a proof of concept for showing the feasibility of implementing the annotation language and execution mechanism of this thesis. Therefore, the following contents will first present a functional evaluation of the annotation language, compared to competing approaches from ASPIDE and LANA. Afterwards, an evaluation of the execution mechanism, compared to ASPIDE is performed.

### 6.2.1 Language features

The approaches considered for comparison with the annotation language of this thesis are ASPIDE’s test specification language and LANA. While the ASPIDE approach does not purely rely on an annotation-based language, it incorporates extensive testing capabilities. Consequently, all approaches enable developers to specify test cases and express specific assertions, that must be fulfilled to obtain a passing test.

Table 6.1 contains a comparative table showing the assertion types available in the evaluated approaches. It shows which type of assertion can be realized with the language capabilities of the given approach. Since the annotation language of this thesis is contained in ASP-WIDE, it is represented by the correspondingly named column. According to the resulting feature matrix, ASP-WIDE and ASPIDE share mostly the same set of features, except the annotation type “has no answer set”, which is supported by ASP-WIDE. Furthermore, LANA lacks several types, especially assertions for fulfilled constraints, but supports

```

File Settings Test Solve Run: [Current File] ASP-WIDE
Workspace
  > asp-examples
  > asp-long-running-examples
    hamiltonian
      32 scope = { "hamiltonianRules" },
      33 input = "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). start(1).
      34
      35 assert = {
      @FalseInAll(atoms = "inPath(2, 1). inPath(3, 1). inPath(4, 1). inPath(5, 1)

Output Tests
  > allNodesReached
  > startingNodeNotTargeted
    Executed code:
    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). start(1).
    inPath(X, Y) | outPath(X, Y) :- edge(X, Y).
    reached(X) :- start(X).
    reached(X) :- inPath(Y, X), reached(Y).
    :- vtx(X), not reached(X).
    :- vtx(X), #count(Y : inPath(X, Y)) > 1.
    :- vtx(X), #count(Y : inPath(Y, X)) > 1.
    :- not inPath(2, 1), not inPath(3, 1), not inPath(4, 1), not inPath(5, 1).

    Output:
    ANSWER
    outPath(1, 2). outPath(2, 3). outPath(3, 4). edge(1, 2). edge(2, 3). edge(3, 4). edge(4, 5). edge(3, 2). edge(1, 3).
    edge(2, 4). edge(5, 1). reached(1). reached(2). reached(3). reached(4). reached(5). start(1). inPath(4, 5). inPath(3, 2).
    inPath(1, 3). inPath(2, 4). inPath(5, 1). vtx(1). vtx(2). vtx(3). vtx(4). vtx(5).

    Test failed

  > checkAnswerSets

```

**Figure 6.4:** A screenshot of ASP-WIDE containing the test results of the modified program.

“has answer set” on the other hand. In conclusion ASP-WIDE supports the largest number of assertion types among the evaluated approaches.

Besides several assertion types, all approaches also provide additional features for building test cases. For instance adding additional input to the test case in form of ASP rules or files containing answer set programs. Therefore, Table 6.2 also compares all general features, that can be identified for defining test cases in all approaches. As a result, ASPIDE still provides the most configuration possibilities for testing in ASP, but lacks the concept of combining rules to blocks. Furthermore, several features are not supported by LANA, while some of them are supported by ASP-WIDE or can be constructed using its annotation language. In ASP-WIDE a whole program in form of an ASP file can be tested by specifying it as input (instead of selecting rules or blocks) with the property “inputFiles” of the `@test(...)` annotation. Although it is parsed correctly by ASP-WIDE, the implementation of this property is not complete. With regards to setting solver options, ASP-WIDE is not capable of setting any arguments for the solver at the moment. Nevertheless, this is a minor disadvantage and can be added easily during future enhancements. Despite the fact, that ASPIDE theoretically allows the selection of the ASP solver, it has been originally developed for DLV. It is unclear to which extent the execution of tests on different solvers, for instance Clingo, is doable.

### 6.2.2 Test execution mechanism

With regards to the evaluation of the test execution mechanism, a comparison with LANA, as well as ASPIDE is intended. Unfortunately LANA does not provide detailed information about the inner workings during test execution. Therefore, the evaluation of the test execution mechanism consists of a comparison between the approaches of ASP-WIDE and ASPIDE. In particular, the execution times of both will be compared, while executing a simple test on an answer set program.

Assertion	<i>ASP-WIDE</i>	<i>ASPIDE</i>	<i>LANA</i>
true in all	✓	✓	✓
true in at least $n$	✓	✓	✓
true in at most $n$	✓	✓	✓
true in exactly $n$	✓	✓	
false in all	✓	✓	✓
false in at least $n$	✓	✓	✓
false in at most $n$	✓	✓	✓
false in exactly $n$	✓	✓	
constraint for all	✓	✓	
constraint for at least $n$	✓	✓	
constraint for at most $n$	✓	✓	
constraint for exactly $n$	✓	✓	
best model cost	✓	✓	
has no answer set	✓		✓
has answer set			✓

**Table 6.1:** Comparison of assertion possibilities in different test specification approaches.

For this scenario the program shown in Listing 6.6 will be used. This program as is requires an execution time of approximately 6 seconds in ASPIDE for computing the resulting 262144 answer sets. All tests are performed on a system using an Intel Core i7-4710MQ CPU with 4 cores, 8 threads and a base frequency of 2.50 GHz (turbo frequency of 3.50 GHz).

**Listing 6.6:** Answer set program for comparing test execution durations.

```

1 b(1). b(2). b(3). b(4). b(5). b(6). b(7). b(8). b(9). b(10).
2 b(11). b(12). b(13). b(14). b(15). b(16). b(17). b(18).
3
4 a(X) | c(X) :- b(X).
```

Since the execution time of a test case, instead of the evaluation time for all answer sets, is relevant, a simple test is defined, which checks if the ground atom `b(1)` is true in all answer sets. This should be true, as it is provided as a fact. Having a test case definition in both development environments, the test is executed ten times while the execution durations are noted in Table 6.3. The execution times for ASPIDE are measured by the IDE itself and presented in the test results window. For ASP-WIDE the test execution times correspond to the HTTP round trips (Client-Server-Client), which can be easily displayed in modern browsers.

According to Febraro et al. [Feb+11], ASPIDE verifies tests by calling an ASP solver and checking its output, which is a naive test execution approach. Based on this approach the test verification takes at least as long as the execution of the program, respectively

Feature	<i>ASP-WIDE</i>	<i>ASPIDE</i>	<i>LANA</i>
testing rules	✓	✓	✓
testing blocks	✓		✓
testing split program		✓	
testing whole program	✓	✓	
input rules	✓	✓	✓
input files	~	✓	
excluding rules		✓	
excluding files		✓	
selecting solver	✓	~	✓
setting solver options		✓	✓
filtering output predicates		✓	

**Table 6.2:** Comparison of test case definition features in different approaches.

	<i>ASP-WIDE</i>	<i>ASPIDE</i>
1. Execution	0.109	6.834
2. Execution	0.129	5.865
3. Execution	0.107	4.500
4. Execution	0.099	7.103
5. Execution	0.104	6.865
6. Execution	0.083	6.411
7. Execution	0.089	7.331
8. Execution	0.072	6.693
9. Execution	0.068	5.805
10. Execution	0.079	4.155

**Table 6.3:** The execution durations of the test case in seconds with ASP-WIDE and ASPIDE.

the rules under test. This can be also observed in Table 6.3. With an average test execution duration of 6.156 seconds, ASPIDE is significantly slower than ASP-WIDE, having an average of 0.093 seconds. The much faster execution of ASP-WIDE is the result of translating the program under test to an intermediate program. Based on this intermediate program, the ASP system, respectively the solver, can immediately recognize a possible contradiction and

therefore conclude that it is unsatisfiable (it has no answer sets) in this case.

While this example gains its advantage against a naive test execution approach through the construction of a contradiction, it also demonstrates the fundamental advantage against other approaches. Also if the intermediate program does not lead to a contradiction, the ASP-WIDE approach will be faster than ASPIDE, because it needs to evaluate less answer sets, thanks to the intermediate programs (as explained in Section 4.4). Thus, the execution of the assertion `@bestModelCost` will not improve, since it does not rely on an intermediate program translation.

## Chapter 7

# Conclusion and prospects

This thesis proposes a new approach for annotation-based testing in answer set programming. Besides a new annotation language, the test execution mechanism constitutes a major improvement compared to previous approaches. This conclusion is validated after comparing test execution times with previously known execution mechanisms. Furthermore, the implementation of the execution mechanism confirms that the validation of test case assertions can be performed by rewriting the original answer set program and creating an intermediate program, which is used for test execution. Hence, an answer set solving system is capable of executing the test, which comes with all peculiarities of answer set programming. Some of them are the possibility to evaluate a fixed number of models or the immediate determination of contradictions, even in complex logical formulations.

Related works in the field of testing answer set programs appear to be rare, but also mature. Especially, when dealing with formal definitions of terms known from the field of unit testing, which are here considered as theoretical contributions, previous works publish exact specifications of concepts like “test case” or “test suite”. Based on these, contributions of practical relevance have been built with the goal to be easy to use for ASP developers. While they are equipped with sound languages for defining test cases, they are cumbersome to use, produce unnecessary overhead, implement an inefficient execution mechanism or lack an integration for a development environment.

In contrast to previous developments, this thesis comes with a lightweight and ready to use approach for implementing test cases in ASP. Not only does it utilize a simple annotation language for adding metadata (test specifications) to answer set programs, but it also incorporates the very same test specification language as part of a development environment. Test cases can be defined in the same file as the answer set program. Nonetheless, combining multiple test cases in a separate file for executing them like a test suite is possible. The evaluation showed, that the expressiveness of the annotation language is comparable with related works, while its execution is significantly faster. Furthermore, the annotations do not interfere with general program executability.

While the goal of this work was not to provide a comprehensive integrated development environment for ASP, it comes with an environment for realizing simple answer set programs and defining test cases. Additionally, ASP-WIDE features a basic file explorer, a code editor with syntax highlighting and an output window, besides the possibility to execute and test answer set programs. All developments regarding ASP-WIDE have been conducted in the light of future enhancements, which are not in the scope of this thesis. Therefore, the selection of appropriate technologies and all implementations had to be carried out carefully



in order to guarantee a solid foundation for future work. Especially, the selection of popular technologies like the Spring Framework (Java) and Angular assure that enhancements by interested individuals can be carried out seamlessly. In particular, the idea is to convert all implementations of this thesis into an open-source project, for instance on a platform like GitHub. This way the actual users of the development environment (ASP developers) can drive the development of ASP-WIDE. First things to improve could be the file explorer (workspace), the suggestions for the auto-completion or the output window of the IDE.

As a result this thesis concludes, that a lightweight annotation language with high expressiveness and comprehensive capabilities for defining tests in answer set programs, can be designed. Furthermore, its implementation, in particular its execution mechanism, can be engineered to be more efficient than previously known approaches. Also the integration of these aspects in a development environment is feasible, while its usage is comfortable and effective. In future all developments may be handed to the community for further improvements.

# Appendix A

## Annotation language

This Appendix contains the grammar of the developed annotation language. While the grammar is specified in EBNF (extended Backus-Naur form), also the contained lexical tokens are defined afterwards.

### A.1 EBNF grammar specification

```
<Expression> ::= [ <Expression> ] <Annotation>
<Annotation> ::= <RuleDefinition>
                | <BlockDefinition>
                | <TestDefinition>
                | <ProgramDefinition>
<RuleDefinition> ::= AT RULE PAR_OPEN NAME EQUAL STRING
                    [
                      COMMA BLOCK EQUAL STRING
                    ]
                    PAR_CLOSE
<BlockDefinition> ::= AT BLOCK PAR_OPEN NAME EQUAL STRING
                    [
                      COMMA RULES EQUAL CURL_OPEN
                      <RuleReferenceList> CURL_CLOSE
                    ]
                    PAR_CLOSE
<RuleReferenceList> ::= [
                        <RuleReferenceList> COMMA
                      ]
                        <RuleReference>
<RuleReference> ::= STRING
<TestDefinition> ::= AT TEST PAR_OPEN NAME EQUAL STRING COMMA
                    SCOPE EQUAL CURL_OPEN <ReferenceList> CURL_CLOSE COMMA
                    [
                      PROGRAM_FILES EQUAL CURL_OPEN
                      <ProgramFilesList> CURL_CLOSE COMMA
                    ]
                    [
                      INPUT EQUAL STRING COMMA
                    ]
                    [
                      INPUT_FILES EQUAL CURL_OPEN
                      <InputFilesList> CURL_CLOSE COMMA
                    ]
                    ]
```



```

<AssertConstraintInAtLeast> ::=  ASSERT_CIAL PAR_OPEN NUM_STR EQUAL NUMBER COMMA
                                CONSTRAINT EQUAL STRING PAR_CLOSE
<AssertConstraintInAtMost> ::=  ASSERT_CIAM PAR_OPEN NUM_STR EQUAL NUMBER COMMA
                                CONSTRAINT EQUAL STRING PAR_CLOSE
<AssertBestModelCost>        ::=  ASSERT_BMC PAR_OPEN COST EQUAL NUMBER COMMA
                                LEVEL EQUAL NUMBER PAR_CLOSE
<AssertNoAnswerSet>         ::=  ASSERT_NAS

```

## A.2 Lexical tokens

NUMBER	1, 2, 3, ...
STRING	"MyRuleName", "MyBlock", ...
AT	@
RULE	rule
RULES	rules
BLOCK	block
NAME	name
TEST	test
CONSTRAINT	constraint
SCOPE	scope
PROGRAM_FILES	programFiles
INPUT	input
INPUT_FILES	inputFiles
ASSERT	assert
ATOMS	atoms
NUM_STR	number
COST	cost
LEVEL	level
PROGRAM	program
ADD_FILES	additionalFiles
EQUAL	=
ASSERT_TIA	trueInAll
ASSERT_TIAL	trueInAtLeast
ASSERT_TIAM	trueInAtMost
ASSERT_TIE	trueInExactly
ASSERT_FIA	falseInAll
ASSERT_FIAL	falseInAtLeast
ASSERT_FIAM	falseInAtMost
ASSERT_FIE	falseInExactly
ASSERT_C	constraintForAll
ASSERT_CIE	constraintInExactly
ASSERT_CIAL	constraintInAtLeast
ASSERT_CIAM	constraintInAtMost
ASSERT_BMC	bestModelCost
ASSERT_NAS	noAnswerSet
PAR_OPEN	(
PAR_CLOSE	)
CURL_OPEN	{
CURL_CLOSE	}
COMMA	,
DOT	.

# References

## Literature

- [Alv+11] Mario Alviano et al. “The Disjunctive Datalog System DLV”. In: *Datalog Reloaded*. Ed. by Oege de Moor et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 282–301 (cit. on p. 4).
- [Alv+13] Mario Alviano et al. “The Fourth Answer Set Programming Competition: Preliminary Report”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Pedro Cabalar and Tran Cao Son. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 42–53 (cit. on p. 6).
- [Alv+17] Mario Alviano et al. “The ASP System DLV2”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Marcello Balduccini and Tomi Janhunen. Cham: Springer International Publishing, 2017, pp. 215–221 (cit. on p. 7).
- [Bar03] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003 (cit. on p. 3).
- [Bec02] Beck. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on p. 1).
- [BET11] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. “Answer Set Programming at a Glance”. *Commun. ACM* 54.12 (Dec. 2011), pp. 92–103. URL: <http://doi.acm.org/10.1145/2043174.2043195> (cit. on p. 3).
- [Bon+10] Piero Bonatti et al. “Answer Set Programming”. In: *A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, GULP*. Ed. by Agostino Dovier and Enrico Pontelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 159–182. URL: [https://doi.org/10.1007/978-3-642-14309-0\\_8](https://doi.org/10.1007/978-3-642-14309-0_8) (cit. on pp. 3, 4).
- [Cal+16] Francesco Calimeri et al. “Design and results of the Fifth Answer Set Programming Competition”. *Artificial Intelligence* 231 (2016), pp. 151–181. URL: <http://www.sciencedirect.com/science/article/pii/S0004370215001447> (cit. on p. 6).
- [De +12] Marina De Vos et al. “Annotating answer-set programs in LANA” (Oct. 2012) (cit. on pp. 9, 12, 15–18).
- [EMT05] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. “On the effectiveness of the test-first approach to programming”. *Software Engineering, IEEE Transactions on* 31 (Apr. 2005), pp. 226–237 (cit. on p. 1).

- [Feb+11] Onofrio Febbraro et al. “Unit Testing in ASPIDE”. *CoRR* abs/1108.5434 (2011). arXiv: 1108.5434. URL: <http://arxiv.org/abs/1108.5434> (cit. on pp. 8, 12–15, 20, 55).
- [Fra+03] Steven Fraser et al. “Discipline and practices of TDD: (test driven development).” In: Jan. 2003, pp. 268–270 (cit. on p. 1).
- [FRR11] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. “ASPIDE: Integrated Development Environment for Answer Set Programming”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by James P. Delgrande and Wolfgang Faber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 317–330 (cit. on pp. 8, 12).
- [GCP17] Stefano Germano, Francesco Calimeri, and Eliana Palermi. “LoIDE: a web-based IDE for Logic Programming - Preliminary Technical Report”. *CoRR* abs/1709.05341 (2017). arXiv: 1709.05341. URL: <http://arxiv.org/abs/1709.05341> (cit. on pp. 7, 34).
- [Geb+14] Martin Gebser et al. “Clingo = ASP + Control: Preliminary Report”. *CoRR* abs/1405.3694 (2014). arXiv: 1405.3694. URL: <http://arxiv.org/abs/1405.3694> (cit. on p. 7).
- [GKS12] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. “Conflict-driven answer set solving: From theory to practice”. *Artificial Intelligence* 187-188 (2012), pp. 52–89. URL: <http://www.sciencedirect.com/science/article/pii/S0004370212000409> (cit. on p. 7).
- [GMR15] Martin Gebser, Marco Maratea, and Francesco Ricca. “The Design of the Sixth Answer Set Programming Competition”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski. Cham: Springer International Publishing, 2015, pp. 531–544 (cit. on p. 6).
- [GMR17] Martin Gebser, Marco Maratea, and Francesco Ricca. “The Design of the Seventh Answer Set Programming Competition”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by Marcello Balduccini and Tomi Janhunen. Cham: Springer International Publishing, 2017, pp. 3–9 (cit. on p. 6).
- [Gom+08] Carla Gomes et al. “Satisfiability Solvers”. In: *Handbook of Knowledge Representation*. Ed. by Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter. Vol. 3. Elsevier, 2008. Chap. 2. URL: <http://www.sciencedirect.com/science/bookseries/15746526/3> (cit. on p. 7).
- [Jan+10] Tomi Janhunen et al. “On Testing Answer-Set Programs”. In: vol. 215. Aug. 2010, pp. 951–956 (cit. on pp. 10, 11, 20).
- [Jan+11] Tomi Janhunen et al. “Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison”. In: *Logic Programming and Nonmonotonic Reasoning*. Ed. by James P. Delgrande and Wolfgang Faber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 242–247 (cit. on p. 12).
- [Kau+16] Benjamin Kaufmann et al. “Grounding and Solving in Answer Set Programming”. *AI Magazine* 37 (Oct. 2016), pp. 25–32 (cit. on p. 6).

- [Lif08] Vladimir Lifschitz. “What is Answer Set Programming?” In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3. AAAI’08*. Chicago, Illinois: AAAI Press, 2008, pp. 1594–1597. URL: <http://dl.acm.org/citation.cfm?id=1620270.1620340> (cit. on pp. 1, 3, 6).
- [LR15] Nicola Leone and Francesco Ricca. “Answer Set Programming: A Tour from the Basics to Advanced Development Tools and Industrial Applications”. In: *Reasoning Web. Web Logic Rules: 11th International Summer School 2015, Berlin, Germany, July 31- August 4, 2015, Tutorial Lectures*. Ed. by Wolfgang Faber and Adrian Paschke. Cham: Springer International Publishing, 2015, pp. 308–326. URL: [https://doi.org/10.1007/978-3-319-21768-0\\_10](https://doi.org/10.1007/978-3-319-21768-0_10) (cit. on pp. 3–5).
- [LT94] Vladimir Lifschitz and Hudson Turner. “Splitting a Logic Program”. In: *Proceedings of International Conference on Logic Programming (ICLP)*. Ed. by Van Hentenryck and Pascal. 1994, pp. 23–37. URL: <http://www.cs.utexas.edu/users/ai-lab/?lif94e> (cit. on p. 13).
- [MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. USA: John Wiley & Sons, Inc., 2004 (cit. on p. 10).
- [Oet+12] Johannes Oetsch et al. “On the Small-scope Hypothesis for Testing Answer-set Programs”. In: *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning. KR’12*. Rome, Italy: AAAI Press, 2012, pp. 43–53. URL: <http://dl.acm.org/citation.cfm?id=3031843.3031849> (cit. on p. 12).
- [OPT11] Johannes Oetsch, Jörg Pührer, and Hans Tompits. “The SeaLion has Landed: An IDE for Answer-Set Programming—Preliminary Report”. *CoRR* abs/1109.3989 (2011). arXiv: 1109.3989. URL: <http://arxiv.org/abs/1109.3989> (cit. on p. 9).
- [Ric03] Francesco Ricca. “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*. 2003. URL: <http://ceur-ws.org/Vol-78/asp03-final-ricca.pdf> (cit. on p. 39).
- [SN98] Timo Soininen and Ilkka Niemelä. “Developing a Declarative Rule Language for Applications in Product Configuration”. In: *Practical Aspects of Declarative Languages*. Ed. by Gopal Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 305–319 (cit. on p. 3).

## Online sources

- [Cal+14] Francesco Calimeri et al. *ASP-Core-2 Input Language Format*. 2014. URL: <https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf> (visited on 03/19/2019) (cit. on p. 5).