Annotating answer-set programs in LANA*

MARINA DE VOS

Department of Computing,University of Bath, BA2 7AY, Bath, UK
(*e-mail:* `mdv@cs.bath.ac.uk`)

DOĞA GIZEM KISA

Faculty of Engineering and Natural Sciences, Sabanci University, Orhanli, Tuzla, Istanbul 34956, Turkey
(*e-mail:* `dgkisa@su.sabanciuniv.edu`)

JOHANNES OETSCH, JÖRG PÜHRER, HANS TOMPITS

Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
(*e-mail:* {oetsch,puehrer,tompits}@kr.tuwien.ac.at)

Abstract

While past research in answer-set programming (ASP) mainly focused on theory, ASP solver technology, and applications, the present work situates itself in the context of a quite recent research trend: *development support for ASP*. In particular, we propose to augment answer-set programs with additional meta-information formulated in a dedicated annotation language, called LANA. This language allows the grouping of rules into coherent blocks and to specify language signatures, types, pre- and postconditions, as well as unit tests for such blocks. While these annotations are invisible to an ASP solver, as they take the form of program comments, they can be interpreted by tools for documentation, testing, and verification purposes, as well as to eliminate sources of common programming errors by realising syntax checking or code completion features. To demonstrate its versatility, we introduce two such tools, viz. (i) ASPDoc, for generating an HTML documentation for a program based on the annotated information, and (ii) ASPUnit, for running and monitoring unit tests on program blocks. LANA is also exploited in the `SeaLion` system, an integrated development environment for ASP based on Eclipse.

KEYWORDS: answer-set programming, program annotations, documentation, unit testing

1 Introduction

Answer-set programming (ASP) (Gelfond and Lifschitz 1988, 1991; Baral 2003) is an established approach for declarative problem solving and non-monotonic reasoning. So far, research in ASP can basically be classified into three categories: (i) theoretical foundations of ASP including language extensions, (ii) performance

and capabilities of ASP solvers, and (iii) case studies and applications involving ASP. More recently, methods and methodologies to support an ASP programmer are increasingly becoming a focus of research interest (De Vos and Schaub 2007, 2009; Oetsch *et al.* 2010).

In this paper, we propose to augment programs with additional meta-information to facilitate the ASP development process. To this end, we devised a dedicated annotation language, LANA, standing for "Language for ANnotating Answer-set programs", that specifies specially formatted program comments. This meta-information is invisible to an ASP solver—therefore not altering the semantics of the program—but different tools may interpret and use the annotated information to various ends like documentation, testing, verification, code completion, or other development support.

One particular and quite central feature of LANA is grouping rules that are related in meaning into coherent blocks. Although different notions of modularity have already been proposed for ASP in the literature (Bugliesi *et al.* 1994; Eiter *et al.* 1997; Gelfond and Gabaldon 1999; Balduccini 2007; Janhunen *et al.* 2009), a strict concept of a program module can sometimes be a too tight corset. In particular, notions of modularity in ASP often come with their own semantics and different kinds of constraints need to be satisfied. For example, DLP-functions (Janhunen *et al.* 2009) require that their input and output signatures are disjoint and two DLP-functions need to satisfy certain syntactic constraints in order to be composable. On the other hand, *lp*-functions are a modular approach to build a logic program from its specification (Gelfond and Gabaldon 1999). That is, an *lp*-function is used to realise some functional specification that relates input and output relations for some domain by means of a logic program. The kind of grouping that we are proposing has, however, no semantical ramifications other than documenting that some rules belong together in a certain sense. Nevertheless, the benefit is that we add some extra structure to a program, improving the clarity and coherence of the program parts, which in turn can be used by other tools for, e.g., unit testing (Beck 2003). While unit testing is an integral element in software development using common languages like Java or C, it has been addressed in ASP only quite recently (Febbraro *et al.* 2011). We provide means to formulate unit tests for single blocks using LANA, allowing for easy regression testing. After rules are grouped into blocks, we may use further annotations to declare respective input and output signatures, which are also useful for testing and verification. Furthermore, we can declare the names and arities of predicates that are used within a block. This information can be exploited by, e.g., an integrated development environment (IDE) for syntax checking and code completion features while a user is writing a program. Besides names, description, and arities of predicates, one can also specify the domains of term arguments of a predicate using respective language features for declaring types. This information can be used for automated detection of type violations. These declarations have the potential to eliminate the source for quite common programmer errors with only little extra cost. For verification purposes, our annotation language can be used to specify assertions like pre- and postconditions for blocks. Preconditions are expected to hold for any input of a block, while postconditions have to hold for any output.

Together, they can be used to verify correctness of an ASP encoding with respect to such assertions.

Our main contributions are thus as follows:

- We introduce an annotation language for ASP that offers various ways to express meta-information for rules and other language elements. This information can be used to support and ease the development process, test and verify programs, and to eliminate many sources for common programmer errors.
- We describe ASPDoc, a JAVADOC[1] inspired tool, which takes an answer-set program, interprets the meta-comments, and automatically generates an HTML file documenting the program.
- We introduce ASPUNIT, an implementation of a unit-testing framework in the spirit of JUNIT[2] based on the structural annotations found in a program. This framework allows to formulate unit tests for individual program blocks, to execute them, and to monitor test runs.

The paper is organised as follows. In Section 2, we provide some background on ASP. Then, in Section 3, we introduce LANA, explain the basic language features, and illustrate them using a running example. Section 4 describes the basic features of ASPDoc while Section 5 does the same for ASPUNIT. Finally, we review related work in Section 6 and conclude in Section 7 with pointers for future work.

2 Background

Answer-set programming (ASP) (Gelfond and Lifschitz 1988, 1991; Baral 2003) is a declarative programming paradigm in which a logic program is used to describe the requirements that must be fulfilled by the solutions of a certain problem. The solutions of the problem can be obtained through the interpretation of the answer sets of the program, usually defined through a variant or extension of the stable-model semantics (Gelfond and Lifschitz 1988). This technique has been successfully applied in various domains such as planning (Eiter *et al.* 2002; Lifschitz 2002), configuration and verification (Soininen and Niemelä 1998), music composition (Boenn *et al.* 2011), or reasoning about biological networks (Dworschak *et al.* 2008) to just name a few. In the following, we briefly cover the essential concepts of ASP; for in-depth coverage, we refer to the well-known textbook by (Baral 2003).

The basic components of the language are *atoms*, elements that can be assigned a truth value. An atom a can be negated using *negation as failure*. A *literal* is an atom a or a negated atom not a. We say that not a is true if we cannot find evidence supporting the truth of a. Atoms and literals are used to create rules of the general form

$$a \ \text{:-} \ b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n,$$

---

[1] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`.
[2] `http://www.junit.org`.

where a, b_i, and c_j are atoms. Intuitively, this means *if all atoms b_i are known/true and no atom c_i is known/true, then a must be known/true*. We refer to a as the head and $b_1, \ldots, b_m, \texttt{not } c_1, \ldots, \texttt{not } c_n$ as the body of the rule. Rules with empty body are called *facts*. Rules with empty head are referred to as *constraints*, indicating that no solution satisfies the body. A (*normal*) *program* is a set of rules. The semantics is defined in terms of *answer sets*, i.e., assignments of true and false to all atoms in the program that satisfy the rules in a minimal and consistent fashion. A program has zero or more answer sets, each corresponding to a solution.

Different extensions to the language have been proposed. On the one hand, we have syntactic extensions, providing mere, but very useful, syntactic sugar. On the other hand, we have semantic extensions where the formalism itself is generalised.

From a programmer's perspective, *choice rules* (Niemelä *et al.* 1999) are probably the most commonly used extension. Many problems require choices between a set of atoms to be made. Although this can be modelled in the basic formalism, it tends to increase to the number of rules and increases the possibility of errors. To avoid this, choices are introduced. Choices, written $L\{l_1, \ldots, l_n\}M$, are a convenient construct to indicate that at least L and at most M literals from the set $\{l_1, \ldots, l_n\}$ must be true in order to satisfy the construct. Bound L defaults to 0 while M defaults to n. Choice rules are often used with a grounding predicate: $L\{A(X) : B(X)\}M$ represents the choice of a number of atoms where $A(X)$ is grounded with all values of X for which $B(X)$ is true.

One of the major extensions to the language was the introduction of disjunction in the head of rules (Gelfond and Lifschitz 1991). When the body of the rule is true, we need to have at least one head atom that is true. Although at first it may seem that disjunctive programs can easily be translated to non-disjunctive programs, this is not the case. Both types of programs are in different complexity classes (under the usual complexity-theoretic proviso that the polynomial hierarchy does not collapse).

Algorithms and implementations for obtaining answer sets of logic programs are referred to as *answer-set solvers*. The most popular and widely used solvers are DLV (Eiter *et al.* 1998), providing solver capabilities for disjunctive programs, and the SAT inspired CLASP (Gebser *et al.* 2007). Alternatives are SMODELS (Niemelä and Simons 1997) and CMODELS (Giunchiglia *et al.* 2004), a solver based on translating the program to SAT.

3 An annotation language for ASP

In this section, we describe our annotation language LANA. An overview of most language elements of LANA appears in Tables 1 and 2 gives a summary of the remaining language elements related to testing in LANA. We make use of a simple answer-set program to illustrate all the language features in a step-by-step fashion. In particular, we use an encoding of the *Battleship* puzzle. A solved instance of Battleship appears in Figure 1. In Battleship, a group of ships is hidden on a grid and one has to find the positions of each. The fleet consists of one four-

Element	Definition	Informal Description
block element	*block* | *atom* | *term* | *input signature* | *output signature* | *precondition* | *postcondition*	LANA elements related to blocks.
block	"@block" *name* "{" [*description*] {*block element*} [*ASP code*] "}"	Groups ASP rules into coherent parts.
atom	"@atom" *name* "("*termList*")" [*description*]	Defines a predicate; *termList* are the predicate's arguments.
term	"@term" *name* [*description*] [*type*]	Declares a term from some *atom* term list, its meaning, and type information.
type	"@from" *groundTerms* | "@with" *ruleBdy* | "@samerangeas" *term*	Type of a term is defined by a list of ground terms, the terms satisfying *ruleBdy*, or as the type of another term.
input signature	"@input" *inputPredicates* | "@requires" *inputPredicates*	Declares input predicates of a block as a list of name/arity pairs.
output signature	"@output" *outputPredicates* | "@defines" *outputPredicates*	Declares output predicates of a block as a list of name/arity pairs.
assertion	"@assert" *name* "{" [*description*] *assertspec* "}"	A logical condition for answer sets.
pre-condition	"@precon" *name* "{" [*description*] *assertSpec* "}"	A logical condition for the inputs of a block.
post-condition	"@postcon" *name* "{" [*description*] *assertspec* "}"	A logical condition for the answer sets of a block.
assertspec	("@always" | "@never") *atmList* [*embASPcode*]	The testmode for assertions, preconditions and postconditions; *embASPcode* is code within the LANA comment environment.

squares long battleship, two three-squares long cruisers, three two-squares long destroyers, and four one-square long submarines—each ship is placed horizontally or vertically on the grid such that no ship is touching any other ship (not even diagonally). To provide hints, some squares may show parts of a ship or water. Moreover, a number besides each row and each column indicates how many squares in that row or column are occupied by ship parts. A solution of a puzzle is a configuration of all the ships that is consistent with the initially given hints.

Assume that a puzzle instance is defined in terms of facts `water/2` and `ship/2` for specifying which squares contain water or parts of a ship, respectively. The facts `rowHint/2` and `colHint/2` determine the numbers associated with each row and each column. Problem solutions are represented by facts `ship(W,X,Y,Z)` expressing that a ship is occupying the squares from position `(W,X)` to `(Y,Z)`.

Table 2. *Test case specification in* Lana

| Element	Definition	Informal Description
test case	"`@testcase`" *name* [*description*] "`@scope`" *blocks* *testcond* [*ASP code*]	A test case for the blocks from *blocks* passes if the blocks joined with *ASP code* satisfy all specified test conditions.
testcond	"`@testhasanswerset`" | "`@testnoanswerset`" | "`@testatoms`" *atmList mode*	A test condition holds if one, resp., no, answer set is found, or all ground atoms in *atmList* are entailed according to *mode*.
mode	"`@trueinall`" | "`@trueinatleast`" *n* | "`@trueinatmost`" *n* | "`@falseinall`" | "`@falseinatleast`" *n* | "`@falseinatmost`" *n*	Mode of entailment of a test condition, i.e., if test atoms are true, resp., false, in all, at most *n*, or at least *n* answer-sets.
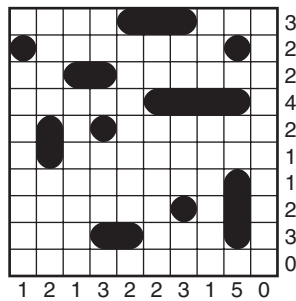
Fig. 1. Solved instance of a Battleship puzzle.

3.1 Blocks

In general, all keywords of our annotation language start with the symbol @. A central feature of Lana is to group rules together. This is done using the `@block` keyword. To specify that we are going to define a couple of rules that encode solutions to the Battleship puzzle, we declare a block with the name `Battleship` as follows:

```
%** @block Battleship { *%

 % encoding of the Battleship puzzle

%** } *%
```

The annotation `@block` is followed by an optional name of the block and the opening bracket "{". Everything between "{" and the closing bracket "}" now belongs to the block `Battleship`. Blocks can be nested but they must not overlap.

Note that every annotation has the form of an ASP comment. ASP block-comments can be used instead of starting every single line with "%" when an ASP solver, in particular its grounding component, supports this feature. To distinguish annotations from ordinary (block-)comments, an additional star "∗" is always placed after "%∗" at the beginning.

3.2 Predicate signatures

LANA allows to declare the names and arities of used predicates. Often, predicates play different roles in an encoding, and it can make sense to explicitly distinguish between these roles. In particular, it can make sense to document which are the relations that are defined by the rules of a block (those will usually appear in the heads of some rules) and which are the relations that are expected to be already defined by some other rules (the required predicates will usually appear in some rule bodies). This distinction is closely related to intensional and extensional predicates in a database setting. If a block is regarded as a declarative problem-solving module, the predicates that are required will usually encode problem instances, and respective predicates form the *input signature* of a block. Accordingly, the relations that are defined form the *output signature* of a block. We note, however, that input and output signatures might overlap in a declarative setting.

We proceed by declaring the predicate names as well as the input and output signatures of our encoding as described above. Within block `Battleship`, we add the following:

```
%**
 @atom water(X,Y)
 there is no ship at position (X,Y)

 @atom ship(X,Y)
 a ship is occupying position (X,Y)

 @atom rowHint(X,H)
 in row X, H squares are occupied

 @atom colHint(Y,H)
 in column Y, H squares are occupied

 @atom ship(X1,Y1,X2,Y2)
 a ship is occupying the squares from (X1,Y1) to (X2,Y2)

 @input water/2, ship/2, rowHint/2, colHint/2

 @output ship/4
*%
```

We use `@atom` to introduce the name of a predicate along with its arity and some information describing its intended use, and we use `@input` and `@output` to determine which predicates symbols are used to represent input for the block and which ones correspond to output. For input and output signatures, we also have to

explicitly give the arity of the involved predicates. This is needed to disambiguate between predicates having the same name but a different number of arguments, as ship in our running example. Note that annotations are optional, declarations are not enforced. We stress that declaring input and output signatures should be done only if this is beneficial for the development process and is consistent with the declarative reading of a program. Input and output are terms that are quite commonly used in declarative programming—e.g., all problem specifications for the benchmark problems used for the ASP competitions explicitly define input and output signatures. However, if rules are seen as more general definitions of relations of some problem domain, it might be more appropriate to use @defines to declare the relations that are defined by a block of rules and @requires to declare the input signature of a block. Again, such annotations are not mandatory. The more information is made explicit, the more it can be used by tools that interpret such information to the benefit of the developer.

3.3 Types

Regarding the declaration of predicates, we can provide information about the types of its term arguments. Assume that row and column positions are specified by ascending integers starting from 1. To make this assumption explicit, we add the following lines to the block:

```
%**
 @term X, Y
 @from 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
 X (Y) is a row (column) index ranging from 1 to 10
*%
```

Here, we use @term to declare variable names and @from to specify the type of a variable by means of a non-empty comma-separated list of admissible ground terms. As an alternative to @from, we can use @with followed by a rule body:

```
%** @with integer(#V), #V>1, #V<10 *%
integer(1..1000).
```

The type of X and Y is now determined by the ground substitutions for the reserved symbol #V that satisfy the rule body given after @with. Here, "integer(1..1000)." is regular ASP code for defining facts that encode the integer range that we are considering. Furthermore, to express that variables are of the same type as ones already known, we can use @samerangeas, as illustrated next:

```
%**
 @term X1, Y1, X2, Y2
 further row and column indices
 @samerangeas X
*%
```

Thus, any of X1, Y1, X2, and Y2 is of the same type as X. For future work, we also plan to extend LANA so that predefined names for at least basic types like strings or integers can directly be used to specify the types of variables.

As mentioned previously, blocks can be nested. To proceed with our Battleship encoding, we add a block of rules within block `Battleship` for guessing solution candidates according to the usual guess-and-check paradigm:

```
%**
 @block Guess {
 guess a configuration of battleships on the grid
*%
r(1..10). c(1..10).
{ship(X1,Y1,X2,Y2):r(X1):c(Y1):r(X2):c(Y2):X2>=X1:Y2>=Y1}.
:- ship(X1,Y1,X2,Y2), X1!=X2, Y1!=Y2.
%** } *%
```

Note that in block `Guess`, all declarations for predicates and terms are inherited from the enclosing block `Battleship`. One can proceed in a similar way to define blocks for constraints (along with auxiliary definitions) to prune away unwanted solution candidates to complete the encoding.

3.4 Assertions

Towards testing and the verification of programs, LANA allows the formulation of general assertions, as well as pre- and postconditions for blocks. A precondition is a logical condition that is assumed to hold for any input while a postcondition has to hold on any output of a block. Together, pre- and postconditions can be regarded as a *contract*: it is the responsibility of any input provider that a block's preconditions are satisfied, and it is the responsibility of the rules in the respective block that its postconditions are satisfied. Both pre- and postconditions are formulated in ASP itself; they are placed within the block they belong to. As an illustration, we formulate as a precondition of our Battleship encoding that no square shows both water and parts of a ship jointly as follows:

```
%**
 @precon Excl {
 no square shows both water and a part of a ship
 @never clash
 clash :- water(X,Y), ship(X,Y).
 }
*%
```

In general, a precondition is introduced with the keyword `@precon` followed by a name for the condition. The actual content is then written enclosed between "{" and "}", similar to the definition of blocks. An optional description follows. Then, we have to specify a non-empty list of ground atoms after the keyword `@never` or `@always`. After that, we add some ASP rules that define the before-mentioned ground atoms. The intended semantics is as follows. Some input, i.e., a set of facts over a block's input signature, passes a precondition if that input combined with the rules of the precondition cautiously entails all the ground atoms after `@always` and the negation of the atoms after `@never`.

Postconditions are expressed analogously to preconditions. To say that battleships must not be longer than four squares, we add the following code to our Battleship block:

```
%**
@postcon Overlength {
battleships are never longer than four squares
@never ov
ov :- ship(X1,Y1,X2,Y2),L=X2-X1,L>4.
ov :- ship(X1,Y1,X2,Y2),L=Y2-Y1,L>4.
}
*%
```

A block and an input for a block satisfy a postcondition if the block joined with the input and the rules of the postcondition cautiously entails all the ground atoms after `@always` and the negation of the atoms after `@never`.

Having pre- and postconditions explicitly formalised offers various ways to support the development process. For one thing, they can be used to automatically generate input instances for testing purposes. This can be automated for systematic testing of a block, including random testing and structure-based testing (Janhunen *et al.* 2010, 2011). Towards program verification, one can check whether a block passes its postconditions for any admissible input, at least from some fixed small scope, i.e., involving only a bounded number of constant symbols. Exhaustive testing with respect to a small scope showed to be quite effective in ASP (Oetsch *et al.* 2012). Though they are formulated in ASP itself and thus tend to duplicate some code from within a block, pre- and postconditions are especially of great value if the rules in a block are changed, e.g., to optimise an encoding, and one wants to be sure that the changes did not render the program incorrect. This can be done, e.g., by searching for inputs within some small scope that violate some postcondition of the optimised program, provided respective tool support is available.

We note that pre- and postconditions make only sense in a setting where we can also distinguish between input and output relations. If this is not the case and one wants to formulate general assertions on the answer sets of a program, LANA allows to define them using `@assert`. Semantically, an assert statement is a postcondition of the whole program.

3.5 Unit tests

Though pre- and postconditions allow to partially verify program correctness, LANA also supports a light-weight form of program verification that is inspired by unit testing. A unit test in procedural languages is commonly a test for an individual function or procedure. While in a related approach for unit testing answer-set programs (Febbraro *et al.* 2011), the scope of a test is defined in terms of sets of rules, unit tests are formulated for blocks or sets of blocks in our setting. To check whether the guessing part of our running example generates solution candidates where one ship occupies precisely the first four horizontal squares of the field, we could formulate a unit test as follows:

```
%**
 @testcase ShipTopLeftCorner
 a ship is horizontally placed at the top-left corner
 @scope Guess
 @testatoms goalShip @trueinatleast 1
*%
goalShip :- ship(1,1,1,4).
```

A unit test starts with the reserved word `@testcase` followed by a name. Then, a short description of the purpose of the unit test may be given as a comment. After `@scope`, a list of block names is expected. In the above example, we used `@testatoms` to declare that `goalShip` is an atom that has to be true in at least one answer set (`@trueinatleast 1`) of block Guess joined with the subsequent rule that defines `goalShip`. Instead of or additional to `@trueinatleast` n, a tester might use `@trueinatmost` m, `trueinall`, `falseinatleast` p, `falseinatmost` q, and `falseinall`, where m, n, p, and q are positive integers. Also, instead of `@testatoms`, one may use `@testhasanswerset` or `@testnoanswerset` to express that at least one or no answer set is expected, respectively.

The semantics of a unit test is as follows. A test case passes iff the answer sets of the rules of the test case combined with all the blocks specified after `@scope` satisfy the testing conditions expressed using any of `@testatoms`, `@testhasanswerset`, and `@testhasnoanswerset`. For example, to additionally test that a ship is never placed diagonally on the field, one could formulate a further test case:

```
%**
 @testcase NoDiagonalShips
 ships are never placed diagonally on the field
 @scope Guess
 @testatoms forbiddenShip @falseinall
*%
forbiddenShip :- ship(1,1,3,3).
```

Of course, this test case can only guarantee that one particular ship is not placed diagonally at some particular position. However, this distinguishes test cases from more general assertions like postconditions. To generalise the above test case to arbitrary ships, we would rather use a postcondition. Typically, test cases represent concrete situations by means of facts that can be easily verified by a user and document individual situations that are allowed or forbidden. They cannot, however, guarantee correctness of an encoding but only increase our confidence regarding its functionality.

Unit testing is a convenient way to test properties of individual blocks of an ASP encoding. Furthermore, they can be used to iteratively develop programs in a test-driven fashion. In test-driven development, unit tests are formulated before the code is written. First, a unit test for a single property of the block that we want to develop is specified. Then, it is checked whether the test case fails for the program under development. If this is the case, the block is extended by the necessary rules to make the failed test case pass. After the code is refactored towards efficiency, readability, etc., and after it is verified that all test cases still pass, the next property is addressed by formulating a respective unit test. This continues until the program is

complete. For illustration, the unit tests `ShipTopLeftCorner` and `NoDiagonalShips` will pass for the current state of the Battleship program. However, if we want to implement the property that ships must not touch each other, we could specify the following test case (which will currently fail):

```
%**
 @testcase TouchingShips
 two ships must not touch each other
 @scope Guess,Touch
 @testatoms forbiddenShip @falseinall
*%
forbiddenShip :- ship(1,1,1,2), ship(1,2,1,4).
```

Then, we would proceed to implement a block `Touch` with a constraint that forbids answer sets with ships that are touching each other and check whether the new test case and the old ones pass.

4 ASPDoc

ASPDoc is a command-line tool that interprets meta-information given in an answer-set program and generates a corresponding HTML documentation file similar to, e.g., JavaDoc for Java programs. Information regarding block structure, input and output signatures, used predicate symbols, etc. is clearly arranged so that the answer-set program can easily be understood, used, or extended by other developers. Such documentation features are especially useful to make ASP problem-solving libraries, i.e., collections of ASP encodings that can be used as building blocks for larger programs, accessible to developers.

The tool is developed in Java; an executable JAR file is available on the web.[3] Assume that the source code of our running example is stored in a file, say `battleship.gr`. A corresponding HTML documentation can be created as follows:

```
java -jar aspdoc.jar -p battleship.gr
```

Different HTML documents are created with `index.html` as the usual entry point. Here, option `-p`, or `-potassco`, is used to tell ASPDoc that the answer-set program is written using `gringo` syntax. For DLV, option `-d` or `-dlv` can be used instead. Furthermore, an output directory d can be specified with option `-o=d`, and help on available options can be obtained with the option `-h`. A summary of ASPDoc options is given in Table 3.

A screenshot of the documentation for the Battleship example presented above is given in Figure 2. The documentation of the complete encoding can be found online.[4] The document contains descriptions of all the blocks of the answer-set program, where sub blocks are indented relative to their parent blocks. To provide an overview, a summary of the block structure of the entire answer-set program is

---

[3] http://students.sabanciuniv.edu/dgkisa/aspdoc-aspunit.
[4] http://www.kr.tuwien.ac.at/research/projects/mmdasp/battleship.

Table 3. *Command-line options for* ASPDOC

Option	Description
-o=*path*	set output directory to *path*
-HA	show hidden atoms
-ha	do not show hidden atoms
-S	include ASP code in the HTML document
-s	do not include ASP code in the HTML document
-A	include LANA code in generated ASP code
-a	do not include LANA code in generated ASP code
-potassco, -p	input language is that of gringo
-dlv, -d	input language is that of DLV
-help, -h	print usage information

Fig. 2. (Colour online) HTML documentation of the Battleship program.

presented at the beginning of the documentation. We note that programmers are not forced to declare blocks at all. If no block is specified in a file, all rules in that file belong to a dedicated default block. For each block, descriptions of the used predicates and types of involved terms, as well as pre- and postconditions are given. By default, hidden atoms, i.e., atoms mentioned neither in a block's input nor in its output signature, are displayed as well in a dedicated section entitled "Hidden Atoms". To hide them, option -ha can be used. The document also contains a link to the actual rules inside a block, unless this is suppressed using option -s. These rules are, by default, displayed together with the meta-comments of LANA. If option -a is used, such comments are not shown. Likewise, the rules for defining pre- and postconditions can be inspected by using the respective links. To enhance navigability between different parts of the document, predicates used in the source code view or in signature declarations are, if available, linked to their respective descriptions. For

instance, to find out the range of a variable in the output atoms section, say $X1$, the user can simply follow the link and thereby navigate to the description of $X1$ (and X) in the term description section. Further options to customise the appearance of the documentation are planned for future work.

5 ASPUnit

ASPUnit is a tool to execute test cases that are formulated in Lana. Like ASPDoc, it is a command-line tool. An executable JAR file can be downloaded from the same web page as ASPDoc. For ASPUnit, each unit test has to be stored in a separate file. Although test cases are required to have a name, we allow that a user may omit an explicit name, in which case the file name is used by default. The tool takes as input a test-suite specification file, i.e., a file that contains the relevant information regarding locations of the answer-set program, the files containing individual test cases, and the ASP solver that is needed to execute them. The syntax of a test-suite specification file is closely related to our annotation language itself. In particular, the specification of a test-suite has the following form:

```
@testsuite name
 description
@program    ASPfiles
@programdir pathToASPfile
@test       testCaseFile1
@test       testCaseFile2
 ...
@testdir    pathToTestFiles
@solvertype ASPsolver
@solver     solverFile
@grounder   grounderFile
```

Hence, a test-suite specification starts with `@testsuite` followed by a name. Then, a short description may be given. A list of file names that together contain the answer-set program under test is expected after `@program`. These file names are relative to a path specified after `@programdir`. For each test case that we want to execute, we have to provide the file name that contains that test case specified by `@test`. The path to these files appears after `@testdir`. Then, information regarding the ASP solver has to be given. For this, `@solvertype` is used; the solver type is one of DLV, `clasp`, or `clingo`. After `@solver`, an absolute file name of the ASP solver is expected. This file name may include additional parameters for that solver. If a separate grounder is needed, like for `clasp`, an absolute file name including command-line parameters have to be specified after `@grounder`.

Now, to run a bunch of test cases specified within a test-suite file, say `testsuite`, ASPUnit is invoked as follows:

```
java -jar aspunit.jar testsuite
```

The tool will run all the unit tests on the answer-set program using the solver settings according to specifications in `testsuite`. A test report is printed to standard-output.

Table 4. *Command-line options for* ASPUNIT

Option	Description
-CE	show counterexample if a test case fails
-ce	do not show counterexample if a test case fails
-D	show description of a test case if it fails
-d	do not show description of a test case if it fails
-help, -h	print usage information

```
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).
```

Fig. 3. A test report for the Battleship program.

This report contains information regarding success or failure for each test case. If a test case fails, a counterexample may be included, depending whether option -CE is set when ASPUNIT is executed. Furthermore, if option -D is used, the test report will contain a short description of each test case that fails, obtained from the specification of the test cases themselves. For illustration, assume we run the test cases presented in Section 3 on the partial encoding of Battleship. Recall that the first and second test case pass while the third one fails. The resulting test report, including a description of each test case and counterexamples for the failing test case, is given in Figure 3. A summary of ASPUNIT command-line options is given in Table 4.

6 Related work

As mentioned earlier, there are many notions of modularity for ASP in existence (Bugliesi *et al.* 1994; Eiter *et al.* 1997; Gelfond and Gabaldon 1999; Balduccini 2007; Janhunen *et al.* 2009). The advantage of the light-weight approach of LANA for grouping rules together by means of declaring blocks is that we do not change the semantics of programs and they can be directly parsed by ASP solvers. However,

there are also disadvantages compared to other approaches to modularise logic programs. For example, related to our approach for annotating type information is RSIG (Balduccini 2007). RSIG is an language extension for specifying simple type information for programs and modules and thus requires its own parser. However, this type information simplifies the program rules as information about the type of variables is not required to be provided. On the other hand, DLP-functions (Janhunen *et al.* 2009) allow to compute the semantics of a logic program based on the semantics of its separate DLP-functions which opens up new paths for potentially more efficient answer-set computation. To support the incremental development of logic programs, `lp`-functions can be used to structure a program and to develop it along with its specification by using specification constructors and their realisation theorems (Gelfond and Gabaldon 1999). Blocks in LANA are not designed to serve one particular purpose, different interpretations are conceivable and are eventually determined by respective tool support like ASPUNIT for unit testing blocks of rules.

In general, developing and debugging a declarative language is quite different from software engineering in a more traditional procedural or object-oriented programming language. With larger programs for real-world applications being written, it is vital to support the programmer with the right tools. In recent years, some work has been done to provide the ASP programmer with dedicated tools. The integrated development environments APE (Sureshkumar *et al.* 2007) and `SeaLion` (Oetsch *et al.* 2011) provide, among other features, syntax colouring and syntax checking for ASP programs and run as an Eclipse front-end to solvers. IDEs for the DLV solver and its extensions are discussed by (Perri *et al.* 2007) and (Febbraro *et al.* 2011). Debugging in ASP is supported by SPOCK (Brain *et al.* 2007), which makes use of ASP to explain and handle unexpected outcomes like missing atoms in an answer set or the absence of an answer set. (Cliffe *et al.* 2008) and (Kloimüllner *et al.* 2011) provide mechanisms to visualise answer sets of a given program to support code debugging.

To support large application developments, traditional languages offer programming tools that automatically generate searchable documentation, like e.g., JAVADOC. Methodologies like test-driven development provide a mechanism to incrementally unit test code and to support regression testing; JUNIT is an example of this for Java. LANA provides the support for both, incorporating the annotation of tests directly into the documentation of the program. The use of assertions in LANA is inspired by the Java Modelling Language (Leavens and Cheon 2006) and annotations as used in Prolog (Kulas 2000).

Similar to our unit testing approach, Prolog offers unit-testing functionality called PLUNIT.[5] As in our approach, where tests are expressed using ASP itself and only a Java wrapper is used to call all tests within a given test-suite, tests in PLUNIT are formulated as Prolog clauses.

---

[5] `http://www.swi-prolog.org/pldoc/package/plunit.html`.

(Febbraro *et al.* 2011) provide a mechanism for unit testing in ASP, which is incorporated in their IDE `Aspide`. They base unit tests on clusters on the dependency graphs or rule labelling, while we allow the user to decide which rules belong to a test by defining blocks.

7 Conclusion and future work

In this paper, we presented LANA, an annotation language for ASP. This language can be used to structure a program into blocks and to declare language elements like predicates with type information, input and output signatures, pre- and postconditions, test cases, etc. Annotations do not interfere with the languages of answer-set solvers as they have the form of program comments. The main advantage of such annotations is that they can be interpreted by tools to support the development process, to automatically test and verify programs, and to increase maintainability by enhancing program documentation. In fact, we implemented and described two such tools, namely ASPDOC for generating an HTML documentation for a program, and ASPUNIT for running and monitoring unit tests. The former tool is especially useful for maintaining and using larger collections of program modules; the latter tool is used for managing a test corpus when a program is developed and to enable test-driven development methods, a methodology popular in industry, e.g., as in extreme and agile programming.

While many interesting features of LANA for development support can be realised by stand-alone tools like ASPDOC and ASPUNIT, things become more interesting when the respective functionalities are available within an IDE for ASP. The two most actively developed IDEs for ASP at present are `SeaLion` (Oetsch *et al.* 2011) and `Aspide` (Febbraro *et al.* 2011). Then, the proposed language can be used as a basis to realise intelligent syntax highlighting, static or dynamic type checking, code completion, and so on. Indeed, LANA can already be parsed by `SeaLion` and the features of ASPDOC are already available from within the IDE. Further integration is planned with the goal that all features sketched in this paper are supported in `SeaLion`.

Furthermore, we want to empirically evaluate to what extent additional meta-information is beneficial for program development within courses on declarative problem solving at our universities. In general, program annotations provide a wealth of information. One of the main issues with debugging ASP programs is the difficulty of working out the program's interpretation of the problem (resp., solution), and the programmer's view of the problem (resp., solution). Using the meta-data, it would be possible to automatically generate a semi-natural language reading of the program, allowing programmers to cross-check their interpretation of the program with that of the program itself. This is only possible if developers use a specific grammar to annotate the various components of the program.

In traditional software engineering, coding standards including documentation are imposed, especially in case of developing large software projects. In this paper, we propose LANA as part of coding standards for ASP. Future work will look into other

best-practices for writing and maintaining ASP programs. The growing number of applications of ASP can provide a wealth of information for this.

References

BALDUCCINI, M. 2007. Modules and signature declarations for A-Prolog: Progress report. See De Vos and Schaub (2007), 41–55.

BARAL, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Cambridge, England.

BECK, K. 2003. *Test-Driven Development: By Example*. Addison-Wesley Professional.

BOENN, G., BRAIN, M., DE VOS, M. AND FITCH, J. 2011. Automatic music composition using answer set programming. *Theory and Practice of Logic Programming 11,* 2–3, 397–427.

BRAIN, M., GEBSER, M., PÜHRER, J., SCHAUB, T., TOMPITS, H. AND WOLTRAN, S. 2007. "That is illogical captain!" – The debugging support tool spock for answer-set programs: System description. See De Vos and Schaub (2007), 71–85.

BUGLIESI, M., LAMMA, E. AND MELLO, P. 1994. Modularity in logic programming. *The Journal of Logic Programming 19 & 20*, 443–502.

CLIFFE, O., DE VOS, M., BRAIN, M. AND PADGET, J. 2008. ASPVIZ: Declarative visualisation and animation using answer set programming. In *Proceedings of the ICLP 2008*. Lecture Notes in Computer Science, vol. 5366. Springer, 724–728.

DE VOS, M. AND SCHAUB, T., Eds. 2007. *First International Workshop on Software Engineering for Answer Set Programming* (*SEA 2007*).

DE VOS, M. AND SCHAUB, T., Eds. 2009. *Second International Workshop on Software Engineering for Answer Set Programming* (*SEA 2009*).

DWORSCHAK, S., GRELL, S., NIKIFOROVA, V. J., SCHAUB, T. AND SELBIG, J. 2008. Modelling biological networks by action languages via answer set programming. *Constraints 12,* 1, 21–65.

EITER, T., FABER, W., LEONE, N., PFEIFER, G. AND POLLERES, A. 2002. The DLV^k planning system: Progress report. In *Proceedings of the JELIA 2002*. Lecture Notes in Computer Science, vol. 2424. Springer, 541–544.

EITER, T., GOTTLOB, G. AND VEITH, H. 1997. Modular logic programming and generalized quantifiers. In *Proceedings of the LPNMR'97*. Lecture Notes in Computer Science, vol. 1265. Springer, 290–309.

EITER, T., LEONE, N., MATEIS, C., PFEIFER, G. AND SCARCELLO, F. 1998. The KR system DLV: Progress report, comparisons and benchmarks. In *Proceedings of the KR'98*. Morgan Kaufmann, 406–417.

FEBBRARO, O., LEONE, N., REALE, K. AND RICCA, F. 2011. Unit testing in ASPIDE. In *Proceedings of the INAP/WLP 2011*. INFSYS Research Report 1843-11-06, 165–176.

FEBBRARO, O., REALE, K. AND RICCA, F. 2011. ASPIDE: Integrated development environment for answer set programming. In *Proceedings of the LPNMR 2011*. Lecture Notes in Computer Science, vol. 6645. Springer, 317–330.

GEBSER, M., KAUFMANN, B., NEUMANN, A. AND SCHAUB, T. 2007. Conflict-driven answer set solving. In *Proceedings of the IJCAI 2007*. AAAI Press/The MIT Press, 386–392.

GELFOND, M. AND GABALDON, A. 1999. Building a knowledge base: An example. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 165–199.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*. MIT Press, 1070–1080.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9,* 3-4, 365–386.

GIUNCHIGLIA, E., LIERLER, Y. AND MARATEA, M. 2004. SAT-based answer set programming. In *Proceedings of the AAAI 2004*. AAAI Press / The MIT Press, 61–66.

JANHUNEN, T., NIEMELÄ, I., OETSCH, J., PÜHRER, J. AND TOMPITS, H. 2010. On testing answer-set programs. In *Proceedings of the ECAI 2010*. Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press, 951–956.

JANHUNEN, T., NIEMELÄ, I., OETSCH, J., PÜHRER, J. AND TOMPITS, H. 2011. Random vs. structure-based testing of answer-set programs: An experimental comparison. In *Proceedings of the LPNMR 2011*. Lecture Notes in Computer Science, vol. 6645. Springer, 242–247.

JANHUNEN, T., OIKARINEN, E., TOMPITS, H. AND WOLTRAN, S. 2009. Modularity aspects of disjunctive stable models. *Journal Artificial Intelligence Research 35*, 813–857.

KLOIMÜLLNER, C., OETSCH, J., PÜHRER, J. AND TOMPITS, H. 2011. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Proceedings of the INAP/WLP 2011*. INFSYS Research Report 1843-11-06, 152–164.

KULAS, M. 2000. Annotations for Prolog - A concept and runtime handling. In *Selected Papers of the 9th International Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 1999)*. Lecture Notes in Computer Science, vol. 1817. Springer-Verlag, 234–254.

LEAVENS, G. T. AND CHEON, Y. 2006. Design by contract with JML. ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf.

LIFSCHITZ, V. 2002. Answer set programming and plan generation. *Journal of Artificial Intelligence 138,* 1-2, 39–54.

NIEMELÄ, I. AND SIMONS, P. 1997. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the LPNMR'97*. Lecture Notes in Computer Science, vol. 1265. Springer, 420–429.

NIEMELÄ, I., SIMONS, P. AND SOININEN, T. 1999. Stable model semantics of weight constraint rules. In *Proceedings of the LPNMR'99*. Lecture Notes in Computer Science, vol. 1730. Springer, 317–331.

OETSCH, J., PRISCHINK, M., PÜHRER, J., SCHWENGERER, M. AND TOMPITS, H. 2012. On the small-scope hypothesis for testing answer-set programs. In *Proceedings of the KR 2012*.

OETSCH, J., PÜHRER, J. AND TOMPITS, H. 2010. Methods and methodologies for developing answer-set programs—project description. In *Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010)*. LIPIcs, vol. 7. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 154–161.

OETSCH, J., PÜHRER, J. AND TOMPITS, H. 2011. The SeaLion has landed: An IDE for answer-set programming—preliminary report. In *Proceedings of the INAP/WLP 2011*. INFSYS Research Report 1843-11-06, 141–151.

PERRI, S., RICCA, F., TERRACINA, G., CIANNI, D. AND VELTRI, P. 2007. An integrated graphic tool for developing and testing DLV programs. See De Vos and Schaub (2007), 86–100.

SOININEN, T. AND NIEMELÄ, I. 1998. Developing a declarative rule language for applications in product configuration. In *Proceedings of the PADL'98*. Lecture Notes in Computer Science, vol. 1551. Springer, 305–319.

SURESHKUMAR, A., DE VOS, M., BRAIN, M. AND FITCH, J. 2007. APE: An Ans Prolog* environment. See De Vos and Schaub (2007), 101–115.