LANA: A Language for Annotating Answer-Set Programs*

Marina De Vos[1], Doğa Gizem Kısa[2], Johannes Oetsch[3], Jörg Pührer[3], and Hans Tompits[3]

[1] Department of Computer Science,
University of Bath, Bath, BA2 7AY, U.K.
mdv@cs.bath.ac.uk

[2] Faculty of Engineering and Natural Sciences,
Sabanci University, Orhanli, Tuzla, Istanbul 34956, Turkey
dgkisa@su.sabanciuniv.edu

[3] Institut für Informationssysteme 184/3,
Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria
{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 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 to group 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, and can help to eliminate sources of common programming errors by realising syntax checking or code completion features. 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.

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 be classified into basically three categories: (i) theoretical foundations of ASP including language extensions, (ii) work on ASP solver technology, and (iii) case studies and applications involving ASP. More recently, methods and methodologies to support programmers during development of answer-set programs are increasingly becoming a focus of research interest (De Vos and Schaub 2007; 2009; Oetsch, Pührer, and Tompits 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 marked 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. Different notions of modularity have been proposed for ASP in the literature already (Bugliesi, Lamma, and Mello 1994; Eiter, Gottlob, and Veith 1997; Gelfond and Gabaldon 1999; Balduccini 2007; Janhunen et al. 2009). However, 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 which can improve the clarity and coherence of the program parts and which 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 can 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

*This work was partially supported by the Austrian Science Fund (FWF) under grant P21698.

term arguments of a predicate using respective language features for declaring types. This information can be used for automatic 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 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 the annotation language LANA 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 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, 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. The tool to automatically transform annotated comments into an HTML documentation is described in Section 4, and a tool for running unit tests is described in Section 5. Finally, we review related work in Section 6 and conclude in Section 7.

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 name just a few. In this paper, 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*, `not`. A *literal* is an atom a or a negated atom $\texttt{not}\ a$. We say that $\texttt{not}\ a$ is true if we cannot find evidence supporting the truth of a. Atoms and literals are used to create rules of the general form

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

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 should be able to satisfy the body. A (*normal*) *program* is a set of rules. The semantics is defined in terms of *answer sets*, i.e., assignments of truth values "true" and "false" to all atoms in the program that satisfy the rules in a minimal and consistent fashion, taking into account that truth of an atom cannot be based on the absence of proof (i.e., the truth of an atom cannot indirectly be inferred by its own negation). 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, syntax sugar. On the other hand, we have semantic extensions that generalise the formalism itself.

From a programmer's perspective, *choice rules* (Niemelä, Simons, and Soininen 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 (Gelfond and Lifschitz 1991) in the head of rules. When the body of the rule is true, we need to have at least one head atom that is true. Although 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, Lierler, and Maratea 2004), a solver based on translating the program to a SAT problem.

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

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

Table 1: Overview of LANA based on BNF.

Language Element	Definition	Informal Description
block element	*block* | *atom* | *term* | *input signature* | *output signature* | *precondition* | *postcondition*	The elements of LANA 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 used in a block where *termList* is the list of the predicate's arguments.
term	"`@term`" *name* [*description*] [*type*]	Declares a term, the name comes from some *atom* term list, its meaning and type information.
type	"`@from`" *groundTerms* | "`@with`" *ruleBody* | "`@samerangeas`" *term*	Type of a term is defined either by a list of ground terms, the terms satisfying a specified rule body, or is equal to the type of an already declared term.
input signature	"`@input`" *inputPrds*	Declares input predicates of a block as a list of name/arity pairs.
output signature	"`@output`" *outputPrds*	Declares output predicates of a block as a list of name/arity pairs.
precondition	"`@precon`" *name* "{" [*description*] {"`@always`" | "`@never`", *atomList*} [*embedded ASP code*] "}"	States a logical condition for the inputs of a block. An input is admissible if all the atoms after "`@always`" are true and the atoms next to "`@never`" are false in in all answer sets of the input joined with the embedded ASP code (i.e., ASP code within the annotation comment environment).
postcondition	"`@postcon`" *name* "{" [*description*] {"`@always`" | "`@never`", *atomList*} [*embedded ASP code*] "}"	States a logical condition for the answer sets of a block. The postcondition is satisfied if the atoms after "`@always`" are true, and the atoms next to "`@never`" are false in all the answer sets of the rules of the block, the embedded ASP code, and any admissible input.
test case	"`@testcase`" *name* [*description*] "`@scope`" *blockNames* *test condition* [*ASP code*]	Specifies a test case for the blocks mentioned in *blockNames*. The test case passes if the blocks joined with *ASP code* satisfy all specified test conditions.
test condition	"`@testhasanswerset`" | "`@testnoanswerset`" | "`@testatoms`" *atomList mode*	A test condition has one of the listed forms. It holds if at least one answer set is found, no answer set is found, or all ground atoms mentioned in *atomList* 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.

3 An Annotation Language for ASP

In this section, we describe our annotation language LANA in more detail. A summary of the language elements of LANA appears in Table 1. 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-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). Initially, some squares may show parts of a ship or water to provide hints. Moreover, a number besides each row and each column is initially given and 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 Battleship 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` respectively determine the numbers associated with each row and each column. Solutions are represented by facts `ship(W,X,Y,Z)`, expressing that a ship is occupying position `(W,X)` to `(Y,Z)`.

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

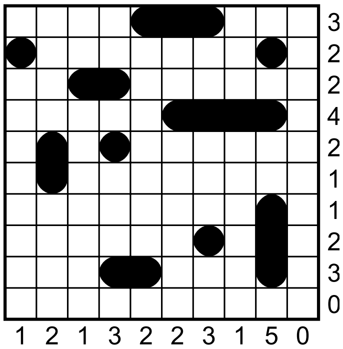


Figure 1: Solved instance of a Battleship puzzle.

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 `%`, providing a solver supports this feature. However, to distinguish annotations from ordinary (block) comments, an additional star `*` is always placed after `%*` at the beginning.

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 are used to represent the input for the block and which ones correspond to the output. For input and output signatures, we also have to explicitly provide the arity of the involved predicates. This is needed to disambiguate between predicates with same names but a different number of arguments, as `ship` in our running example. Note that annotations are optional, declarations are not enforced. However, the more information is made explicit, the more it can be used to the benefit of the developer by tools that interpret such meta-information.

Regarding the declaration of predicates, we can even 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 somewhere in 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).
```

Intuitively, 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 already known ones, 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 be directly used to specify the types of variables.

As mentioned already, 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.

Towards testing and verification of the program, LANA allows the formulation of 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. For illustration, we formulate the precondition of our Battleship encoding that no square contains both water and parts of a ship jointly as follows:

```
%** @precon Excl {
 no square contains 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 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 in any solution, 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 if a block passed 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 valuable 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.

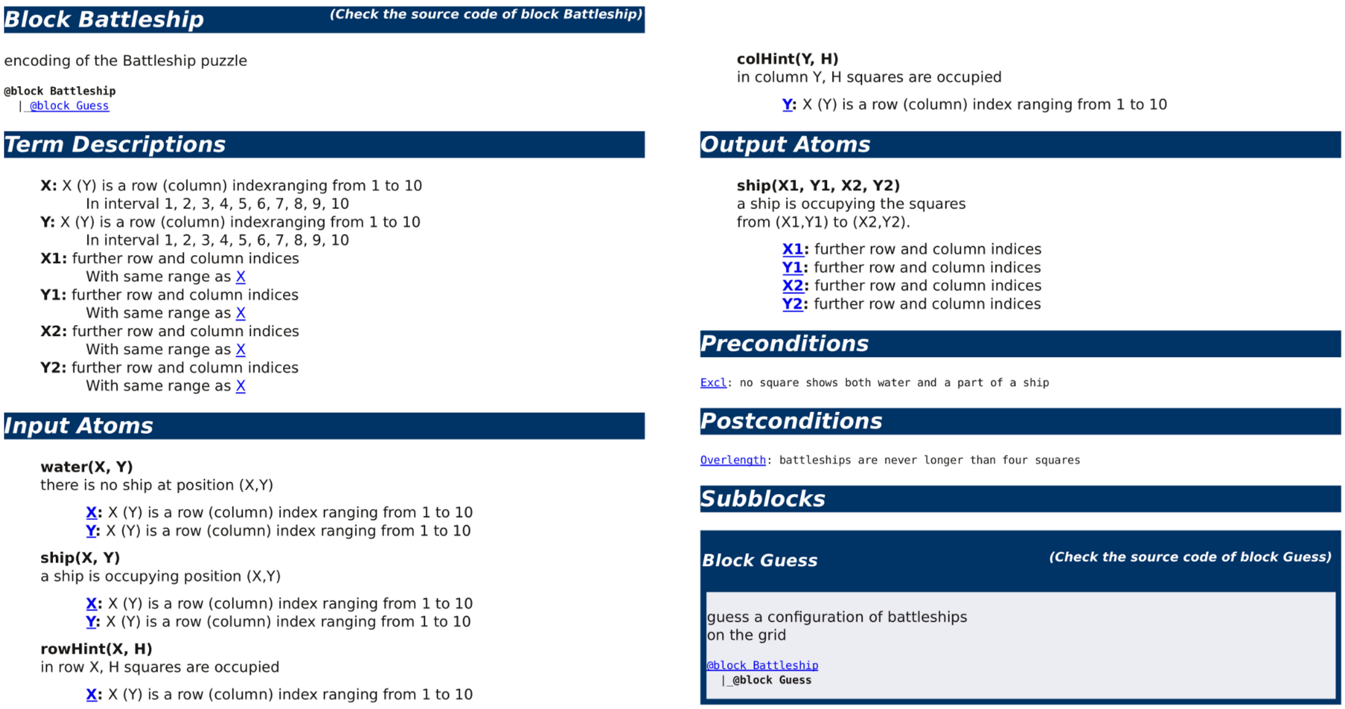
Though pre- and postconditions allow to partially verify program correctness, LANA also supports a light-weight form of program validation that is inspired by unit testing. A unit test in procedural languages is commonly a test for an individual function or procedure. While in a related approach for 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 if 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 four cells long 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 a least one answer set (`@trueinatleast 1`) of block `Guess` joined with the subsequent rule that defines `goalShip`. Instead, 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 sets 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
```



Block Battleship
(Check the source code of block Battleship)

encoding of the Battleship puzzle

@block Battleship
| @block Guess

Term Descriptions

X: X (Y) is a row (column) indexranging from 1 to 10
In interval 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Y: X (Y) is a row (column) indexranging from 1 to 10
In interval 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
X1: further row and column indices
With same range as X
Y1: further row and column indices
With same range as X
X2: further row and column indices
With same range as X
Y2: further row and column indices
With same range as X

Input Atoms

water(X, Y)
there is no ship at position (X,Y)

X: X (Y) is a row (column) index ranging from 1 to 10
Y: X (Y) is a row (column) index ranging from 1 to 10

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

X: X (Y) is a row (column) index ranging from 1 to 10
Y: X (Y) is a row (column) index ranging from 1 to 10

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

X: X (Y) is a row (column) index ranging from 1 to 10

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

Y: X (Y) is a row (column) index ranging from 1 to 10

Output Atoms

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

X1: further row and column indices
Y1: further row and column indices
X2: further row and column indices
Y2: further row and column indices

Preconditions

Excl: no square shows both water and a part of a ship

Postconditions

Overlength: battleships are never longer than four squares

Subblocks

Block Guess (Check the source code of block Guess)

guess a configuration of battleships
on the grid

@block Battleship
| @block Guess

Figure 2: HTML documentation of the Battleship program.

```
 @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 if the new test case and the old ones pass.

4 ASPDoc

ASPDoc is a command-line tool which 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, atoms, etc. is well-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`.

[3] `http://students.sabanciuniv.edu/dgkisa/aspdoc-aspunit`.

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. Help on available options can be obtained with the option `-h`.

A screenshot of the documentation document for the Battleship encoding interpreted by some standard web browser is given in Figure 2. The document contains descriptions of all the blocks of the answer-set program, where sub-blocks are indented relative to their parent blocks. Also, to provide an overview, a summary of the block structure of the entire answer-set program is 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 with predicates mentioned neither in a block's input nor in its output signature, are displayed as well in a dedicated section. 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` in the term description section.

5 ASPUNIT

ASPUNIT is a tool to execute test cases that are formulated in LANA. Like ASPDOC, ASPUNIT 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. Though 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 solvers that are 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
```

That is, 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 has 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 then 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 the standard-output device. This report contains information regarding success or failure for each test case. If a test case fails, a counter example 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.

6 Related Work

With larger programs for real-world applications being written using ASP, 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, Pührer, and Tompits 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, Reale, and Ricca (2011). Debugging in ASP is supported by SPOCK (Gebser et al. 2009). It makes use of ASP to explain and

```
Test Case ShipTopLeftCorner: Successful

Test Case NoDiagonalShips  : Successful

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

  Failed Test : @falseinall forbiddenShip
    Counter example:
      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).
```

Figure 3: A test report for the Battleship program.

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 provide a mechanism for debugging code.

To support the development of large applications, traditional languages offer programming tools that automatically generate searchable documentation, like e.g., JAVADOC. Methodologies like test-driven development provide developers mechanisms to incrementally unit test their code and support regression testing. JUNIT is an example of this for Java. LANA provides 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 used in Prolog (Kulas 2000).

Similar to our unit testing approach, Prolog offers unit-testing functionality called PLUNIT.[4] 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 testsuite, tests in PLUNIT are written in Prolog. Febbraro et al. (2011) provide a mechanism for unit testing in ASP which is incorporated in their IDE ASPIDE. They base their unit tests on clusters on the dependency graphs or rule labelling, while we allow the programmer to decide which rules belong to a test by defining blocks.

Related to our approach for annotating type information is RSIG (Balduccini 2007). While RSIG really is an language extension for specifying simple type information for programs and modules and thus requires its own parser, programs that contain LANA code can be used with any ASP solver since annotations take the form of program comments.

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 post-conditions, 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 as they are quite popular in, e.g., extreme programming.

While lots of 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 functionality is available within an IDE for ASP. The two most actively developed IDEs for ASP at present are SEALION (Oetsch, Pührer, and Tompits 2011) and ASPIDE (Febbraro, Reale, and Ricca 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. LANA is currently integrated into SEALION, and the features of ASPDOC are already available from within the IDE. 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 answer-set 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.

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

References

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

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

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.

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 24th International Conference on Logic Programming* (*ICLP 2008*), volume 5366 of *LNCS*, 724–728. Springer.

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.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. The KR system DLV: Progress report, comparisons and benchmarks. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 1998*). Morgan Kaufmann. 406–417.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2002. The DLV^k planning system: Progress report. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence* (*JELIA 2002*), volume 2424 of *LNCS*, 541–544. Springer.

Eiter, T.; Gottlob, G.; and Veith, H. 1997. Modular logic programming and generalized quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 1997*), volume 1265 of *LNCS*, 290–309. Springer.

Febbraro, O.; Leone, N.; Reale, K.; and Ricca, F. 2011. Unit testing in ASPIDE. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (*INAP 2011*) *and the 25th Workshop on Logic Programming* (*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 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), volume 6645 of *LNCS*, 317–330. Springer.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (*IJCAI 2007*), 386–392. AAAI Press/The MIT Press.

Gebser, M.; Pührer, J.; Schaub, T.; Tompits, H.; and Woltran, S. 2009. spock: A debugging support tool for logic programs under the answer-set semantics. In *Proceedings of the 17th International Conference Applications of Declarative Programming and Knowledge Management* (*INAP 2007*) *and the 21st Workshop on Logic Programming* (*WLP 2007*)*, Revised Selected Papers*, volume 5437 of *LNCS*, 247–252. Springer.

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*, 1070–1080. MIT Press.

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 19th National Conference on Artificial Intelligence* (*AAAI 2004*) *and the 16th Conference on Innovative Applications of Artificial Intelligence* (*IAAI 2004*), 61–66. AAAI Press / The MIT Press.

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

Janhunen, T.; Niemelä, I.; Oetsch, J.; Pührer, J.; and Tompits, H. 2010. On testing answer-set programs. In *Proceedings of the 19th European Conference on Artificial Intelligence* (*ECAI 2010*), volume 215 of *Frontiers in Artificial Intelligence and Applications*, 951–956. IOS Press.

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 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), volume 6645 of *LNCS*, 242–247. Springer.

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 19th International Conference on Applications of Declarative Programming and Knowledge Management* (*INAP 2011*) *and the 25th Workshop on Logic Programming* (*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*), volume 1817 of *LNCS*, 234–254. Springer.

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. *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 4th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 1997*), volume 1265 of *LNCS*, 420–429. Springer.

Niemelä, I.; Simons, P.; and Soininen, T. 1999. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 1999*), volume 1730 of *LNCS*, 317–331. Springer.

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 13th International Conference on Principles of Knowledge Representation and Reasoning* (*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*), volume 7 of *LIPIcs*, 154–161. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Oetsch, J.; Pührer, J.; and Tompits, H. 2011. The SeaLion has landed: An IDE for answer-set programming—Preliminary report. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (*INAP 2011*) *and the 25th Workshop on Logic Programming* (*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. In 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 1st International Workshop on Practical Aspects of Declarative Languages* (*PADL 1998*), volume 1551 of *LNCS*, 305–319. Springer.

Sureshkumar, A.; De Vos, M.; Brain, M.; and Fitch, J. 2007. APE: An AnsProlog* environment. In De Vos and Schaub (2007), 101–115.