A Pragmatic Programmer's Guide to Answer Set Programming

Martin Brain, Owen Cliffe* and Marina De Vos*

Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
{mjb,occ,mdv}@cs.bath.ac.uk

Abstract. With the increasing speed and capacity of answer set solvers and showcase applications in a variety of fields, Answer Set Programming (ASP) is maturing as a programming paradigm for declarative problem solving. Comprehensive programming methodologies have been developed for procedural and object-oriented paradigms to assist programmers in developing their programs from the problem specification. In many cases, however it is not clear how, or even if, such methodologies can be applied to answer set programming. In this paper, we present a first and rather pragmatic methodology for ASP and illustrate our approach through the encoding of graphical puzzle.

1 Introduction

Answer Set Programming (ASP) is a declarative programming paradigm based on the answer set semantics [16, 1]. Like other declarative programming languages, the programmer specifies *what* needs to be achieved, rather than *how* it should be achieved. It therefore lends itself naturally to applications in the domain of artificial intelligence, such as plan generation and reasoning in agents. In ASP, programs are written in *AnsProlog* and describe the requirements for the solutions of certain problem. The answer sets of the program are interpreted to give theses solutions. The possible answer sets for an *AnsProlog* input program are computed with a program called a solver. Current solvers include SMODELS [19, 21], DLV [11, 12], CLASP [14], CMODELS [17], SUP [18].

A report by the Working group on Answer Set Programming (WASP)[1] acknowledges that better tools are required to support programming in this paradigm [20]. However in order to identify the aspects that require better support, and thus develop the appropriate tools to support them, a better understanding of the programming process is needed.

The process of engineering solutions to problems in declarative languages differs from conventional procedural software engineering in many regards. Conventional software engineering approaches (e.g. UML) (but excluding more recent agile approaches) focus on building specifications of data structures and functional units in advance, before proceeding to their implementation. However the declarative approach used in

* Work supported by the ALIVE project (FP7 215890)

[1] http://wasp.unime.it/

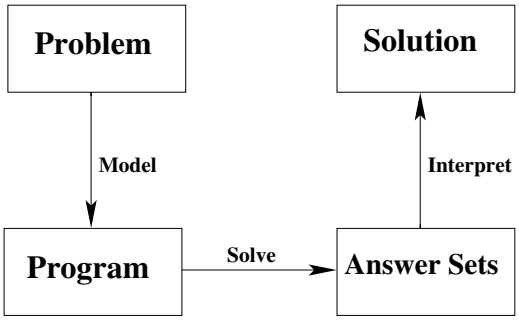

Fig. 1. The Four Box Diagram

AnsProlog means that programs essentially act as their own specification. Thus the process of understanding, decomposing and encoding problem structures (the specification) is necessarily blurred with the process of solving the problem itself (the implementation).

AnsProlog is increasingly being used to solve practical problems both in and out of the academic domain. At present it is our experience that developers who use these systems to solve real problems tend to develop either without a specific methodology, or build their own methodological process. Just as the community is trying to reach consensus on language standards [22] and intermediate formats for program representation and such [15, 22], we feel there is a need to reach a similar consensus on practical processes by which programs are developed and maintained. However, as with all programing techniques we cannot claim to know *the best* way to solve problems using ASP, all we can talk about is how *we* write programs and relate our experiences working in a number of domains. Thus, in this paper we do not try to document "best practice", simply "our practice". We hope that this can be the start of a discussion and that we, as a community, can start building up an idea of best practice.

The structure of the remainder of paper is as follows, we start by addressing issues relating to the whole process of problem solving using ASP, specifically we focus on methodological questions relating to how problems are captured. We then discuss some specific observations relating to problem encodings themselves through a guided example. Finally we look at the process of resolving problems within existing *AnsProlog* programs (debugging).

2 Towards a Development Process

ASP can be described via the four box diagram, as shown in Figure 1. One starts with a problem which is modelled as a *AnsProlog* program. This program is passed then to a solver. The answer sets are then interpreted to obtain the solutions.

A common problem we have found when encouraging students and collaborators from other fields to become active involved in developing applications using ASP is that while they understand how the tools work (the solving link in Figure 1) and finished

models (the program box), they do not understand how to go from their problem to a program (the modelling link). As far as we know there is no documentation that we can point them to and say "this is how to solve a problem using ASP". The authors of [8] give a very nice break down of how a logical model is structured and is currently the best resource for this. However they present the models as a finished product and do not discuss the process, reasoning or tools that created them.

3 The Methodology

In this section we provide a detailed description of our methodology for using ASP to solve problems. As a running example we will use the Hashiwokakero puzzle. The puzzles creators, Nikoli[2] describe the puzzle as follows:

> Hashiwokakero is played on a rectangular grid with no standard size, although the grid itself is not usually drawn. Some cells start out with (usually encircled) numbers from 1 to 8 inclusive; these are the islands. The rest of the cells are empty.
>
> The goal is to connect all of the islands into a single connected group by drawing a series of bridges between the islands. The bridges must follow certain criteria:
>
> 1. Connect islands (the dots with numbers) with as many bridges as the number in the island.
> 2. There can be no more than two bridges between two islands.
> 3. Bridges cannot go across islands or other bridges.
> 4. The bridges will form a continuous link between all the islands.

We also use examples from our music composition system ANTON [3] and our superoptimisation application, TOAST [5] which are among the largest applications using ASP.

Although some debugging tools exist [4], we have come to believe that a more incremental, test-driven [2] approach allows for easy verification at every stage. While it does not make debugging tools superfluous, a systematic approach prevents programmers from making certain errors and increases productivity.

3.1 Step 1: Start Simple

We advocate starting with a simplified model of the problem that to be solved, because it is much easier to build up a correct model into something more complicated than it is to fix a complicated but broken program. For example, in the case of ANTON we started out composing one part for 8 notes. This point cannot be stressed enough, start simple, start laughably simple, and work upwards.

[2] http://www.nikoli.com/en/puzzles/hashiwokakero/rule.html

Example 1. To start encoding our puzzle, we need to decide which literals to use to represent each concept. We need to consider what information will be in the instances, what the constraints are and what information you wish to infer. In this case, we decided to use the following:

Atom	Concept represented
Instance Specific Information	
`col(X)`	There is a column labelled X (ascending integers from 1).
`row(Y)`	There is a row labelled Y (ascending integers from 1).
`island(X,Y,N)`	There is an island in column X, row Y with value N.
Information to be Inferred	
`singleHorizontal(X,Y)`	There is a single horizontal bridge in column X, row Y.
`doubleHorizontal(X,Y)`	There is a double horizontal bridge in column X, row Y.
`singleVertical(X,Y)`	There is a single vertical bridge in column X, row Y.
`doubleVertical(X,Y)`	There is a double vertical bridge in column X, row Y.

At this stage, our program now looks like as Listing 1.

```
row(1..height).
col(1..width).
```

Listing 1. The first step towards our Hashiwokakero program

Deciding the meaning of the base literals should be enough to create and visualise (see next step) a problem instance. It is best to start with as small an instance as is meaningful, as otherwise it will be difficult to check by hand. This also helps mitigate any scaling issues during development. Given we are advocating incremental development, one does not want to have to wait more than a few seconds per run during development, so one needs to pick an appropriately sized example.

Listing 2 shows (with modified formatting) an instance of our puzzle.

3.2 Step 2: Visualisation

The next step is a visualisation or post-processing mechanism. This is effectively the interpretation link in the four box diagram. Post-processing and visualisation are oft-neglected parts of the development process, but very important ones as they allow the developer to see the program and its development in terms of the initial problem and solution, allowing a much more intuitive development process. This stage closes the semantic gap between the program encoding and the problem domain and aids debugging as it allows programmers to see what they wrote and easily compare it with what they intended to write. As argued in [6], when writing programs there is often a mismatch between the answer set you get and what you expected. Visualisation makes it much easier to see what one has, and specifically it often makes it easier to see *when* what one

```
#const height=13.
#const width=13.

uniqueStart(1,1).

island(1,1,2).   island(3,1,4).   island(5,1,3).
island(1,4,2).   island(3,4,3).   island(5,3,2).
island(1,6,1).   island(3,6,5).   island(5,5,2).
island(1,10,2).  island(3,8,4).   island(5,8,4).
island(1,13,3).  island(3,10,2).  island(5,10,3).
                 island(3,12,1).  island(5,12,2).

island(6,4,2).   island(7,1,1).   island(8,6,1).   island(9,1,2).
island(6,6,2).   island(7,3,3).   island(8,8,3).   island(9,3,2).
island(6,11,2).  island(7,5,5).   island(8,11,4).  island(9,5,3).
island(6,13,3).  island(7,7,2).   island(8,13,1).  island(9,7,2).
                                                   island(9,10,3).

island(10,2,3).  island(11,5,4).  island(12,1,1).  island(13,2,1).
island(10,4,3).  island(11,7,4).  island(12,4,1).  island(13,7,2).
island(10,11,4). island(11,10,2). island(12,6,2).  island(13,10,3).
island(10,13,2).                  island(12,8,3).  island(13,13,2).
                                  island(12,11,3).
```

Listing 2. An Hashiwokakero instance

has is incorrect. How the visualisations/interpretations are produced is up to the programmer and different types of program will lend themselves to different visualisation and processing approaches. One possibility is using a scripting language (e.g. Perl) and ASCII art. A more sophisticated choice would be ASPVIZ [9], a ASP visualisation tool using *AnsProlog* as its representation language. GraphViz [3] is also frequently a useful tool for visualising problems solutions.

The utility of visualisation cannot be overstated. It is often the most time consuming part of development but it does pay dividends by simplifying the process of isolating programming errors. After this it should be possible to visualise all of the elements of the full search space.

Example 2. The visualisation program written in ASPVIZ for Hashiwokakero can be found in Listing 3 (see [9] for a description of the language). The rules produce graphical artifacts for each island, shown as an ellipse containing the number of required bridges and the connecting bridges. An example of the output, for the program and instance given so far can be found in Figure 2.

3.3 Step 3: The Search Space Generator

The first part of the program to be written should be the search space generator. This tends to be a set of choice rules that define the search space of the problem in question[4]; to start with, this should give an answer set for each possible element of the search space. In the case when the problem being solved is co-NP, search generation is especially important. Working with programs with no answer sets is problematic. This should be obvious if one accepts the argument [6] that one knows there is a bug if there

[3] http://www.graphviz.org/

[4] We are of the opinion that all uses of ASP should effectively be phrased as search problems. As a consequence, we see the 'hard' part of the problem as the search phase. We note that there is a view that the procedural, data processing side is more significant.

```
% Draw islands and numbers
draw_ellipse(dflb,p(X*16,Y*16),14,14) :- island(X,Y,N).
draw_text(dflf,c,c,p(X*16,Y*16),N):- island(X,Y,N).

% Draw single bridges
draw_line(dflb,p(X*16,(Y*16)-8),p(X*16,(Y*16)+8)):-
                    verticalBridge(X,Y), not doubleVertical(X,Y).
draw_line(dflb,p((X*16)-8,Y*16),p((X*16)+8,Y*16)) :-
                    horizontalBridge(X,Y), not doubleHorizontal(X,Y).

% Draw double bridges
draw_line(dflb,p((X*16)-8,Y*16+2),p((X*16)+8,Y*16+2)) :- doubleHorizontal(X,Y).
draw_line(dflb,p((X*16)-8,Y*16-2),p((X*16)+8,Y*16-2)) :- doubleHorizontal(X,Y).

draw_line(dflb,p(X*16+2,(Y*16)-8),p(X*16+2,(Y*16)+8)):- doubleVertical(X,Y).
draw_line(dflb,p(X*16-2,(Y*16)-8),p(X*16-2,(Y*16)+8)):- doubleVertical(X,Y).
```

Listing 3. The visualisation program for Hashiwokakero

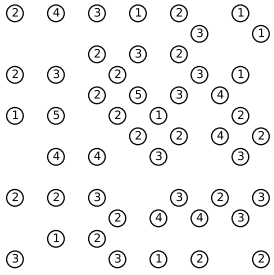

Fig. 2. Visualisation of an unsolved instance of the Hashiwokakero puzzle

is a difference between the expected and given answers. When the expected result is that no answer sets are produced and one gets no answer sets, how does one know if the code is working correctly or whether one has introduced a contradiction? Just using the number of branches or other solver performance stats is not reliable. Thus one wants to spend as much time as possible *with* answer sets, to the point of commenting out some constraints or deliberately working with instances that are 'incorrect'.

Example 3. We now need to add rules to generate the search space. The program up to this stage can be found in Listing 4. Adding these first rules allows us to see the solution taking shape as we build the model.

We now run our program together with the instance provided in Listing 2. From this, we obtain an answer set for every combination of bridges (a very large number). At this point, we do not yet care whether bridges join up or not. We can exploit the randomisation functionality of most answer set solvers to generate a small selection of answer sets which are visualised in Figure 3.

3.4 Commenting

Comments are very important during development. *AnsProlog* is very expressive and allows modelling of some very sophisticated concepts with very little code. Without

```
row(1..height).
col(1..width).

%% We need to know which squares contain island
%% So we project out the value of the island
isIsland(X,Y) :- island(X,Y,N).

%% For each square that is not an island, pick whether it should contain
%% a bridge or not and if so, what kind of bridge.
1 { empty(X,Y), singleHorizontal(X,Y), doubleHorizontal(X,Y),
    singleVertical(X,Y), doubleVertical(X,Y) } 1 :- row(Y), col(X),
                                                    not isIsland(X,Y).
```

Listing 4. Adding the search space for Hashiwokakero

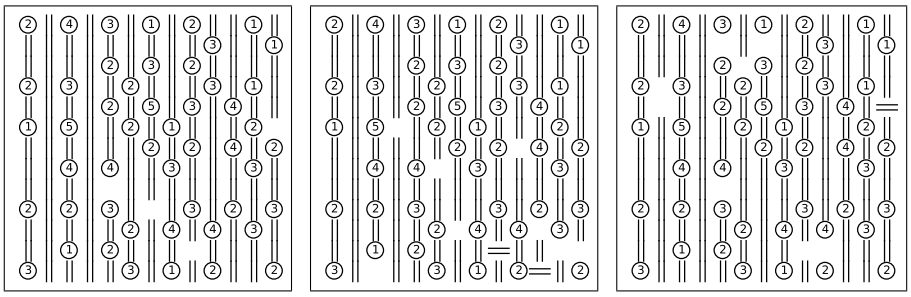

Fig. 3. Example outputs for random bridge assignments

comments this compactness can lead to terseness and unreadability (commonly referred to as write-only code in procedural languages). Normally each rule or group of related rules will be preceded by a comment, saying (in natural language) what aspect or idea from the problem it is intended to encode and (in the few cases where the encoding is complicated enough), how it is encoded. As well as making the program more intelligible, heavy commenting is very important for debugging and maintenance as it expresses, as much as is possible, what was intended by the code. As noted in [6] a bug in an *AnsProlog* program is a mismatch between what is written and what was intended. When no expression of intent is available except for the rules, then only the programmer who wrote the code can debug it, and generally only while writing it. Maintenance and further development become very difficult as first one has to infer what the programmer was thinking when they wrote the code. Given the (frequently) small semantic gap between the comments and the code, natural language generation may offer an interesting route to program support/debugging tools. If what is written does not provide a correct description of the problem, then generally one can see which part of the description is a problem. We also tend to use comments to record what each of the key propositions presents about the world, i.e "move(T,X,Y) is known if the move at time step T is to position (X,Y)". Ideally it should be possible to read the comments top to bottom as a description of the model.

```
row(1..height).
col(1..width).

%% We need to know which squares contain island
%% So we project out the value of the island
isIsland(X,Y) :- island(X,Y,N).

%% For each square that is not an island, pick whether it should contain
%% a bridge or not and if so, what kind of bridge.
1 { empty(X,Y), singleHorizontal(X,Y), doubleHorizontal(X,Y),
    singleVertical(X,Y), doubleVertical(X,Y) } 1 :- row(Y), col(X),
                                                    not isIsland(X,Y).

%% A single horizontal bridge must lead to another single bridge or an island
 :- singleHorizontal(X,Y), not singleHorizontal(X+1,Y), not isIsland(X+1,Y).

%% Likewise for double horizontal bridges
 :- doubleHorizontal(X,Y), not doubleHorizontal(X+1,Y), not isIsland(X+1,Y).

%% Similarly for vertical bridges
 :- singleVertical(X,Y), not singleVertical(X,Y+1), not isIsland(X,Y+1).
 :- doubleVertical(X,Y), not doubleVertical(X,Y+1), not isIsland(X,Y+1).
```

Listing 5. The first increment of Hashiwokakero

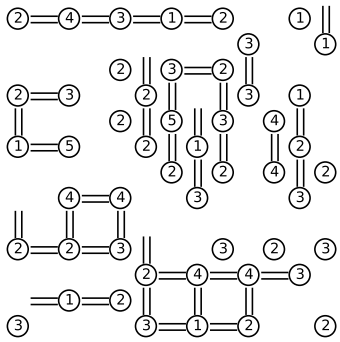

Fig. 4. An answer set visualisation of Hashiwokakero after the first increment.

3.5 Step 4: Incremental Development

Development now progresses in an incremental fashion, using very small increments. A set of rules (normally corresponding to one comment / idea) is written, then the solver is re-run and the answer set(s) visualised. This may seem slow but it removes a lot of bugs and saves time while trying to diagnose the causes of problems. In more complex applications the visualisation program may need to be updated along with the program to show the effect of new literals.

Example 4. Increment One: We now add rules saying that bridges must continue in their existing direction (and with the same width) or stop at an island. The program at this stage is shown in Listing 5. Note that these are only phrased in terms of increasing row / width as the decreasing case follows via symmetry. A visualisation of one the answer sets of our puzzle after we added the first increment is shown in Figure 4.

```
%% We want to be able to talk about either kind of horizontal / vertical bridge
horizontalBridge(X,Y) :- singleHorizontal(X,Y).
horizontalBridge(X,Y) :- doubleHorizontal(X,Y).
verticalBridge(X,Y) :- singleVertical(X,Y).
verticalBridge(X,Y) :- doubleVertical(X,Y).

%% A horizontal bridge cannot start a vertical bridge
 :- horizontalBridge(X,Y), verticalBridge(X,Y+1).

%% Neither can an empty square
 :- empty(X,Y), verticalBridge(X,Y+1).

%% Correspondingly for vertical bridges
 :- verticalBridge(X,Y), horizontalBridge(X+1,Y).
 :- empty(X,Y), horizontalBridge(X+1,Y).
```

Listing 6. The second increment of Hashiwokakero

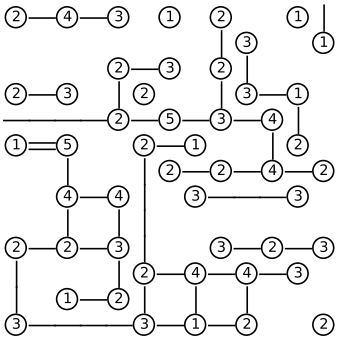

Fig. 5. An answer set visualisation of Hashiwokakero after the second increment.

Example 5. Increment Two: From the visualisation in Figure 4 is is clear that answer sets are beginning to resemble solutions but a few conditions are missing as there are cases where bridges are running into each other or starting from nothing. Thus we project the existence of horizontal / vertical bridges and prevent the other kind from starting immediately after them. Similarly for empty squares. The projection is a trade off between the number of literals and the number of rules. Given that rules are more often the limit of scalability, it is often worth doing. The rules added to the program are shown in Listing 6 and its output in Figure 5.

Example 6. Increment Three: The bridges are beginning to look correct, however the visualisation (Figure 5) points out a few conditions that have been missed. As we have been focusing on the 'next' location, we have edge conditions at the border. These are easily removed as there should never be vertical bridges in the top or bottom row, nor horizontal ones in either edge. The code in Listing 7 is added to our program which results in output like Figure 6.

Example 7. Increment Four: The bridges look correct but are a little sparse, so the next condition to add is the number of bridges coming in to each island. The obvious way to

```
%% No vertical bridges in the top or bottom rows
 :- verticalBridge(X,1).
 :- verticalBridge(X,height).

%% Likewise no horizontal bridges in the edge columns.
 :- horizontalBridge(1,Y).
 :- horizontalBridge(width,Y).
```

Listing 7. The third increment of Hashiwokakero

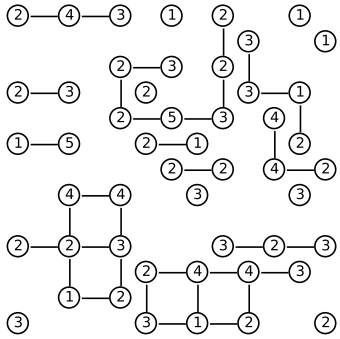

Fig. 6. An answer set visualisation of Hashiwokakero after the third increment.

do this is using a weight constraint. The additions to the program are shown in Listing 8. Again, the visualisation (Figure 7) makes it relatively easy to check that the effect of these rules matches the intend behind them.

Example 8. Fifth and final Increment: Now for the last constraint: all of the islands need to be connected. Careful examination of the output (Figure 7) will show that the islands are not connected. We formalise connectivity as reachability from a unique starting point and then require that all islands must be reachable. This raises the question of how to pick a unique starting point. We opt for the easiest solution of adding this to the instance requirements. The final addition to our program is shown in Listing 9. The entire program resulting in a complete (and unique) solution (Figure 8)

3.6 Coding Conventions

A number of coding conventions have been found to be useful and practical. Firstly literals are only ever used with one arity. This means that omissions of variables should be a detectable problem, rather than silently changing the meaning of the programs. Likewise, where at all possible, variable names are only ever used for one domain, i.e. `T` will always be a quantification over the `time()` domain, if `X` is a distance, it will always be used as one, in the same direction. These meanings are program specific, the key point is that it should be consistent across the program. Likewise the position, and ordering of variables should always be the same. If several literals refer to a 2D position at a given time step then they will all start with `(T,X,Y,...)`, etc.

```
%% For an island to be correct connected the number of bridges
%% entering a island must equal the weight at the island.
correctlyConnected(X,Y) :- island(X,Y,N),
                           N [ singleHorizontal(X-1,Y) = 1,
                               doubleHorizontal(X-1,Y) = 2,
                               singleHorizontal(X+1,Y) = 1,
                               doubleHorizontal(X+1,Y) = 2,
                               singleVertical(X,Y-1) = 1,
                               doubleVertical(X,Y-1) = 2,
                               singleVertical(X,Y+1) = 1,
                               doubleVertical(X,Y+1) = 2 ] N.

%% All islands must be correctly connected
 :- isIsland(X,Y), not correctlyConnected(X,Y).
```

Listing 8. The fourth increment of Hashiwokakero

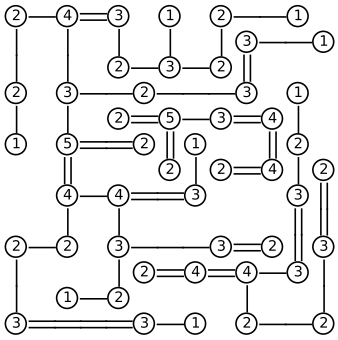

Fig. 7. An answer set visualisation of Hashiwokakero after the fourth increment.

3.7 Extension and Maintenance

Once the program is working, development shifts to extension and maintenance. While previously we were intentionally cutting down the space of possibilities, in the maintenance and enhancement phase the aim is often to make changes without inadvertently removing portions of the solution space (or making the program inconsistent, i.e. removing all of the solutions). Regression testing was used during the development of ANTON to address this problem. If we view ASP as solving a search problem, then the solution is a point in the search space and can be used as an initial condition for the search. If it gives the (single) solution that was found originally then it is still a solution, if it is inconsistent then the space of solutions has decreased, indicating a possible error. Once the original composition system was working, its output was checked by an expert and a collection of valid (and respectively, invalid) pieces was created. After each step of the development (one or more new rules), the regression tests could be run to automatically check that each of these was/was not obtainable and thus that no bugs had been introduced. This helped isolate bugs early in the development cycle and add a considerable degree of certainty to the development process.

```
%% The unique start is reachable
reachable(X,Y) :- uniqueStart(X,Y).

%% If an island is reachable then all neighbouring bridges are reachable
reachable(X-1,Y) :- isIsland(X,Y), reachable(X,Y), horizontalBridge(X-1,Y).
reachable(X+1,Y) :- isIsland(X,Y), reachable(X,Y), horizontalBridge(X+1,Y).
reachable(X,Y-1) :- isIsland(X,Y), reachable(X,Y), verticalBridge(X,Y-1).
reachable(X,Y+1) :- isIsland(X,Y), reachable(X,Y), verticalBridge(X,Y+1).

%% If a horizontal bridge is reachable then so are neighbouring bridges/islands
reachable(X-1,Y) :- horizontalBridge(X,Y), reachable(X,Y), horizontalBridge(X-1,Y).
reachable(X+1,Y) :- horizontalBridge(X,Y), reachable(X,Y), horizontalBridge(X+1,Y).
reachable(X-1,Y) :- horizontalBridge(X,Y), reachable(X,Y), isIsland(X-1,Y).
reachable(X+1,Y) :- horizontalBridge(X,Y), reachable(X,Y), isIsland(X+1,Y).

%% Likewise vertical bridges
reachable(X,Y-1) :- verticalBridge(X,Y), reachable(X,Y), verticalBridge(X,Y-1).
reachable(X,Y+1) :- verticalBridge(X,Y), reachable(X,Y), verticalBridge(X,Y+1).
reachable(X,Y-1) :- verticalBridge(X,Y), reachable(X,Y), isIsland(X,Y-1).
reachable(X,Y+1) :- verticalBridge(X,Y), reachable(X,Y), isIsland(X,Y+1).

%% Every island must be reachable
 :- isIsland(X,Y), not reachable(X,Y).
```

Listing 9. The final additions to Hashiwokakero

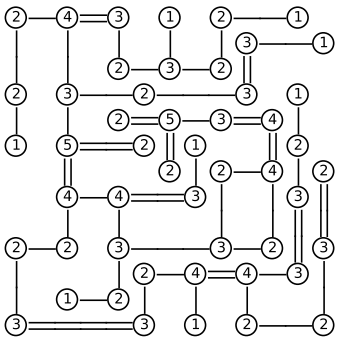

Fig. 8. The visualisation of the solution to the Hashiwokakero.

3.8 Generating Program Instances and Encodings

Literature on ASP often refers to splitting programs into instances and encoding. This is an intuitive division because in many cases there is a general class of problems and a particular instance of the problem we wish to solve. However there is often an assumption that this split is syntactic; facts for the instance and rules for the encoding. From our experience, this is not always the case. It is perfectly possible for the instance to include constants, extra constraints as well as atoms. Constants tend to be the most significant because these often alter the remaining program significantly.

In the case where the instance encoding is non-trivial we normally write a script that takes obvious, human readable parameters (as in ANTON) or a problem specific input format (as in TOAST) and generates at least the instance if not the whole program. This is where the distinction between instance and encoding begins to break down, as

part of the reason for having these scripts is combining the necessary program fragments, which are often instance specific. ANTON includes a `#include` construct to manage dependencies between program fragments and simplify program generation. In TOAST , the search program component included the architecture file which could be considered both instance and encoding. If one thinks of the visualisation scripts as the interpretation arrow in the four-box diagram Figure 1, then the program generation script that puts together the *AnsProlog* program is the representation arrow.

4 Debugging Programs

While the current debugging tools [6, 4] have their use, we have come to believe that existing tools do not yet provide the necessary support for programmers. The main problem is the amount of time the interface takes because one has to (implicitly or explicitly) mark which parts of the program are right and wrong. In practise this is too slow. One may think that it is no coincidence that none of the papers on debugging give more than a trivial, propositional example. All of the existing debuggers are primarily focused on computation of the reasons why certain properties of the answer sets hold. A topic that has received little research or implementation attention is the question of how to present the resultant symbolic information back to the user. This problem is discussed in more detail in [7].

However, with careful, incremental development and computing and visualising models after each change the programmer will already have a rough idea about what caused the problem and probably also why. The regression tests should catch the more obscure cases the programmer did not necessarily think of when going back and changing an older definitions.

5 Conclusion

In this paper we have presented an anecdotal methodology for software development in answer set programming. In summary:

1. Our approach is incremental and test-driven, allowing for early error detection.
2. After a suitable characterisation of the key concepts is determined, we recommend that a visualisation tool for the answer sets is used.
3. Having a graphical, auditory or more readable representation of the answer sets in terms of the problem domain will make it a lot easier to spot problems with the code.
4. Code documentation is even more important in ASP than it is in procedural languages. As errors are typically the result of a misalignment between the what was meant and what was written down, clearly stating what was intended in a human-readable form near to the relevant code makes identifying such errors easier.
5. The methodology is constructed around the notion that all our problems are search problems.
6. We model the entire search space first, before adding constraints to find acceptable solutions.

7. The characterisation of solutions and the constraints that reduce the search space should be done incrementally in order to detect bugs as early as possible.
8. Regression testing is used to verify that previous functionality is maintained after changes. A database containing good and bad solutions (cf. unit tests in procedural programming) can be constructed for this purpose.

While we find our approach to be useful in the context of our experience in developing complex *AnsProlog* systems, the methodology itself is informal and in no way comprehensive. The objective of this paper is to stimulate a discussion on possible best practice in the field.

One area which we have not considered is the ability to add assertions or 'sanity checks' that verify implicit properties of the encoding (for example the symmetric case of the constraints added in Listing 4). Even with the current state of the art in preprocessors [13], adding these as conventional constraints seems to bulk the program and result in much slower computation of solutions. Thus some technique for marking constraints as implicit, so that the encoding could be verified against them but they would be omitted from the normal program, could prove a useful development aid.

References

1. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
2. Kent Beck. *Test-driven Development*. Addison-Wesley, 2003.
3. Georg Boenn, Martin Brain, Marina De Vos, and John ffitch. Automatic Composition of Melodic and Harmonic Music by Answer Set Programming. In *Proceedings of ICLP08*, December 2008.
4. M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. "That is illogical captain!" – The debugging support tool spock for answer-set programs: System description. In De Vos and Schaub [10], pages 71–85.
5. Martin Brain, Tom Crick, Marina De Vos, and John Fitch. Toast: Applying answer set programming to superoptimisation. In *International Conference on Logic Programming*, LNCS. Springer, August 2006.
6. Martin Brain and Marina De Vos. Debugging Logic Programs under the Answer Set Semantics. In Marina De Vos and Alessandro Provetti, editors, *ASP05: Answer Set Programming: Advances in Theory and Implementation*, pages 142–152, Bath, UK, July 2005. Research Press International. Also available from http://CEUR-WS.org/Vol-142/files/page141.pdf.
7. Martin Brain and Marina De Vos. Answer set programming – a domain in need of explanation. In *Exact08: International Workshop on Explanation-aware Computing*, 2008.
8. M. Cayli, A. G. Karatop, E. Kavlak, H. Kaynar, F. Ture, and E. Erdem. Solving challenging grid puzzles with answer set programming. In *Proceedings of the 4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, pages 175–190, Porto, Portugal, September 2007.
9. Owen Cliffe, Marina De Vos, Martin Brain, and Julian Padget. Aspviz: Declarative visualisation and animation using answer set programming. In *Logic Programming*, Lecture Notes in Computer Science, pages 724–728. Springer Berlin / Heidelberg, 2008.
10. M. De Vos and T. Schaub, editors. *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, 2007.
11. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In Anthony G. Cohn,

Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.

12. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.
13. Niklas En and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT05*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
14. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007. Available at http://www.ijcai.org/papers07/contents.php.
15. Martin Gebser, Tomi Janhunen, Max Ostrowski, Torsten Schaub, and Sven Thiele. A versatile intermediate language for answer set programming. In Maurice Pagnucco and Michael Thielscher, editors, *Proceedings of the 12th International Workshop on Nonmonotonic Reasoning*, pages 150–159, Sydney, Australia, September 2008. University of New South Wales, School of Computer Science and Engineering, Techical Report, UNSW-CSE-TR-0819.
16. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, Seattle, Washington, August 1988. The MIT Press.
17. Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, pages 346–350. Springer, 2004.
18. Yuliya Lierler. Abstract Answer Set Solvers. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 377–391, Berlin, Heidelberg, 2008. Springer-Verlag.
19. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.
20. Ilkka Niemelä, editor. *WASP WP3 Report: Language Extensions and Software Engineering for ASP*. 2005.
21. T. Syrjänen and I. Niemelä. The Smodels System. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, 2001.
22. Marek Truszczyński, Janhunen Tommi, Martin Brain, Wolfgang Faber, Marco Maratea, Axel Polleres, Torsten Schaub, and Roman Schindlauer. Language forum. In De Vos and Schaub [10], pages 3–39.