spock: A Debugging Support Tool for Logic Programs under the Answer-Set Semantics*

Martin Gebser[1], Jörg Pührer[2], Torsten Schaub[1],
Hans Tompits[2], and Stefan Woltran[2]

[1] Institut für Informatik, Universität Potsdam,
August-Bebel-Straße 89, D-14482 Potsdam, Germany
{gebser,torsten}@cs.uni-potsdam.de
[2] Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9–11, A–1040 Vienna, Austria
{puehrer,tompits}@kr.tuwien.ac.at
woltran@dbai.tuwien.ac.at

Abstract. Answer-set programming (ASP) is an emerging logic-programming paradigm that strictly separates the description of a problem from its solving methods. Despite its semantic elegance, ASP suffers from a lack of support for program developers. In particular, tools are needed that help engineers in detecting erroneous parts of their programs. Unlike in other areas of logic programming, applying tracing techniques for debugging logic programs under the answer-set semantics seems rather unnatural, since employing imperative solving algorithms would undermine the declarative flavour of ASP. In this paper, we present the system spock, a debugging support tool for answer-set programs making use of ASP itself. The implemented techniques maintain the declarative nature of ASP within the debugging process and are independent of the actual computation of answer sets.

1 General Information

Answer-set programming (ASP) [1] has become an important logic-programming paradigm for declarative problem solving, incorporating fundamental concepts of nonmonotonic reasoning. A major reason why ASP has not yet found a more widespread popularity as a problem-solving technique, however, is its lack of suitable *engineering tools* for developing programs. In particular, realising tools for *debugging* answer-set programs is a clearly recognised issue in the ASP community, and several approaches in this direction have been proposed in recent years [2–5].

From a theoretical point of view, the nonmonotonicity of answer-set programs is an aggravating factor for detecting sources of errors, since every rule of a program might significantly influence the resulting answer sets. On the other hand, applying tracing techniques for debugging logic programs under the answer-set semantics seems rather unnatural, since employing imperative solving algorithms would undermine the declarative flavour of ASP.

* This work was partially supported by the Austrian Science Fund (FWF) under project P18019.

In this paper, we discuss the main features of the system spock [6], which supports developers of answer-set programs in locating errors in their programs by exploiting the declarative nature of ASP itself, but being independent of specific ASP solvers. The name "spock" makes reference to the fact that detecting errors is done by means of logic, just like the popular Vulcan of Star Trek fame.

The theoretical background of the implemented methods was introduced in previous work [5], exploiting and extending a *tagging technique* as used by Delgrande et al. [7] for compiling ordered logic programs into standard ones. In our approach, a program to debug, Π, is augmented with dedicated meta-atoms, called *tags*, serving two purposes: Firstly, they allow for controlling and manipulating the applicability of rules, and secondly, tags occurring in the answer sets of the extended program reflect various properties of Π. Our tool implements the tagging process and further related translations for a program Π to debug, allowing for an extrapolation of non-existing answer sets in combination with explanations why an interpretation is not an answer set of Π.

For illustration of the debugging questions addressed, consider the problem of inviting guests to a party when it is known that some of them would appear only if certain others do or do not attend the festivity. An instance of such a setting is encoded in program Π_{inv}, where each atom represents the appearing of a potential party visitor:

$$\begin{array}{lcllcl} r_1 & = & jim \leftarrow uhura, & r_4 & = & chekov \leftarrow not\ bones, \\ r_2 & = & jim \leftarrow not\ chekov, & r_5 & = & bones \leftarrow jim, \\ r_3 & = & uhura \leftarrow chekov, not\ scotty, & r_6 & = & scotty \leftarrow not\ uhura. \end{array}$$

This program has two answer sets, viz., $\{chekov, scotty\}$ and $\{bones, jim, scotty\}$. Assume that Sulu, the programmer, is quite perplexed by this result, wondering why there is a scenario where only Chekov and Scotty, who merely have a neutral relation to each other rather than a friendship, attend. On the other hand, he is astonished as there is no possibility such that Uhura and Jim can jointly be invited. With the help of the tool spock, reasons for such mismatches between the expected and the actual semantics of a program can be found.

2 Background

2.1 Answer-Set Programs

A (*normal*) *logic program* (over an alphabet $\mathcal{A}$) is a finite set of rules of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, \tag{1}$$

where a and $b_i, c_j \in \mathcal{A}$ are atoms, for $0 \leq i \leq m$, $0 \leq j \leq n$. A *literal* is an atom a or its negation $not\ a$. For a rule r as in (1), let $head(r) = a$ be the *head* of r and $body(r) = \{b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n\}$ the *body* of r. Furthermore, we define $body^+(r) = \{b_1, \ldots, b_m\}$ and $body^-(r) = \{c_1, \ldots, c_n\}$. For a logic program Π, a set X of atoms is an *answer set* of Π iff X is a minimal model of $\{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}$. For uniformity, we assume that any integrity constraint $\leftarrow body(r)$ is expressed as a rule $w \leftarrow body(r), not\ w$, where w is a globally new atom. Moreover,

we allow nested expressions of form $not\ not\ a$, where a is some atom, in the body of rules. Such rules are identified with normal rules in which $not\ not\ a$ is replaced by $not\ a^\star$, where $a^\star$ is a globally new atom, together with an additional rule $a^\star \leftarrow not\ a$.

2.2 Tagging-Based Debugging

In what follows, we sketch the theoretical principles underlying our system spock. For a more detailed discussion, we refer to Brain et al. [5]. The main idea of tagging is to split the head from the body, for each rule in a program, and thereby to intervene into the applicability of rules. After this division, tags are installed for triggering rules. This way, the formation of answer sets can be controlled, and tags in the answer sets of the transformed (or tagged) program reflect inherent properties of the original program.

Technically, a program Π (over alphabet $\mathcal{A}$) to debug is rewritten into a program $\mathcal{T}_\mathsf{K}[\Pi]$ over an extended alphabet $\mathcal{A}^+$. Let Π be a logic program over $\mathcal{A}$ and consider a bijection n, assigning to each rule r over $\mathcal{A}$ a unique name n_r. Then, the program $\mathcal{T}_\mathsf{K}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for $r \in \Pi$, $b \in body^+(r)$, and $c \in body^-(r)$:

$$head(r) \leftarrow \mathsf{ap}(n_r), not\ \mathsf{ko}(n_r), \tag{2}$$
$$\mathsf{ap}(n_r) \leftarrow \mathsf{ok}(n_r), body(r), \tag{3}$$
$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ b, \tag{4}$$
$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ not\ c, \tag{5}$$
$$\mathsf{ok}(n_r) \leftarrow not\ \overline{\mathsf{ok}}(n_r). \tag{6}$$

The tags $\mathsf{ap}(n_r)$ and $\mathsf{bl}(n_r)$ express whether a rule r is applicable or blocked, respectively, while the *control tags* $\mathsf{ko}(n_r)$, $\mathsf{ok}(n_r)$, and $\overline{\mathsf{ok}}(n_r)$ are used for manipulating the application of r. Intuitively, the rules of Π are split into rules of forms (2) and (3), separating the applicability of a rule from the actual occurrence of the respective rule head in an interpretation. Analogously, rules of forms (4) and (5) elicit which rules are blocked. Tags stating whether rule r is applicable or blocked are only derived if $\mathsf{ok}(n_r)$ holds, which is by default the case, as expressed by rules of form (6).

We call $\mathcal{T}_\mathsf{K}[\Pi]$ the *kernel tagging* of Π, since it serves as a basic submodule for more enhanced programs facilitating certain debugging requests. One such extension scenario is the extrapolation of non-existing answer sets of a program Π over $\mathcal{A}$. Using further translations, $\mathcal{T}_\mathsf{P}$, $\mathcal{T}_\mathsf{C}$, and $\mathcal{T}_\mathsf{L}$ [5], the occurrence of *abnormality tags*, $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$, respectively, in an answer set X^+ of the transformed program provides information why an interpretation $X = X^+ \cap \mathcal{A}$ is not an answer set of Π. Here, we make use of the Lin-Zhao theorem [8], which qualifies answer sets as models of the *completion* [9] and the *loop formulas* of a program. In particular, the program-oriented abnormality tag $\mathsf{ab}_p(n_r)$ indicates that rule r is applicable but not satisfied with respect to an interpretation. The completion-oriented abnormality tag $\mathsf{ab}_c(a)$ signals that a is in the considered interpretation but all rules having a as head are blocked. Finally, the presence of a loop-oriented abnormality tag $\mathsf{ab}_l(a)$ indicates that the derivation of atom a might recursively depend on a itself and, therefore, violate the minimality criterion for answer sets. Note that all transformations used are polynomial in the size of the input program and can be constructed for all programs under consideration, even for programs without answer sets.

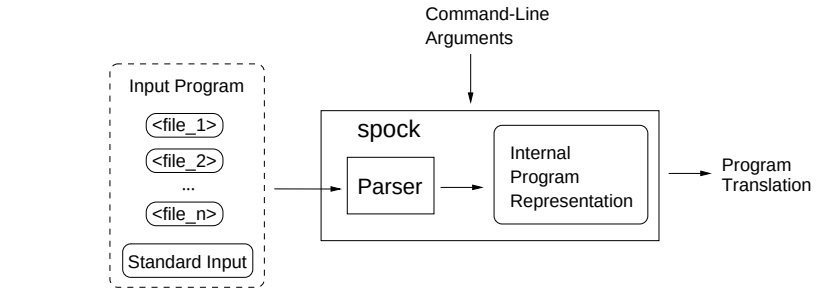

Fig. 1. Data flow of program translations

3 System

`spock` is a command-line oriented tool, written in Java 5.0 and published under the GNU general public license [10]. It is publicly available at

`http://www.kr.tuwien.ac.at/research/debug`

as a jar-package including binaries and sources.

The data flow for all transformations is depicted by Fig. 1. First, the input program is parsed and represented in an internal data structure. Then, the actual program transformation is performed, as specified by command-line arguments.

The tagging technique uses labels to refer to individual rules. Therefore, we allow the programmer to add labels to the rules of the program to debug. As this requires an extension of the program syntax, `spock` offers an interface to `dlv` [11] and `lparse`/`smodels` [12] for computing answer sets of labelled programs.

For illustration of the debugging process, reconsider program Π_{inv}, having the answer sets $X_1 = \{chekov, scotty\}$ and $X_2 = \{bones, jim, scotty\}$, and assume that it is stored in file `FILE`. The kernel tagging $\mathcal{T}_{\mathsf{K}}[\Pi_{inv}]$ is then obtained by the call

```
java -jar spock.jar -k FILE .
```

By piping the result of the command to an answer-set solver, we obtain the answer sets

$$X_1^+ = X_1 \cup \{\mathsf{ap}(n_{r_4}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_5})\} \cup \mathsf{OK} \text{ and}$$
$$X_2^+ = X_2 \cup \{\mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_5}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4})\} \cup \mathsf{OK},$$

where $\mathsf{OK} = \{\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4}), \mathsf{ok}(n_{r_5}), \mathsf{ok}(n_{r_6})\}$, extending X_1 and X_2 by information about the applicability of rules. E.g., the presence of $\mathsf{ap}(n_{r_4})$ in X_1^+ indicates that rule r_4 is applicable with respect to X_1, and hence $chekov \in X_1$ but $bones \notin X_1$, while $\mathsf{bl}(n_{r_3}) \in X_1^+$ indicates that r_3 is blocked with respect to X_1. This is because $scotty \in X_1$.

The flags '`-expo`', '`-exco`', and '`-exlo`' activate the extrapolation translations $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$, respectively. Instead of using all three flags simultaneously, setting

'-ex' produces the union of the resulting programs. Furthermore, in order to restrict the scope of transformation $\mathcal{T}_\mathsf{P}$ to a subprogram Π' (respectively, translations $\mathcal{T}_\mathsf{C}, \mathcal{T}_\mathsf{L}$ to sets A_C, A_L of atoms), the names of the considered rules (respectively, atoms) can be explicitly stated in a comma-separated list following the '-exrules=' (resp., '-exatomsC=' and '-exatomsL=') flag. Finally, spock allows for computing only abnormality-minimum answer sets by means of dlv-specific weak constraints. The flags '-minab', '-minabp', '-minabc', or '-minabl' make spock output weak constraints for minimising all abnormality tags, program-oriented abnormality tags, completion-oriented abnormality tags, or loop-oriented abnormality tags, respectively.

As for our example, recall that Sulu wanted to know why there is no chance for Uhura and Jim to attend the same party. Therefore, we add the constraints $\leftarrow \mathit{not}\ \mathit{uhura}$ and $\leftarrow \mathit{not}\ \mathit{jim}$ to Π_{inv}. Let file FILE2 contain the overall program, which does not have answer sets. The (optimal) answer sets of the tagged program obtained by the call

```
java -jar spock.jar -k -ex -exrules=r1,r2,r3,r4,r5,r6
                    -minab FILE2 ,
```

projected to the atoms occurring in Π_{inv} and the abnormality tags, are given by $\{\mathsf{ab}_c(\mathit{chekov}),\ \mathit{bones},\ \mathit{chekov},\ \mathit{jim},\ \mathit{uhura}\}$, $\{\mathsf{ab}_c(\mathit{uhura}),\ \mathit{bones},\ \mathit{jim},\ \mathit{uhura}\}$, and $\{\mathsf{ab}_p(n_{r_5}), \mathit{chekov}, \mathit{jim}, \mathit{uhura}\}$, indicating that $\{\mathit{bones}, \mathit{chekov}, \mathit{jim}, \mathit{uhura}\}$ is not an answer set of Π_{inv} because atom chekov is not supported. Likewise, uhura is not supported with respect to $\{\mathit{bones}, \mathit{jim}, \mathit{uhura}\}$. Finally, $\{\mathit{chekov}, \mathit{jim}, \mathit{uhura}\}$ is not an answer set as it does not satisfy rule r_5.

4 Discussion and Related Work

In this paper, we presented spock, a prototype implementation of a debugging support tool for answer-set programs. The implemented methodology relies on theoretical results of previous work [5] and is based on the idea that programs to be debugged are translated into other programs having answer sets that offer debugging-relevant information about the original programs. After an initial kernel transformation, we get insight into the applicability of rules with respect to individual answer sets. In a further step, the system allows for identifying causes why interpretations are not answer sets. Here, spock distinguishes between abnormalities due to missing or spare atoms, or atoms whose presence in an interpretation is self-caused. In order to restrict the amount of information returned to the programmer, standard ASP optimisation techniques can be used to focus on interpretations with a minimal number of abnormalities. In addition to the tagging technique described here, spock also supports another approach towards debugging answer-set programs based on meta-programming [13, 14]. Future work includes the integration of further aspects of the translation approach and the design of a graphical user interface to ease the use of the features spock provides.

Implementations of related techniques include smdebug [3], a prototype debugger focusing on odd-cycle-free inconsistent programs. The system is designed to find minimal sets of constraints, restoring consistency when removed from a program. Brain and De Vos [2] present the system *IDEAS*, implementing two query algorithms addressing the questions why a set of literals is true with respect to some or false with respect to

all answer sets of a program. Both algorithms are procedural and similar to the ones used in ASP solvers, suggesting that an approach using program-level transformations would be more practical. Pontelli and Son [4] developed a preliminary implementation for their adoption of so-called *justifications* [15, 16] to the problem of debugging answer-set programs. Their system returns visual output in form of graphs explaining why atoms are (not) present in an answer set.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In: Proc. ASP'05. Volume 142, CEUR Workshop Proceedings (CEUR-WS.org) (2005) 141–152
3. Syrjänen, T.: Debugging Inconsistent Answer Set Programs. In: Proc. NMR'06. (2006) 77–83
4. Pontelli, E., Son, T.: Justifications for Logic Programs under Answer Set Semantics. In: Proc. ICLP'06. Springer (2006) 196–210
5. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP Programs by means of ASP. In: Proc. LPNMR'07. Springer (2007) 31–43
6. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: "That is Illogical Captain!" – The Debugging Support Tool spock for Answer-Set Programs: System Description. In: Proc. SEA'07. (2007) 71–85
7. Delgrande, J., Schaub, T., Tompits, H.: A Framework for Compiling Preferences in Logic Programs. Theory and Practice of Logic Programming **3** (2003) 129–187
8. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. Artificial Intelligence **157** (2004) 115–137
9. Clark, K.: Negation as Failure. In: Logic and Data Bases. Plenum Press (1978) 293–322
10. GNU General Public License – Version 2, June 1991. Free Software Foundation Inc. (1991) http://www.gnu.org/copyleft/gpl.html
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7** (2006) 499–562
12. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
13. Pührer, J.: On Debugging of Propositional Answer-Set Programs. Master's thesis, Vienna University of Technology, Austria (2007)
14. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A Meta-Programming Technique for Debugging Answer-Set Programs. In: Proc. AAAI'08. (2008) To appear
15. Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I.: Justifying Proofs using Memo Tables. In: Proc. PPDP'00. (2000) 178–189
16. Specht, G.: Generating Explanation Trees even for Negations in Deductive Database Systems. In: Proc. LPE'93. (1993) 8–13