"That is Illogical Captain!" – The Debugging Support Tool spock for Answer-Set Programs: System Description*

Martin Brain[1], Martin Gebser[2], Jörg Pührer[3], Torsten Schaub[2], Hans Tompits[3], and Stefan Woltran[3]

[1] Department of Computer Science, University of Bath, Bath, BA2 7AY, United Kingdom
mjb@cs.bath.ac.uk
[2] Institut für Informatik, Universität Potsdam, August-Bebel-Straße 89, D-14482 Potsdam, Germany
{gebser,torsten}@cs.uni-potsdam.de
[3] Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9–11, A–1040 Vienna, Austria
{puehrer,tompits,stefan}@kr.tuwien.ac.at

Abstract. Answer-set programming (ASP) is a logic programming paradigm for declarative problem solving which gained increasing importance during the last decade. However, so far hardly any tools exist supporting software engineers in developing answer-set programs, and there are no standard methodologies for handling unexpected outcomes of a program. Thus, writing answer-set programs is sometimes quite intricate, especially when large programs for real-world applications are required. In order to increase the usability of ASP, the development of appropriate debugging strategies is therefore vital. In this paper, we describe 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 from the actual computation of answer sets.

1 Introduction

Answer-set programming (ASP) [1] is an important logic-programming paradigm for declarative problem solving, based on principles of nonmonotonic reasoning. Any answer-set program consists of logical rules specifying a problem, for which each of the program's answer sets is a solution. Since every rule of a program might significantly influence the resulting answer sets, it is hard to find the sources of errors in large programs in case of a mismatch between the program's output and the user's expectations. For example, consider the problem of inviting guests to a party at the renowned starship Enterprise. Sulu wants to give a party for his colleagues, however facing the complication that some of them would appear only if certain others do or do not attend the festivity. Knowing the social preferences of potential party guests, Sulu tries to get

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

an overview of possible invitation scenarios by means of answer-set programming and ends up with the following rules for a program Π_{inv}, where each atom represents the actual appearing of a potential party visitor:

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

This program has two answer sets, viz. $\{\mathit{chekov}, \mathit{scotty}\}$ and $\{\mathit{bones}, \mathit{jim}, \mathit{scotty}\}$. Sulu is quite perplexed by this result, wondering why there is a scenario where only Chekov and Scotty attend who merely have a neutral relation to each other rather than a friendship. On the other hand, Sulu is astonished as there is no satisfactory possibility such that Uhura and Jim can jointly be invited. The only way out appears to consult his half-Vulcan half-Human friend, Spock, for advice.

In this paper, we describe a system helping developers of answer-set programs to detect and locate errors in their programs. We call our system `spock`, making reference to its ability of supporting users detecting errors based on principles of logic, since the implemented techniques make use of ASP itself for debugging answer-set programs. In contrast to other debugging strategies in logic programming, our methodology works independently of specific ASP solvers and preserves the declarative nature of ASP.

The theoretical background for our approach was introduced in previous work [2], and relies on a *tagging technique* as used by Delgrande et al. [3] for compiling ordered logic programs into standard ones. In our approach, a program to debug, Π, is translated into another program, $\mathcal{T}_{\mathsf{K}}[\Pi]$, equipped with several meta atoms, called *tags*, which allow for controlling the formation of answer sets and reflect different properties (like the applicability status of a rule, for instance). This way, we have the possibility of investigating the actual answer sets of Π. $\mathcal{T}_{\mathsf{K}}[\Pi]$ can be regarded as a *kernel transformation* that may be extended for further debugging techniques. One such extension, featured by `spock`, is the extrapolation of non-existing answer sets in combination with explanations why an interpretation is not an answer set of Π.

The paper is organised as follows. Section 2 gives the relevant prerequisites about ASP, while Section 3 reviews the theoretical background of our tool. The main features of our tool, then, are described in Section 4. The paper is concluded with Section 5 containing some general remarks and a discussion about related work. An appendix lists specific commands of `spock`.

2 Background

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

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

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 $\mathit{not}\ a$. For a rule r as in (1), let $\mathit{head}(r) = a$ be the *head* of r and $\mathit{body}(r) = \{b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n\}$ the *body* of r. Furthermore, we define $\mathit{body}^+(r) = \{b_1, \ldots, b_m\}$ and $\mathit{body}^-(r) = \{c_1, \ldots, c_n\}$. The set of atoms occurring

in a program Π is denoted by $At(\Pi)$. For collecting rules sharing the same head a, we use $def(a,\Pi) = \{r \in \Pi \mid head(r) = a\}$. 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 $a^\star \leftarrow not\ a$. We also take advantage of (singular) *choice rules* of form $\{a\} \leftarrow body(r)$ [4], which are an abbreviation for $a \leftarrow body(r), not\ not\ a$. A program Π is *positive* if $body^-(r) = \emptyset$, for all $r \in \Pi$. By $Cn(\Pi)$, we denote the smallest model of a positive program Π.

The definition of an answer set is as follows. The *reduct*, Π^X, of a program Π relative to a set X of atoms is the positive program $\{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}$. Then, X is an *answer set* of Π iff $Cn(\Pi^X) = X$. The set of all answer sets of a program Π is denoted by $AS(\Pi)$.

An alternative characterisation of answer sets is provided by the Lin-Zhao Theorem [5], qualifying answer sets as models of the *completion* of a program in the sense of Clark [6] and the *loop formulas* of the program. We make use of this perspective on the answer-set semantics to identify sources of errors when extrapolating non-existing answer sets as described in the following section.

3 Tag-Based Debugging Methodology

Our approach relies on the *tagging technique* as used by Delgrande et al. [3]. In what follows, we sketch the theoretical principles underlying our system `spock`. For a more detailed discussion, we refer to Brain et al. [2].

The basic idea of tagging is to decompose the rules of a program Π over $\mathcal{A}$ into several other rules, in order to gain control over their applicability and for analysing their mutual interferences. To be able to refer to individual rules, we use a bijection, n, assigning each rule r over $\mathcal{A}$ a unique name n_r. We call a pair $n_r : r$, comprising a rule r and its name n_r, a *labeled rule*, and a set of labeled rules a *labeled program*. The semantics of a labeled program Π is given by the semantics of the ordinary program $\{r \mid n_r : r \in \Pi\}$. In view of this straightforward correspondence between programs (resp., rules) and labeled programs (resp., labeled rules), we will usually not distinguish between them in the sequel.

For decomposing the rules of a program, so-called *tags* are introduced, which are new, pairwise distinct propositional atoms, given by $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, $\mathsf{ok}(n_r)$, $\overline{\mathsf{ok}}(n_r)$, $\mathsf{ko}(n_r)$, $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$, for each $r \in \Pi$ and $a \in At(\Pi)$. Intuitively, $\mathsf{ap}(n_r)$ and $\mathsf{bl}(n_r)$ indicate whether some rule r is currently applicable or blocked, respectively, while $\mathsf{ok}(n_r)$, $\overline{\mathsf{ok}}(n_r)$, and $\mathsf{ko}(n_r)$ are used to include or exclude particular rules from a debugging request. Furthermore, the *abnormality* tags $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$ inform the user what went wrong in case no answer set for the program under consideration exists. We explain their particular functioning in detail below.

In a first transformation step, the *kernel transformation*, $\mathcal{T}_\mathsf{K}$, rewrites a given program, Π, such that, for every $r \in \Pi$, $\mathsf{ap}(n_r)$ (resp., $\mathsf{bl}(n_r)$) is contained in an answer set of $\mathcal{T}_\mathsf{K}[\Pi]$ whenever r can be applied (resp., is blocked). Apart from tags, the answer

sets of Π and $\mathcal{T}_{\mathsf{K}}[\Pi]$ are preserved. Formally, $\mathcal{T}_{\mathsf{K}}$ maps a logic program Π over $\mathcal{A}$ into another program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over an extended alphabet $\mathcal{A}^+$ in the following way: for every $r \in \Pi$, $b \in \mathit{body}^+(r)$, and $c \in \mathit{body}^-(r)$, $\mathcal{T}_{\mathsf{K}}[\Pi]$ contains

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

Intuitively, every $r \in \Pi$ is split into Rules (2) and (3), separating the head and the body of r, thereby decoupling the applicability of r, indicated by the tag $\mathsf{ap}(n_r)$, from the conclusion $\mathit{head}(r)$ of r. Rules (4) and (5) derive tags $\mathsf{bl}(n_r)$ whenever r is blocked. The tag $\mathsf{ok}(n_r)$, along with $\overline{\mathsf{ok}}(n_r)$, provides a handle for switching r "on or off".

The program $\mathcal{T}_{\mathsf{K}}[\Pi]$ plays the role of a basic module for various debugging requests. Extension modules may add new rules, using tags $\mathsf{ok}(n_r)$, $\overline{\mathsf{ok}}(n_r)$, and $\mathsf{ko}(n_r)$ for manipulating the applicability of a rule r, in order to analyse the behaviour of Π.

Example 1. Reconsider the program Π_{inv} from the introduction, having the answer sets $\{\mathit{chekov}, \mathit{scotty}\}$ and $\{\mathit{bones}, \mathit{jim}, \mathit{scotty}\}$. The answer sets of $\mathcal{T}_{\mathsf{K}}[\Pi_{inv}]$ are

$$X_1 = \{\mathit{chekov}, \mathit{scotty}, \mathsf{ap}(n_{r_4}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_5})\} \cup \mathit{OK},$$

and

$$X_2 = \{\mathit{bones}, \mathit{jim}, \mathit{scotty}, \mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_5}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4})\} \cup \mathit{OK},$$

where $\mathit{OK} = \{\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4}), \mathsf{ok}(n_{r_5}), \mathsf{ok}(n_{r_6})\}$. 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 $\mathit{chekov} \in X_1$ but $\mathit{bones} \notin X_1$, while $\mathsf{bl}(n_{r_3}) \in X_1$ indicates that r_3 is blocked with respect to X_1. This is because $\mathit{scotty} \in X_1$. ◊

As stated above, the tagged kernel program $\mathcal{T}_{\mathsf{K}}[\Pi]$ can be used as a basic submodule for more enhanced programs, facilitating debugging requests. One such extension scenario is the extrapolation of non-existing answer sets of a program Π over $\mathcal{A}$. Using further translations of the original program, we may investigate why an interpretation is not an answer set of Π. An answer set, X^+, of the transformed program offers information about the interpretation $X = X^+ \cap \mathcal{A}$ of Π in form of the three abnormality tags, $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$. Their presence signals why X is not an answer set, by detecting problems originating from the program, its completion, and its non-trivial loop formulas, respectively. For the detection of these three problem sources, we have the corresponding program translations $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$, which are used together with the kernel tagging of the respective program, yielding an overall transformation $\mathcal{T}_{\mathsf{Ex}}[\Pi, X] = \mathcal{T}_{\mathsf{K}}[\Pi] \cup \mathcal{T}_{\mathsf{P}}[\Pi] \cup \mathcal{T}_{\mathsf{C}}[\Pi, X] \cup \mathcal{T}_{\mathsf{L}}[X]$, where $X \subseteq \mathit{At}(\Pi)$.

The program-oriented abnormality tag $\mathsf{ab}_p(n_r)$ indicates that rule r is applicable but not satisfied with respect to X, i.e., $\mathit{body}^+(r) \subseteq X$, $\mathit{body}^-(r) \cap X = \emptyset$, but

$head(r) \notin X$. The respective translation $\mathcal{T}_{\mathsf{P}}[\Pi]$ over $\mathcal{A}^+$ is given by the set of all rules

$$\mathsf{ko}(n_r) \leftarrow , \tag{7}$$

$$\{head(r)\} \leftarrow \mathsf{ap}(n_r), \tag{8}$$

$$\mathsf{ab}_p(n_r) \leftarrow \mathsf{ap}(n_r), not\ head(r), \tag{9}$$

for $r \in \Pi$. By adding the facts of form (7), the rules of form (2) are blocked. Their purpose, deriving the consequences of the original rules, is now fulfilled by the rules of form (8). However, the head atom of an original rule r is not necessarily derived, even when r is applicable. Whenever an applicable rule is not applied, a rule of form (9) provides the program-oriented abnormality tag $\mathsf{ab}_p(n_r)$.

Example 2. Consider program $\Pi_p = \{\, n_r : chekov \leftarrow not\ bones \,\}$. The empty set is not an answer set of Π_p, since r is applicable with respect to $\emptyset$ but $chekov \notin \emptyset$. This is reflected by $\mathcal{T}_{\mathsf{Ex}}[\Pi_p, At(\Pi_p)]$ in that it possesses an answer set X^+ containing abnormality tag $\mathsf{ab}_p(n_r)$ and $X^+ \cap At(\Pi_p) = \emptyset$. ♢

The completion-oriented abnormality tag $\mathsf{ab}_c(a)$ is included in X^+ whenever a is in the considered interpretation but all rules having a as head are blocked. The logic program $\mathcal{T}_{\mathsf{C}}[\Pi, X]$ over $\mathcal{A}^+$, for $X \subseteq At(\Pi)$, is given by the set of all rules

$$\{a\} \leftarrow \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k}), \tag{10}$$

$$\mathsf{ab}_c(a) \leftarrow \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k}), a, \tag{11}$$

for $a \in X$, where $\{r_1, \ldots, r_k\} = def(a, \Pi)$.

The rules of form (10) allow an atom $a \in At(\Pi)$ to be derived even if all rules $r \in def(a, \Pi)$ are blocked. Whenever this happens, a rule of form (11) provides the completion-oriented abnormality tag $\mathsf{ab}_c(a)$.

Example 3. Consider program $\Pi_c = \{\, n_r : uhura \leftarrow chekov \,\}$. The interpretation $X = \{uhura\}$ is not an answer set of Π_c, since the only rule deriving $uhura$ is not applicable. Accordingly, there is an answer set X^+ of $\mathcal{T}_{\mathsf{Ex}}[\Pi_c, At(\Pi_c)]$ containing abnormality tag $\mathsf{ab}_c(uhura)$ and $X^+ \cap At(\Pi_c) = X$. ♢

Finally, the presence of a loop-oriented abnormality tag $\mathsf{ab}_l(a)$ in X^+ indicates that the occurrence of atom a might recursively depend on a itself and, therefore, violate the minimality criterion for answer sets. The corresponding translation $\mathcal{T}_{\mathsf{L}}[X]$ over $\mathcal{A}^+$, for $X \subseteq At(\Pi)$, is given by the following set of rules, for each $a \in X$:

$$\{\mathsf{ab}_l(a)\} \leftarrow not\ \mathsf{ab}_c(a), \tag{12}$$

$$a \leftarrow \mathsf{ab}_l(a). \tag{13}$$

The rules of form (12) allow to add a loop-oriented abnormality tag $\mathsf{ab}_l(a)$ for $a \in X^+$, providing a is supported. The rules of form (13) ensure that a is actually contained in X^+.

Example 4. Consider program Π_l, consisting of

$$n_{r_1} : jim \leftarrow bones \quad \text{and} \quad n_{r_2} : bones \leftarrow jim.$$

The interpretation $X = \{bones, jim\}$ is a classical model of Π_l but does not satisfy the loop formulas of Π_l. So, every answer set X^+ of $\mathcal{T}_{\mathsf{Ex}}[\Pi_l, At(\Pi_l)]$ such that $X^+ \cap At(\Pi_l) = X$ includes one of the abnormality tags $\mathsf{ab}_l(bones)$ or $\mathsf{ab}_l(jim)$. ♢

Table 1. Labeled program syntax of `spock`.

program	:= $(\ddots)^*$rule$((\ddots)^*$*rule*$)^*(\ddots)^*$
rule	:= (rulelabel ... ':' ...)? (head ... '.' |
	head ... ':-' ... body ... '.' | ':-' ... body ... '.')
head	:= atom
body	:= literal(',' ... literal)*
literal	:= atom | 'not' ... atom
atom	:= symb ('(' ... term(',' ... term)* ... ')')?
term	:= variable | symb
rulelabel	:= ('a' – 'z' | 'A' – 'Z' | '0' – '9')*
variable	:= ('A' – 'Z')('a' – 'z' | 'A' – 'Z' | '0' – '9' | '_')*
symb	:= ('a' – 'z' | '0' – '9')('a' – 'z' | 'A' – 'Z' | '0' – '9' | '_')*
$\ddots$	:= (...)* '\n' (...)*
...	:= ('␣' | '\t')*

4 System

Our debugging system `spock` implements the program translations described in the previous section. It is a command-line oriented tool, parsing and translating its input, which is taken from standard input and text files. The program is written in Java 5.0 and published under the GNU General Public License [7]. It can be used either with `DLV` [8] or with `Smodels` [4] (together with `lparse`) and is publicly available at

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

as a jar-package including binaries and sources.

4.1 Usage

Generally, `spock` is executed by a shell command of the form

`java -jar spock.jar { OPTION | FILENAME }`* ,

assuming `java` is the execution command for the Java virtual machine. If no filename is given, `spock` expects input from the operation system's standard input. A list of important options is given in Appendix A.

4.2 System Input

The input primarily consists of the logic programs which are to be debugged. Additionally, `spock` also accepts debugging statements, and various solver-specific input. The accepted program syntax is closely related to the core languages of `DLV` and `Smodels`. Here, we restrict ourselves to labeled normal logic programs albeit `spock` accepts also programs with a richer syntax like disjunctive logic programs. The basic input language of `spock` is depicted in Table 1 using regular expressions.

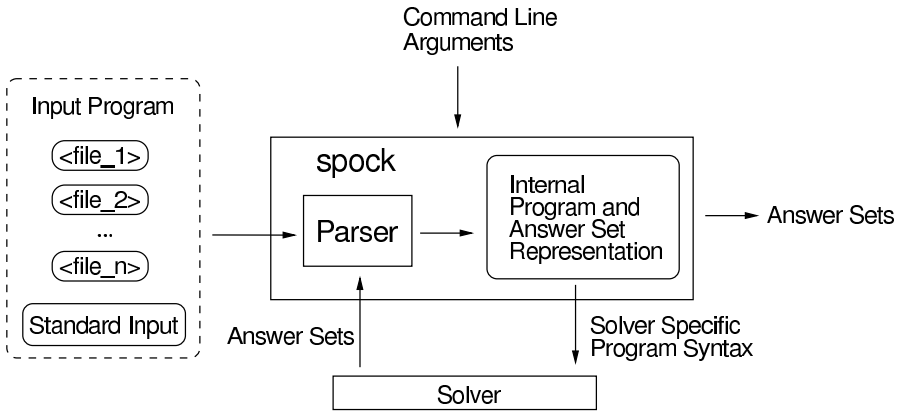

Fig. 1. Data flow of answer-set computation for labeled normal programs.

Rule labeling is introduced as a device to explicitly refer to certain rules. As stated in Table 1, a rule may have its label omitted. For a previously unlabeled rule, `spock` automatically assigns the label `r`n according to the line number n in which it appears in the program. Note that duplicate rule labels will produce a warning message. If the input is spread over multiple input files, their contents will be internally joined as if it were only one file. Additional content read from standard input when using the '`--`' flag is also appended to any input from files.

Since labeled rules cannot be read by conventional ASP solvers, `spock` offers an interface to `DLV` and `Smodels` providing answer-set computation for labeled programs, described next.

4.3 Answer-Set Computation for Labeled Normal Programs

In order to perform answer-set computation for labeled programs, `DLV` or `Smodels` (the latter in combination with its grounder `lparse`) must be found in the command search path of the used system.

Internally, `spock` transforms the parsed input program Π into a solver-compatible representation before forwarding it to the externally called answer-set solver. The resulting set of answer sets, $AS(\Pi)$, is then parsed and stored for further processing. When using flag '`-o`', `spock` outputs $AS(\Pi)$. Command line arguments for externally called systems can be forwarded using the flags '`-dlvarg`', '`-lparg`', and '`-smarg`' (see also Appendix A). Fig. 1 illustrates the typical data flow of answer-set computation with `spock`.

Example 5. Consider input file `file5`, containing our example program Π_{inv}:

```
r1 : jim :- uhura.
r2 : jim :- not chekov.
r3 : uhura :- chekov, not scotty.
```

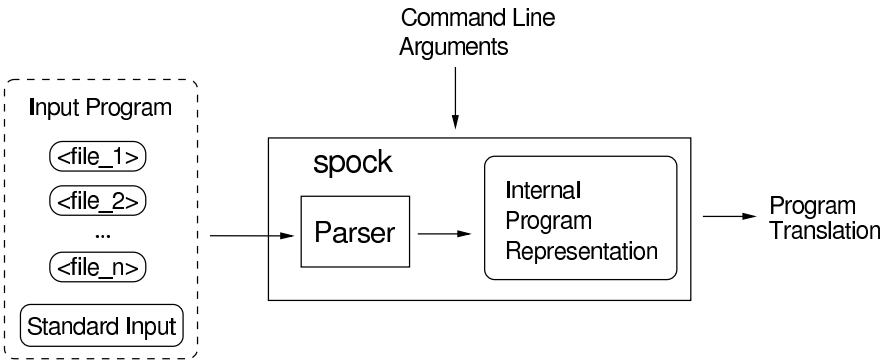

Fig. 2. Data flow of program translations.

```
r4 : chekov :- not bones.
r5 : bones :- jim.
r6 : scotty :- not uhura.
```

The answer sets for this program can be computed using `DLV` by the command:

```
java -jar spock.jar -x -o file5.
```

Flag '`-x`' calls `DLV` externally on the input program and '`-o`' triggers the output of its answer sets. Note that the call yields the output of the corresponding answer sets in lexicographic order:

```
{bones, jim, scotty}
{chekov, scotty}.
```

The same result can be achieved using `Smodels` and `lparse` in a similar manner:

```
java -jar spock.jar -xsm -o file5.
```
◊

4.4 Kernel Translation

The kernel translation $\mathcal{T}_{\mathsf{K}}[\Pi]$ over $\mathcal{A}^{+}$ of a logic program Π over $\mathcal{A}$ can be obtained by the call

```
java -jar spock.jar -k FILE1 FILE2 ...,
```

where the files `FILE1`, `FILE2`, ..., contain a representation of Π. As visualised in Fig. 2, `spock` first creates an internal representation for the input program before computing and returning its translation.

Example 6. For file `file5` from Example 5, when executing the command

```
java -jar spock.jar -k file5,
```

`spock` returns the translated program $\mathcal{T}_{\mathsf{K}}[\Pi_{inv}]$:

```
jim :- ap(r1), not ko(r1).
ap(r1) :- ok(r1), uhura.
bl(r1) :- ok(r1), not uhura.
ok(r1) :- not -ok(r1).
jim :- ap(r2), not ko(r2).
ap(r2) :- ok(r2), not chekov.
bl(r2) :- ok(r2), not not chekov.
ok(r2) :- not -ok(r2).
uhura :- ap(r3), not ko(r3).
ap(r3) :- ok(r3), chekov, not scotty.
bl(r3) :- ok(r3), not chekov.
bl(r3) :- ok(r3), not not scotty.
ok(r3) :- not -ok(r3).
chekov :- ap(r4), not ko(r4).
ap(r4) :- ok(r4), not bones.
bl(r4) :- ok(r4), not not bones.
ok(r4) :- not -ok(r4).
bones :- ap(r5), not ko(r5).
ap(r5) :- ok(r5), jim.
bl(r5) :- ok(r5), not jim.
ok(r5) :- not -ok(r5).
scotty :- ap(r6), not ko(r6).
ap(r6) :- ok(r6), not uhura.
bl(r6) :- ok(r6), not not uhura.
ok(r6) :- not -ok(r6).
:- falsum.
```

When solving this program, we obtain the answer sets X_1 and X_2 (cf. Example 1). $\Diamond$

4.5 Translations for Extrapolating Answer Sets

Translations for the extrapolation of non-existing answer sets of a program Π can be invoked analogously to the kernel transformation. However, here, the consideration may be restricted to the generation of extrapolation tagging on a subset of Π. This way, the developer can focus the search for errors on a subprogram. The data flow is still the one depicted in Fig. 2.

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 these program translations. In order to restrict the generation of an extrapolation tagging to a subprogram of Π, the names of the considered rules must be explicitly stated in a comma-separated list following the '`-exrules`' flag. Since programs translated via $\mathcal{T}_\mathsf{P}$, $\mathcal{T}_\mathsf{C}$, and $\mathcal{T}_\mathsf{L}$ involve `Smodels`-specific choice rules, we need to set the '`-sm`' flag to activate `Smodels` syntax. Otherwise, `spock` will produce disjunctive rules, simulating the respective choice rules.

Example 7. Consider input file `file7`:

```
r1: jim :- not chekov.
r2: bones :-  not jim.
r3: chekov :- not bones.
```

Since Bones would definitely attend if Jim did, the programmer seemed to err when specifying `r2`. By calling

```
java -jar spock.jar -ex -exrules=r1,r2 -sm file7,
```

we get the extrapolation tagging of the subprogram consisting of the rules labeled `r1` and `r2`, where we expect an error:

```
ko(r1).
{jim} :- ap(r1).
ab_p(r1) :- ap(r1), not jim.
ko(r2).
{bones} :- ap(r2).
ab_p(r2) :- ap(r2), not bones.
{bones} :- bl(r2).
ab_c(bones) :- bl(r2), bones.
{chekov}.
ab_c(chekov) :- chekov.
{jim} :- bl(r1).
ab_c(jim) :- bl(r1), jim.
{ab_l(bones)} :- not ab_c(bones).
bones :- ab_l(bones).
{ab_l(chekov)} :- not ab_c(chekov).
chekov :- ab_l(chekov).
{ab_l(jim)} :- not ab_c(jim).
jim :- ab_l(jim).
```

Since the extrapolation taggings make only sense in conjunction with the kernel tagging, we usually also use the '`-k`' flag to output both translations at once. In order to compute the answer sets of the obtained program, it can be piped from the output of `spock` into another instantiation of it:

```
java -jar spock.jar -k -ex -exrules=r1,r2 -sm file7 |
            java -jar spock.jar -xsm -o.
```

The output of this operation yields nine answer sets; among them are the following:

$$A_1 = \{\mathsf{ab}_c(\mathit{bones}), \mathsf{ab}_c(\mathit{chekov}), \mathsf{ab}_c(\mathit{jim}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathit{bones}, \mathit{chekov}, \mathit{jim}\} \cup S,$$

$$A_2 = \{\mathsf{ab}_c(\mathit{bones}), \mathsf{ab}_l(\mathit{jim}), \mathsf{ap}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathit{bones}, \mathit{jim}\} \cup S,$$

$$A_3 = \{\mathsf{ab}_c(\mathit{bones}), \mathsf{ap}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathit{bones}, \mathit{jim}\} \cup S,$$

where

$$S = \{\mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3})\}.$$

The conclusions drawn from these answer sets depend on the considered interpretation. For example, the abnormality tags in A_1 provide an explanation why $\{bones, chekov, jim\}$ is not an answer set, because all rules having $bones$, $chekov$, or jim in their heads are blocked.

Interpretations A_2 and A_3 provide information why $I = \{bones, jim\}$ is not an answer set. Note that A_2 is a superset of A_3 and contains the additional abnormality tag $\mathsf{ab}_l(jim)$. This is a consequence of the definition of translation $\mathcal{T}_\mathsf{L}$ (and the choice rule used therein). The existence of A_3 makes the information in A_2 obsolete, since the occurrence of atom jim in I is not (positively) depending on itself.

In this debugging situation, A_3 delivers the most relevant information for the programmer since, firstly, he or she expects Bones and Jim to be compatible party guests, and, secondly, A_3 contains only one abnormality tag, $\mathsf{ab}_c(bones)$, focusing the source of error to the question why Bones is not coming. From that, the programmer can identify the erroneous rule `r2` of `file7`. $\Diamond$

In order to reduce the amount of debugging information in a translated program, one can make use of standard ASP optimisation techniques, such as *minimise statements* in `Smodels` or *weak constraints* in `DLV`. The idea is to take only answer sets with a minimum number of abnormality tags into consideration.

By using the flags '`-minab`', '`-minabp`', '`-minabc`', or '`-minabl`', `spock` produces weak constraints for minimising all abnormality tags, all program-oriented abnormality tags, all completion-oriented abnormality tags, or all loop-oriented abnormality tags, respectively.

Example 8. Let us reconsider the program Π_{inv} from the introduction and recall that Sulu wanted to know why there is no chance for Uhura and Jim to attend the same party. For this purpose, we add the two constraints

$$\leftarrow \mathit{not}\ \mathit{uhura} \quad \text{and} \quad \leftarrow \mathit{not}\ \mathit{jim}$$

to Π_{inv} in order to investigate only scenarios including Uhura and Jim as guests. Note that this restriction could also be achieved by using the *assigned* statement of the debugging language presented in our companion work [2], which is partly implemented in `spock` but not further discussed here. The modified program is stored in file `file8`:

```
r1 : jim :- uhura.
r2 : jim :- not chekov.
r3 : uhura :- chekov, not scotty.
r4 : chekov :- not bones.
r5 : bones :- jim.
r6 : scotty :- not uhura.

c1 : :- not uhura.
c2 : :- not jim.
```

The following call returns extrapolation answer sets with a minimum number of abnormality tags:

```
java -jar spock.jar -k -ex
  -exrules=r1,r2,r3,r4,r5,r6 -minab file8 |
  java -jar spock.jar -x -as.
```

Note that we do not use the '-sm' flag since weak constraints for minimisation require the use of DLV as external solver. In the present case, choice rules are simulated by head disjunctions, introducing new auxiliary atoms. They are filtered out automatically, in the second invocation of spock, giving the following answer sets as output:

```
{ab_c(chekov), ap(r1), ap(r3), ap(r5), bl(c1),
 bl(c2), bl(r2), bl(r4), bl(r6), bones, chekov, jim,
 ko(r1), ko(r2), ko(r3), ko(r4), ko(r5), ko(r6),
 ok(c1), ok(c2), ok(r1), ok(r2), ok(r3), ok(r4),
 ok(r5), ok(r6), uhura}

{ab_c(uhura), ap(r1), ap(r2), ap(r5), bl(c1), bl(c2),
 bl(r3), bl(r4), bl(r6), bones, jim, ko(r1), ko(r2),
 ko(r3), ko(r4), ko(r5), ko(r6), ok(c1), ok(c2),
 ok(r1), ok(r2), ok(r3), ok(r4), ok(r5), ok(r6),
 uhura}

{ab_p(r5), ap(r1), ap(r3), ap(r4), ap(r5), bl(c1),
 bl(c2), bl(r2), bl(r6), chekov, jim, ko(r1), ko(r2),
 ko(r3), ko(r4), ko(r5), ko(r6), ok(c1), ok(c2),
 ok(r1), ok(r2), ok(r3), ok(r4), ok(r5), ok(r6),
 uhura}
```

The atom ab_c(chekov) in the first answer set, corresponding to interpretation $\{bones, chekov, jim, uhura\}$, identifies *chekov* as not being supported by any applicable rule. The only rule with head *chekov*, r_4, would require *bones* not to be in the interpretation in order to be applicable. Analogously, ab_c(uhura) signals that *uhura* lacks support when considering interpretation $\{bones, jim, uhura\}$.

The tag ab_p(r5) in the third answer set indicates the applicability of the rule labeled r5 with respect to interpretation $\{chekov, jim, uhura\}$ and hence Bones' incompatible party participation. Clearly, there is no solution for this problem instance that is satisfactory for everybody, given that Jim and Uhura should jointly come and that the respective social preferences are all respected. However, the last answer set indicates an obvious solution for Sulu's diplomatic conflict, viz. not inviting Bones. $\Diamond$

All three answer sets in Example 8 give us a potential handle for resolving our problem, each of them involving a minimum number of abnormalities. However, they are not of the same quality in terms of a real-life solution. So, resolving problems in the context of ASP still depends in large part on knowledge about the domain.

5 Discussion and Related Work

In this paper, we gave an overview about spock, a prototype implementation of a debugging support tool for answer-set programs. The implemented methodology is based

on theoretical results presented in a companion paper [2] and relies on a tagging technique similar to one used for compiling ordered logic programs into standard ones [3].

With `spock`, programs to debug 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, `spock` outputs translations for extrapolating putative, yet non-existing answer sets. In this application scenario, the system allows to identify explanations why interpretations are not answer sets. Here, `spock` distinguishes between abnormalities due to missing or spare atoms, or atoms whose presence in the 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. Future work includes the integration of further aspects of the translation approach as well as the design of a graphical user interface to ease the applicability of the different features `spock` provides.

Implementations of related techniques include `smdebug` [9], a prototype debugger focusing on odd-cycle-free inconsistent programs. For programs without odd cycles, inconsistency can always be linked to conflicting integrity constraints. The system is designed to find minimal sets of constraints, restoring consistency when removed from the program. In most real-world applications, odd cycles are bugs, so, on the one hand, `smdebug` technically catches many of the common programming errors. On the other hand, actual error recovery is often related to normal rules, since constraints, used for restricting the solution space, are more likely to be semantically correct.

Brain and De Vos [10] present the system *IDEAS* (Interactive Development and Evaluation Tool for Answer-Set Semantics), implementing two query algorithms, answering the questions why a set S is in some answer set A and why a set S is not in any answer set. Both algorithms are procedural and similar to the ones used in ASP solvers, suggesting that an approach using a program-level transformation would be more practical.

Pontelli and Son [11] developed a preliminary implementation for their adoption of so-called *justifications* [12–14] to the problem of debugging answer-set programs. The system is embedded in ASP – PROLOG [15] and returns visual output in form of justifications, which are graphs explaining why an atom is in an answer set.

Appendix A Selected Argument Options of `spock`

`--`	If a filename is given, `spock` does not read from standard input, unless this flag is set.
`-p`	Outputs the given program with rule labels.
`-c`	Outputs the given program without rule labels.
`-x`	Runs `DLV` on the given program.
`-xsm`	Runs `Smodels` on the given program.
`-n=NR`	Computes maximally *NR* many answer sets.

`-sm`	Formats various output in `Smodels` syntax, otherwise `DLV` syntax is used.
`-o`	Outputs all computed or read answer sets.
`-as`	Displays all computed or read answer sets in a GUI frame.
`-k`	Outputs the kernel tagging $\mathcal{T}_{\mathsf{K}}[\Pi]$ of a given program Π.
`-ex`	Outputs the extrapolation tagging $\mathcal{T}_{\mathsf{Ex}}[\Pi, At(\Pi)]$ of a given program Π (like `-expo -exco -exlo`; see next).
`-expo`	Outputs the program-oriented extrapolation tagging $\mathcal{T}_{\mathsf{P}}[\Pi]$ of a given program Π.
`-exco`	Outputs the completion-oriented extrapolation tagging $\mathcal{T}_{\mathsf{C}}[\Pi, At(\Pi)]$ of a given program Π.
`-exlo`	Outputs the loop-oriented extrapolation tagging $\mathcal{T}_{\mathsf{L}}[At(\Pi)]$ of a given program Π.
`-exrules=r,s,...`	Restricts extrapolation tagging generation to rules labeled `r`, `s`, ...
`-minab`	Outputs weak constraints to minimise abnormality tags (like the ones described next).
`-minabp`	Outputs weak constraints to minimise program-oriented abnormality tags.
`-minabc`	Outputs weak constraints to minimise completion-oriented abnormality tags.
`-minabl`	Outputs weak constraints to minimise loop-oriented abnormality tags.
`-koall`	Outputs atom `ko(`n_r`)` for every rule r in the given program.
`-nas`	Outputs the number of computed or read answer sets.
`-cig`	Outputs the given program, grounded by `lparse`, having each ground atom replaced by a constant (Constant Intelligent Grounding; CIG). Using flag `-ca`, `spock` provides a table of these constants together with the corresponding atoms.
`-ca`	Outputs a table of constant aliases from a CIG, together with the ground atoms they represent. This list can be used in another invocation of `spock` using flag `-ocr` to re-translate the answer sets of a CIG.
`-ocr`	Outputs all computed or read answer sets of a CIG, having the constant aliases substituted by the corresponding ground atoms, provided that a list of constant aliases was read.
`-dlvarg ARG`	Adds an argument for external calls of `DLV`.
`-lparg ARG`	Adds an argument for external calls of `lparse`.
`-smarg ARG`	Adds an argument for external calls of `Smodels`.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Springer-Verlag (2007) 31–43
3. Delgrande, J., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming **3**(2) (2003) 129–187

4. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
5. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
6. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press (1978) 293–322
7. Free Software Foundation Inc.: GNU General Public License - Version 2, June 1991 (1991) http://www.gnu.org/copyleft/gpl.html
8. 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**(3) (2006) 499–562
9. Syrjänen, T.: Debugging inconsistent answer set programs. In Dix, J., Hunter, A., eds.: Proceedings of the 11th International Workshop on Nonmonotonic Reasoning (NMR'06). Number IFI-06-04 in Technical Report Series, Clausthal University of Technology, Institute for Informatics (2006) 77–83
10. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In De Vos, M., Provetti, A., eds.: Proceedings of the 3rd International Workshop on Answer Set Programming (ASP'05). CEUR Workshop Proceedings (2005) 141–152
11. Pontelli, E., Son, T.: Justifications for logic programs under answer set semantics. In Etalle, S., Truszczyński, M., eds.: Proceedings of the 22nd International Conference on Logic Programming (ICLP'06). Springer-Verlag (2006) 196–210
12. Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I.: Justifying proofs using memo tables. In: Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'00). (2000) 178–189
13. Pemmasani, G., Guo, H., Dong, Y., Ramakrishnan, C., Ramakrishnan, I.: Online justification for tabled logic programs. In Kameyama, Y., Stuckey, P., eds.: Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS'04). Springer-Verlag (2004) 24–38
14. Specht, G.: Generating explanation trees even for negations in deductive database systems. In: Proceedings of the 5th Workshop on Logic Programming Environments (LPE'93). (1993) 8–13
15. El-Khatib, O., Pontelli, E., Son, T.: ASP-PROLOG: A system for reasoning about answer set programs in Prolog. In Delgrande, J., Schaub, T., eds.: Proceedings of the 10th International Workshop on Nonmonotonic Reasoning (NMR'04). (2004) 155–163