TLP **18** (1): 30–80, 2018.
doi:10.1017/S1471068417000217 First published online 17 October 2017

*Stepwise debugging of answer-set programs**

JOHANNES OETSCH

Technische Universität Wien, Institut für Informationssysteme 184/3, Favoritenstrasse 9-11, A-1040 Vienna, Austria,
(*e-mail:* johannes.oetsch@tuwien.ac.at)

JÖRG PÜHRER

Universität Leipzig, Institut für Informatik, Augustusplatz 10, D-04109 Leipzig, Germany,
(*e-mail:* puehrer@informatik.uni-leipzig.de)

HANS TOMPITS

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

submitted 6 July 2016; revised 19 May 2017; accepted 31 May 2017

Abstract

We introduce a *stepping methodology* for answer-set programming (ASP) that allows for debugging answer-set programs and is based on the stepwise application of rules. Similar to debugging in imperative languages, where the behaviour of a program is observed during a step-by-step execution, stepping for ASP allows for observing the effects that rule applications have in the computation of an answer set. While the approach is inspired from debugging in imperative programming, it is conceptually different to stepping in other paradigms due to non-determinism and declarativity that are inherent to ASP. In particular, unlike statements in an imperative program that are executed following a strict control flow, there is no predetermined order in which to consider rules in ASP during a computation. In our approach, the user is free to decide which rule to consider active in the next step following his or her intuition. This way, one can focus on interesting parts of the debugging search space. Bugs are detected during stepping by revealing differences between the actual semantics of the program and the expectations of the user. As a solid formal basis for stepping, we develop a framework of computations for answer-set programs. For fully supporting different solver languages, we build our framework on an abstract ASP language that is sufficiently general to capture different solver languages. To this end, we make use of abstract constraints as an established abstraction for popular language constructs such as aggregates. Stepping has been implemented in `SeaLion`, an integrated development environment for ASP. We illustrate stepping using an example scenario and discuss the stepping plugin of `SeaLion`. Moreover, we elaborate on methodological aspects and the embedding of stepping in the ASP development process.

KEYWORDS: answer-set programming, debugging, stepping, program analysis

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698, the German Research Foundation (DFG) under Grants BR-1817/7-1 and BR 1817/7-2 and the European Commission under project IST-2009-231875 (OntoRule).

1 Introduction

Answer-set programming (ASP) (Marek and Truszczyński 1999; Niemelä 1999) is a paradigm for declarative problem solving that is popular amongst researchers in artificial intelligence and knowledge representation. Yet, it is rarely used by software engineers outside academia so far. Arguably, one obstacle preventing developers from using ASP is a lack of support tools for developing answer-set programs. One particular problem in the context of programming support is *debugging of answer-set programs*. Due to the fully declarative semantics of ASP, it can be quite tedious to detect an error in an answer-set program. In recent years, debugging in ASP has received some attention (Brain and De Vos 2005; Syrjänen 2006; Brain *et al.* 2007b; Pührer 2007; Gebser *et al.* 2008; Gebser *et al.* 2009; Pontelli *et al.* 2009; Oetsch *et al.* 2010a; Oetsch *et al.* 2010b; Oetsch *et al.* 2011; Oetsch *et al.* 2012b; Frühstück *et al.* 2013; Polleres *et al.* 2013; Shchekotykhin 2015). These previous works are important contributions towards ASP development support, however current approaches come with limitations to their practical applicability. First, existing techniques and tools *only capture a basic ASP language fragment* that does not include many language constructs that are available and frequently used in modern ASP solver languages, e.g., aggregates or choice rules are not covered by current debugging strategies (with the exception of the work by Polleres *et al.* (2013), where cardinality constraints are dealt with by translation). Second, *usability aspects are often not considered* in current approaches, in particular, the programmer is required to either provide a lot of data to a debugging system or he or she is confronted with a huge amount of information from the system (tackling this problem in query-based debugging has been addressed by Shchekotykhin (2015)).

This paper introduces a *stepping methodology for ASP*, which is a novel technique for debugging answer-set programs that is general enough to *deal with current ASP solver languages* and is *intuitive and easy to use*. Our method is similar in spirit to the widespread and effective debugging strategy in imperative programming, where the idea is to gain insight into the behaviour of a program by executing statement by statement following the program's control flow. In our approach, we allow for stepwise constructing interpretations by considering rules of an answer-set program at hand in a successive manner. This method guarantees that either an answer set will be reached, or some error will occur that provides hints why the semantics of the program differs from the user's expectations. A major difference to the imperative setting is that, due to its declarativity, ASP lacks any control flow. Instead, we allow the user to follow his or her intuition on which rule instances to become active. This way, one can focus on interesting parts of the debugging search space from the beginning. For illustration, the following answer-set program has $\{a\}$ as its only answer set.

```
a :- not b.
b :- not a.
a :- b.
```

Let's step through the program to obtain explanations why this is the case. In the beginning of a stepping session, no atom is considered to be true. Under this

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

premise, the first two rules are active. The user decides which of these rules to apply. Choosing a rule to be applied in this manner is considered a *step* in our approach. In case the user chooses the first rule, the atom `a` is derived. Then, no further rule is active and one of the answer sets, `{a}` has been reached. If, on the other hand, the user chooses the second rule in the first step, atom `b` is derived and `a` is considered false. Then, the third rule becomes active. However, this rule would derive `a` that is already considered false when choosing the second rule. In this case, the user sees that no answer set can be reached based on the initial choice.

Besides single steps that allow the user to consider one rule instance at a time, we also lay the ground for so-called *jumps*. The intuition is that in a jump multiple rule instances and even multiple non-ground rules can be considered at once. Jumping significantly speeds up the stepping process, which makes our technique a usable tool for debugging in practice. Consider the following encoding of an instance of the three-colouring problem in the `Gringo` language (Gebser *et al.* 2011):

```
% generate
1{color(X,red;green;blue)}1 :- node(X).

% check
:- edge(X,Y), color(X,C), color(X,C).

% instance
node(X):-edge(X,Y).
node(Y):-edge(X,Y).
edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,4). edge(3,5). edge(3,6).
edge(4,5). edge(5,6).
```

The user expects the program to have answer sets but it does not. Following our approach, the reason for that can be found after two actions. First, trusting the 'instance' part of the program, the user applies a jump on all rules of this part, and, intuitively, gets all atoms implied by these rules as an intermediate result. Second, the user applies an arbitrary instance of the rule

```
1{color(X,red;green;blue)}1 :- node(X).
```

that is active under the atoms derived during the jump. Suppose, the user chooses the instance

```
1 {color(1, red), color(1, green), color(1, blue)} 1 :- node(1).
```

and selects `color(1, red)` to be true. Then, the debugging system reveals that the instance

```
:- edge(1, 2), color(1, red), color(1, red).
```

of the 'check' constraint becomes unexpectedly active. Now, the users sees that the second occurrence of `color(X,C)` in the constraint has to be replaced by

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

`color(Y,C)`. Generally, bugs can be detected whenever stepping reveals differences between the actual semantics of the program and the expectations of the user.

In order to establish a solid formal basis for our stepping technique, we developed a framework of computations for answer-set programs. For fully supporting current solver languages, we were faced with several challenges. For one, the languages of answer-set solvers differ from each other and from formal ASP languages in various ways. In order to develop a method that works for different solvers, we need an abstract ASP language that is sufficiently general to capture actual solver languages. To this end, we make use of abstract constraints (Marek and Remmel 2004; Marek and Truszczyński 2004) as an established abstraction for language constructs such as aggregates, weight constraints and external atoms. We rely on a semantics for arbitrary abstract-constraint programs with disjunctions that we introduced for this purpose in previous work (Oetsch *et al.* 2012a). In contrast to other semantics for this type of programs, it is compatible with the semantics of all the ASP solvers we want to support, namely, `Clasp` (Gebser *et al.* 2012), `DLV` (Leone *et al.* 2006) and `DLVHEX` (Redl 2016). Note that our framework for computations for abstract-constraint programs differs from the one by Liu *et al.* (2010). We did not build on this existing notion for three reasons. First, it does not cover rules with disjunctive heads which we want to support. Second, steps in this framework correspond to the application of multiple rules. Since our method is rooted in the analogy to stepping in procedural languages, where an ASP rule corresponds to a statement in an imperative language, we focus on steps corresponding to application of a single rule. Finally, the semantics of non-convex literals differs from that of `DLVHEX` in the existing approach. A thorough discussion on the relation of the two notions of computations is given in Section 5.

Another basic problem deals with the grounding step in which variables are removed from answer-set programs before solving. In formal ASP languages, the grounding of a program consists of all rules resulting from substitutions of variables by ground terms. In contrast, actual grounding tools apply many different types of simplifications and preevaluations for creating a variable-free program. In order to close this gap between formal and practical ASP, Pührer developed abstractions of the grounding step together with an abstract notion of non-ground answer-set program as the base language for the stepping methodology in his PhD thesis (Pührer 2014). Based on that, stepping can easily be applied to existing solver languages and it becomes robust to changes to these languages. As we focus on the methodological aspects of stepping in this paper, we do not present these abstractions and implicitly use grounding as carried out by actual grounding tools.

The stepping technique has been implemented in `SeaLion` (Oetsch *et al.* 2013), an integrated development environment (IDE) for ASP. We discuss how `SeaLion` can be used for stepping answer-set programs written in the `Gringo` or the `DLV` language.[1]

[1] The framework introduced in this paper subsumes and significantly extends previous versions of the stepping technique for normal logic programs (Oetsch *et al.* 2010b; Oetsch *et al.* 2011) and DL-programs (Oetsch *et al.* 2012b).

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

1.1 Outline

Next, we provide the formal background that is necessary for our approach. We recall the syntax of disjunctive abstract-constraint programs and the semantics on which we base our framework (Oetsch *et al.* 2012a). Section 3 introduces a framework of computations that allows for breaking the semantics down to the level of individual rules. After defining states and computations, we show several properties of the framework, most importantly soundness and completeness in the sense that the result of a successful computation is an answer set and that every answer set can be constructed with a computation. Moreover, we study language fragments for which a simpler form of computation suffices. In Section 4, we present the stepping technique for debugging answer-set programs based on our computation framework. We explain *steps* and *jumps* as a means to progress in a computation using an example scenario. Moreover, we discuss methodological aspects of stepping on the application level (how stepping is used for debugging and program analysis) and the top level (how stepping is embedded in the ASP development process). We illustrate the approach with several use cases and describe the stepping interface of `SeaLion`. Related work is discussed in Section 5. We compare stepping to other debugging approaches for ASP and discuss the relation of our computation framework to that of Liu *et al.* (2010) and transition systems for ASP (Lierler 2011; Brochenin *et al.* 2014; Lierler and Truszczyński 2016). We conclude the paper in Section 6.

In supplementary Appendix A we compile guidelines for stepping and give general recommendations for ASP development. Selected and short proofs are included in the main text and all remaining proofs are provided in supplementary Appendix B.

2 Background

As motivated in the introduction, we represent grounded answer-set programs by abstract-constraint programs (Marek and Remmel 2004; Marek and Truszczyński 2004; Oetsch *et al.* 2012a). Non-ground programs will be denoted by programs in the input language of `Gringo`. Thus, we implicitly assume that grounding translates (non-ground) `Gringo` rules to rules of abstract-constraint programs. For a detailed formal account of our framework in the non-ground setting we refer the interested reader to the dissertation of Pührer (2014).

We assume a fixed set $\mathscr{A}$ of ground atoms.

Definition 1
An *interpretation* is a set $I \subseteq \mathscr{A}$ of ground atoms. A ground atom a is *true* under interpretation I, symbolically $I \models a$, if $a \in I$, otherwise it is *false* under I.

We will use the symbol $\not\models$ to denote the complement of a relation denoted with the symbol $\models$ in different contexts.

For better readability, we sometimes make use of the following notation when the reader may interpret the intersection of two sets I and X of ground atoms as a projection from I to X.

Definition 2
For two sets I and X of ground atoms, $I|_X = I \cap X$ is the *projection* of I to X.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

2.1 Syntax of abstract-constraint programs

Rule heads and bodies of abstract-constraint programs are formed by so-called *abstract-constraint atoms*.

Definition 3 (*Marek and Remmel 2004; Marek and Truszczyński 2004*)
An *abstract constraint*, *abstract-constraint atom*, or C-*atom*, is a pair $A = \langle D, C \rangle$, where $D \subseteq \mathscr{A}$ is a finite set called the *domain* of A, denoted by D_A, and $C \subseteq 2^D$ is a collection of sets of ground atoms, called the *satisfiers* of A, denoted by C_A.

We can express atoms also as C-atoms. In particular, for a ground atom a, we identify the C-atom $\langle \{a\}, \{\{a\}\} \rangle$ with a. We call such C-atoms *elementary*.

We will also make use of default negation in abstract-constraint programs. An *abstract-constraint literal*, or C-*literal*, is a C-atom A or a default negated C-atom $not\, A$.

Unlike the original definition, we introduce abstract-constraint programs with disjunctive rule heads.

Definition 4
An *abstract-constraint rule*, or simply C-*rule*, is an expression of the form

$$A_1 \vee \cdots \vee A_k \leftarrow A_{k+1}, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\ , \tag{1}$$

where $0 \leqslant k \leqslant m \leqslant n$ and any A_i, for $1 \leqslant i \leqslant n$, is a C-atom.

Note that if all disjuncts share the same domain they can be expressed by a single C-atom (see Pührer 2014) but in general disjunction adds expressivity.

We identify different parts of a C-rule and introduce some syntactic properties.

Definition 5
For a C-rule r of form (1),

(1) $\mathrm{B}(r) = \{A_{k+1}, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}$ is the *body* of r,
(2) $\mathrm{B}^+(r) = \{A_{k+1}, \ldots, A_m\}$ is the *positive body* of r,
(3) $\mathrm{B}^-(r) = \{A_{m+1}, \ldots, A_n\}$ is the *negative body* of r, and
(4) $\mathrm{H}(r) = \{A_1, \ldots, A_k\}$ is the *head* of r.

If $\mathrm{B}(r) = \emptyset$ and $\mathrm{H}(r) \neq \emptyset$, then r is a C-*fact* . For C-facts, we usually omit the symbol "←". A C-rule r of form (1) is *normal* if $k = 1$ and *positive* if $m = n$.

We define the domain of a default negated C-atom $not\, A$ as $D_{not\ A} = D_A$. Then, the domain D_S of a set S of C-literals is given by

$$D_S = \bigcup_{L \in S} D_L \cdot$$

Finally, the domain of a C-rule r is

$$D_r = \bigcup_{X \in \mathrm{H}(r) \cup \mathrm{B}(r)} D_X \cdot$$

Definition 6
An *abstract-constraint program*, or simply C-*program*, is a finite set of C-rules. A C-program is *normal*, respectively *positive*, if it contains only normal, respectively positive, C-rules. A C-program is *elementary* if it contains only elementary C-atoms.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

2.2 Satisfaction relation

Intuitively, a C-atom $\langle D, C \rangle$ is a literal whose truth depends on the truth of all atoms in D under a given interpretation. The satisfiers in C explicitly list which combinations of true atoms in D make the C-atom true.

Definition 7
An interpretation I *satisfies* a C-atom $\langle D, C \rangle$, symbolically $I \models \langle D, C \rangle$, if $I|_D \in C$. Moreover, $I \models not\ \langle D, C \rangle$ iff $I \not\models \langle D, C \rangle$.

Important criteria for distinguishing classes of C-atoms are concerned with their semantic behaviour with respect to growing (or shrinking) interpretations. In this respect, we identify monotonicity properties in the following.

Definition 8
A C-literal L is *monotone* if, for all interpretations I and I', if $I \subseteq I'$ and $I \models L$, then also $I' \models L$. L is *convex* if, for all interpretations I, I', and I'', if $I \subseteq I' \subseteq I''$, $I \models L$, and $I'' \models L$, then also $I' \models L$. Moreover, a C-program P is monotone (respectively, convex) if for all $r \in P$ all C-literals $L \in \mathrm{H}(r) \cup \mathrm{B}(r)$ are monotone (respectively, convex).

Next, the notion of satisfaction is extended to C-rules and C-programs in the obvious way.

Definition 9
An interpretation I satisfies a set S of C-literals, symbolically $I \models S$, if $I \models L$ for all $L \in S$. For brevity, we will use the notation $I \models^{\exists} S$ to denote that $I \models L$ for some $L \in S$. Moreover, I satisfies a C-rule r, symbolically $I \models r$, if $I \models \mathrm{B}(r)$ implies $I \models^{\exists} \mathrm{H}(r)$. A C-rule r such that $I \models \mathrm{B}(r)$ is called *active under* I. As well, I satisfies a set P of C-rules, symbolically $I \models P$, if $I \models r$ for every $r \in P$. If $I \models P$, we say that I is a *model* of P.

2.3 Viewing ASP constructs as abstract constraints

We want to use abstract constraints as a uniform means to represent common constructs in ASP solver languages. As an example, we recall how weight constraints (Simons *et al.* 2002) can be expressed as C-atoms. In a similar fashion, we can use them as abstractions of e.g., aggregates (Faber *et al.* 2004; Faber *et al.* 2011) or external atoms (Eiter *et al.* 2005). Note that the relation between abstract constraints and ASP constructs is well known and motivated abstract constraints in the first place (cf. Marek and Remmel 2004; Marek and Truszczyński 2004).

Definition 10 (*Simons et al. 2002*)
A *weight constraint* is an expression of form

$$l\ [a_1 = w_1, \ldots, a_k = w_k, not\ a_{k+1} = w_{k+1}, \ldots, not\ a_n = w_n]\ u\,, \tag{2}$$

where each a_i is a ground atom and each weight w_i is a real number, for $1 \leqslant i \leqslant n$. The lower bound l and the upper bound u are either a real number, ∞, or $-\infty$.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

For a weight constraint to be true, the sum of weights w_i of those atoms a_i, $1 \leqslant i \leqslant k$, that are true and the weights of the atoms a_i, $k < i \leqslant n$, that are false must lie within the lower and the upper bound. Thus, a weight constraint of form (2) corresponds to the C-atom $\langle D, C \rangle$, where the domain $D = \{a_1, \ldots, a_n\}$ consists of the atoms appearing in the weight constraint and

$$C = \left\{ X \subseteq D \mid l \leqslant \left(\sum_{1 \leqslant i \leqslant k, a_i \in X} w_i + \sum_{k < i \leqslant n, a_i \notin X} w_i \right) \leqslant u \right\}.$$

2.4 Semantics and characterisations based on external support and unfounded sets

The semantics we use (Oetsch *et al.* 2012a) extends the FLP-semantics (Faber *et al.* 2004, 2011) and coincides with the original notion of answer sets by Gelfond and Lifschitz (1991) on many important classes of logic programs, including elementary C-programs. Similar to the original definition of answer sets, Faber *et al.* make use of a program reduct depending on a candidate interpretation I for determining whether I satisfies a stability criterion and thus is considered an answer set. However, the reduct of Faber, Leone and Pfeifer differs in spirit from that of Gelfond and Lifschitz as it does not reduce the program to another syntactic class (the Gelfond–Lifschitz reduct of an elementary C-program is always positive). Instead, the so-called *FLP-reduct*, defined next, keeps the individual rules intact and just ignores all rules that are not active under the candidate interpretation.

Definition 11
Let I be an interpretation, and let P be a C-program. The FLP-reduct of P with respect to I is given by $P^I = \{r \in P \mid r \text{ is active under } I\}$.

The notion of answer sets for abstract-constraint programs defined next provides the semantic foundation for the computation model we use for debugging.

Definition 12 (*Oetsch et al. 2012a*)
Let P be a C-program, and let I be an interpretation. I is an *answer set* of P if $I \models P$ and there is no $I' \subset I$ such that P, I and I' satisfy the following condition:

($\star$) for every $r \in P^I$ with $I' \models \mathrm{B}(r)$, there is some $A \in \mathrm{H}(r)$ with $I' \models A$ and $I'|_{D_A} = I|_{D_A}$.

The set of all answer sets of P is denoted by $AS(P)$.

The purpose of Condition ($\star$) is to prevent minimisation within C-atoms: the requirement $I'|_{D_A} = I|_{D_A}$ ensures that a satisfier $\{a, b\}$ can enforce b to be true in an answer set even if the same C-atom has a satisfier $\{a\}$. As a consequence answer sets need not be subset minimal (see Pührer 2014 for details).

Our choice of semantics has high solver compatibility as its objective as we want to support `Gringo`, `DLV` and `DLVHEX`. We need an FLP-style treatment of non-convex literals for being compatible with `DLVHEX`, disjunctions to support `DLV` and `DLVHEX`, and we must allow for weight constraints in rule heads for compatibility with

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

`Gringo`. Note that `Gringo/Clasp` treats aggregates in the way suggested by Ferraris (2011). As a consequence, its semantics differs from our semantics in some cases, when recursion is used through negated C-atoms, as ours is an extension of the FLP semantics. For an in-depth comparison of FLP-semantics and Ferraris semantics we refer to work by Truszczyński (2010). An example where the semantics differ is given by the single rule `Gringo` program `a :- not 0{a}0` that has only the empty set as answer set under our semantics, whereas `Clasp` also admits $\{a\}$ as an answer set. In practice, this difference only hardly influences the compatibility with `Gringo`, as aggregates are seldom used in this way. We examined all `Gringo` encodings send to the second ASP competition and could not find any such usage.

Our framework of computations for stepping is based on a characterisation of the semantics of Definition 12 in terms of *external supports*. Often, answer sets are computed following a two-step strategy: First, a model of the program is built, and second, it is checked whether this model obeys a foundedness condition ensuring that it is an answer set. Intuitively, every set of atoms in an answer set must be 'supported' by some active rule that derives one of the atoms. Here, it is important that the reason for this rule to be active does not depend on the atom it derives. Such rules are referred to as *external support* (Lee 2005). The extension of this notion to our setting is the following.

Definition 13 (*Oetsch et al. 2012a*)
Let r be a C-rule, X a set of atoms, and I an interpretation. Then, r is an *external support for X with respect to I* if

(i) $I \models \mathrm{B}(r)$,
(ii) $I \setminus X \models \mathrm{B}(r)$,
(iii) there is some $A \in \mathrm{H}(r)$ with $X|_{D_A} \neq \emptyset$ and $I|_{D_A} \subseteq S$, for some $S \in C_A$, and
(iv) for all $A \in \mathrm{H}(r)$ with $I \models A$, $(X \cap I)|_{D_A} \neq \emptyset$ holds.

Condition (i) ensures that r is active. Condition (ii) prevents self-support by guaranteeing the support to be "external" of X, i.e., r is also be active without the atoms in X. In case I is a model, Items (iii) and (iv) jointly ensure that there is some C-atom A in the head of r that is satisfied by I and derives some atom of X.

We can express the absence of an external support in an interpretation by the concept of an *unfounded set*.

Definition 14 (*Oetsch et al. 2012a*)
Let P be a C-program, X a set of atoms, and I an interpretation. Then, X is *unfounded in P with respect to I* if there is no C-rule $r \in P$ that is an external support for X with respect to I.

Corollary 1 (*Oetsch et al. 2012a*)
Let P be a C-program and I an interpretation. Then, I is an answer set of P iff I is a model of P and there is no set X with $\emptyset \subset X \subseteq I$ that is unfounded in P with respect to I.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

3 Computation framework

In this section, we want to break the conceptual complexity of the semantics down to artefacts the programmer is familiar with: the rules the user has written or, more precisely, their ground instances. To this end, we introduce a framework of computations that captures the semantics described in the previous section. In this computation model, on top of which we will introduce stepping in Section 4, an interpretation is built up step-by-step by considering an increasing number of rule instances to be active. A *computation* in our framework is a sequence of *states*, which are structures that keep information which rules and atoms have already been considered and what truth values were assigned to those atoms. Utilising the framework, only one rule and the atoms it contains have to be considered at once, while building up an interpretation until an answer set is reached or a source for the unexpected behaviour becomes apparent.

In the next two subsections, we introduce states and computations. In Section 3.3, we define and show some properties of computations that we need later on when we describe stepping. Section 3.4 is concerned with the existence of a stable computation which is a simpler form of computation that suffices for many popular classes of answer-set programs. We discuss existing work related to our computation framework later in Section 5.

3.1 States

Our framework is based on sequences of states, reassembling computations, in which an increasing number of ground rules are considered that build up a monotonically growing interpretation. Besides that interpretation, states also capture literals which cannot become true in subsequent steps and sets that currently lack external support in the state's interpretation.

Definition 15
A *state structure* S is a tuple $\langle P, I, I^-, \Upsilon \rangle$, where P is a set of C-rules, I is an interpretation, I^- a set of atoms such that I and I^- are disjoint, and Υ is a collection of sets of atoms. We call $D_S = I \cup I^-$ the *domain* of S and define $P_S = P$, $I_S = I$, $I^-_S = I^-$, and $\Upsilon_S = \Upsilon$.

A state structure $\langle P, I, I^-, \Upsilon \rangle$ is a *state* if

(i) $I \models \mathrm{B}(r)$ and $I \models^{\exists} \mathrm{H}(r)$ for every $r \in P$,
(ii) $D_r \subseteq D_S$ for every $r \in P$, and
(iii) $\Upsilon = \{X \subseteq I \mid X \text{ is unfounded in } P \text{ with respect to } I\}$.

We call $\langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$ the *empty state*.

Intuitively, we use the first component P of a state to collect C-rules that the user has considered to be active and satisfied. The interpretation I collects atoms that have been considered true. Condition (i) ensures that P and I are compatible in the sense that every C-rule that is considered active and satisfied is active and satisfied with respect to I. Dual to I, the interpretation I^- collects atoms that the user has considered to be false. We require that all atoms appearing in a C-rule in P is either

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

in I or in I^-, which is expressed in Condition (ii). Finally, the set Υ keeps track of unfounded subsets of I, as stated in Condition (iii). Intuitively, as we will see later, when building a computation, the aim is to get rid of all unfounded sets (except for the empty set) in order to compute an answer set of a C-program. If a state does not contain such unfounded sets then we call it stable:

Definition 16
A state S is *stable* if $I_S \in AS(P_S)$.

The intuition is that when a state S is stable, no more C-rules need to be added to P_S to provide missing external support for the atoms in the current interpretation I_S. Note that a state S is stable exactly when $\Upsilon_S = \{\emptyset\}$. For example, the empty state is a stable state.

Example 1
Consider the C-rules

$$r_1 : \langle\{a,b\},\{\emptyset,\{a\},\{b\},\{a,b\}\rangle \leftarrow not\ a$$
$$r_2 : \qquad\qquad\qquad\qquad\qquad\qquad b \leftarrow a$$

and the state structures

$$S_1 = \langle\{r_1\},\emptyset,\{a,b\},\{\emptyset\}\rangle, \qquad S_2 = \langle\{r_1\},\{b\},\{a\},\{\emptyset\}\rangle,$$
$$S_3 = \langle\{r_1\},\{a,b\},\emptyset,\{\emptyset\}\rangle, \qquad S_4 = \langle\{r_2\},\{a,b\},\emptyset,\{\emptyset\}\rangle,$$
$$S_5 = \langle\{r_2\},\{a,b\},\emptyset,\{\{b\},\{a,b\}\}\rangle.$$

S_1 and S_2 are stable states. S_3 is not a state as $I_{S_3} \not\models \mathrm{B}(r_1)$. S_4 is not a state as the sets $\{b\}$ and $\{a,b\}$ are unfounded in P_{S_4} with respect to I_{S_4} but $\{b\} \notin \Upsilon_{S_4}$ and $\{a,b\} \notin \Upsilon_{S_4}$. S_5 is a state but not stable.

3.2 Computations

Next, we show how we can proceed forward in a computation, i.e., which states might follow a given state using a successor relation for state structures.

Definition 17
For a state $S = \langle P, I, I^-, \Upsilon\rangle$ and a state structure $S' = \langle P', I', I'^-, \Upsilon'\rangle$, S' is a *successor of* S if there is a C-rule $r \in P' \setminus P$ and sets $\Delta, \Delta^- \subseteq D_r$ such that

(i) $P' = P \cup \{r\}$,
(ii) $I' = I \cup \Delta$, $I'^- = I^- \cup \Delta^-$, and $D_S \cap (\Delta \cup \Delta^-) = \emptyset$,
(iii) $D_r \subseteq D_{S'}$,
(iv) $I \models \mathrm{B}(r)$,
(v) $I' \models \mathrm{B}(r)$ and $I' \models^{\exists} \mathrm{H}(r)$, and
(vi) $X' \in \Upsilon'$ iff $X' = X \cup \Delta'$, where $X \in \Upsilon$, $\Delta' \subseteq \Delta$, and r is not an external support for X' with respect to I'.

We denote r by $r_{new}(S, S')$.

Condition (i) ensures that a successor state considers exactly one rule more to be active. Conditions (ii) and (iii) express that the interpretations I and I^- are

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

extended by the so far unconsidered literals in Δ and Δ^- appearing in the new C-rule $r_{new}(S, S')$. Note that from S' being a state structure we get that Δ and Δ^- are distinct. A requirement for considering $r_{new}(S, S')$ as next C-rule is that it is active under the current interpretation I, expressed by Condition (iv). Moreover, $r_{new}(S, S')$ must be satisfied and still be active under the succeeding interpretation, as required by Condition (v). The final condition ensures that the unfounded sets of the successor are extensions of the previously unfounded sets that are not externally supported by the new rule.

Here, it is interesting that only extended previous unfounded sets can be unfounded sets in the extended C-program P' and that $r_{new}(S, S')$ is the only C-rule which could provide external support for them in P' with respect to the new interpretation I' as seen next.

Theorem 1
Let S be a state and S' a successor of S, where $\Delta = I_{S'} \setminus I_S$. Moreover, let X' be a set of literals with $\emptyset \subset X' \subseteq I_{S'}$. Then, the following statements are equivalent:

(i) X' is unfounded in $P_{S'}$ with respect to $I_{S'}$.
(ii) $X' = \Delta' \cup X$, where $\Delta' \subseteq \Delta$, $X \in \Upsilon_S$, and $r_{new}(S, S')$ is not an external support for X' with respect to $I_{S'}$.

The result shows that determining the unfounded sets in a computation after adding a further C-rule r can be done locally, i.e., only supersets of previously unfounded sets can be unfounded sets, and if such a superset has some external support then it is externally supported by r. The result also implies that the successor relation suffices to "step" from one state to another.

Corollary 2
Let S be a state and S' a successor of S. Then, S' is a state.

Proof
We show that the Conditions (i), (ii), and (iii) of Definition 15 hold for S'. Consider some rule $r \in P_{S'}$. In case $r = r_{new}(S, S')$, $I_{S'} \models \mathrm{B}(r)$ and $I_{S'} \models^{\exists} \mathrm{H}(r)$ hold because of Item (v) of Definition 17 and $D_r \subseteq D_{S'}$ because of Item (iii) of the same definition. Moreover, in case $r \neq r_{new}(S, S')$, we have $r \in P_S$. As S is a state, we have $D_r \subseteq D_S$. Hence, since $D_S \subseteq D_{S'}$ also $D_r \subseteq D_{S'}$. Note that $I_{S'}|_{D_r} = I_S|_{D_r}$ because of Item (ii) of Definition 17. Therefore, as $I_S \models \mathrm{B}(r)$ and $I_S \models^{\exists} \mathrm{H}(r)$, also $I_{S'} \models \mathrm{B}(r)$ and $I_{S'} \models^{\exists} \mathrm{H}(r)$. From these two cases, we see that Conditions (i) and (ii) of Definition 15 hold for S'. Finally, Condition (iii) follows from Item (vi) of Definition 17 and Theorem 1. □

Next, we define computations based on the notion of a state.

Definition 18
A *computation* is a sequence $C = S_0, \ldots, S_n$ of states such that S_{i+1} is a successor of S_i, for all $0 \leqslant i < n$. We call C *rooted* if S_0 is the empty state and *stable* if each S_i is stable, for $0 \leqslant i \leqslant n$.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

3.3 Properties

We next define when a computation has failed, gets stuck, is complete, or has succeeded. Intuitively, failure means that the computation reached a point where no answer set of the C-program can be reached. A computation is stuck when the last state activated rules deriving literals that are inconsistent with previously chosen active rules. It is considered complete when there are no more unconsidered active rules. Finally, a computation has succeeded if an answer set has been reached.

Definition 19
Let P be a C-program and $C = S_0, \ldots, S_n$ a computation such that $P_{S_n} \subseteq P$. Then, C is called a *computation for* P. Moreover,

- C has *failed for* P *at step* i if there is no answer set I of P such that $I_{S_i} \subseteq I$, $I^-{}_{S_i} \cap I = \emptyset$, and $P_{S_i} \subseteq P^I$;
- is *complete for* P if for every rule $r \in P^{I_{S_n}}$, we have $r \in P_{S_n}$;
- is *stuck in* P if it is not complete for P but there is no successor S_{n+1} of S_n such that $r_{new}(S_n, S_{n+1}) \in P$;
- *succeeded for* P if it is complete and S_n is stable.

Example 2
Let P_{Ex2} be the C-program consisting of the C-rules

$$\begin{array}{lll} r_1: & & a \leftarrow \langle\{a,b\},\{\emptyset,\{a,b\}\}\rangle \\ r_2: & & b \leftarrow a \\ r_3: & & a \leftarrow b \\ r_4: & \langle\{c\},\{\emptyset,\{c\}\}\rangle & \leftarrow \\ r_5: & & \leftarrow c \end{array}$$

that has $\{a,b\}$ as its single answer set, and consider the sequences

- $C_1 = \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle,$
  $\langle\{r_4\},\{\},\{c\},\{\emptyset\}\rangle,$
  $\langle\{r_4,r_1\},\{a,b\},\{c\},\{\{a\},\{b\}\}\rangle,$
- $C_2 = \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle, \langle\{r_4\},\{\},\{c\},\{\emptyset\}\rangle, \langle\{r_4,r_1\},\{a,b\},\{c\},\{\{a\},\{b\}\}\rangle,$
  $\langle\{r_4,r_1,r_2\},\{a,b\},\{c\},\{\{a\}\}\rangle, \langle\{r_4,r_1,r_2,r_3\},\{a,b\},\{c\},\{\emptyset\}\rangle,$
- $C_3 = \langle\{r_4,r_1,r_2,r_3\},\{a,b\},\{c\},\{\emptyset\}\rangle,$
- $C_4 = \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle, \langle\{r_4\},\{c\},\emptyset,\{\emptyset\}\rangle,$
- $C_5 = \langle\{r_4,r_1,r_2,r_3\},\{a,b,c\},\emptyset,\{\emptyset\}\rangle,$
- $C_6 = \langle\{r_5\},\emptyset,\{c\},\{\emptyset\}\rangle,$ and
- $C_7 = \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle, \langle\{r_4,r_1\},\{a,b\},\{c\},\{\{a\},\{b\}\}\rangle.$

C_1, C_2, C_3, C_4 and C_5 are computations for P_{Ex2}. The sequence C_6 is not a computation, as $\langle\{r_5\},\emptyset,\{c\},\{\emptyset\}\rangle$ is not a state. C_7 is not a computation, as the second state in C_7 is not a successor of the empty state. C_1, C_2 and C_4 are rooted. C_3, C_4 and C_5 are stable. C_2 and C_3 are complete and have succeeded for P_{Ex2}. C_1 is complete for $P_{Ex2} \setminus \{r_2, r_3\}$ but has failed for $P_{Ex2} \setminus \{r_2, r_3\}$ at Step 0 because

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

$P_{Ex2} \setminus \{r_2, r_3\}$ has no answer set. C_4 has failed for P_{Ex2} at Step 1. C_5 has failed for P_{Ex2} at Step 0 and is stuck in P_{Ex2}.

The following result guarantees the soundness of our framework of computations.

Theorem 2
Let P be a C-program and $C = S_0, \ldots, S_n$ a computation that has succeeded for P. Then, I_{S_n} is an answer set of P.

Proof
As C is complete for P, we have $P^{I_{S_n}} \subseteq P_{S_n}$. Conversely, we have $P_{S_n} \subseteq P^{I_{S_n}}$ because for each $r \in P_{S_n}$ we have $r \in P$ and $I_{S_n} \models \mathrm{B}(r)$. By stability of S_n, we get that $I_{S_n} \in AS(P_{S_n})$. The conjecture holds since then $I_{S_n} \in AS(P^{I_{S_n}})$. □

The computation model is also complete in the following sense:

Theorem 3
Let S_0 be a state, P a C-program with $P_{S_0} \subseteq P$, and I an answer set of P with $I_{S_0} \subseteq I$ and $I \cap I^-_{S_0} = \emptyset$. Then, there is a computation $S_0, \ldots, S_n$ that has succeeded for P such that $P_{S_n} = P^I$ and $I_{S_n} = I$.

As the empty state, $\langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$, is trivially a state, we can make the completeness aspect of the previous result more apparent in the following corollary:

Corollary 3
Let P be a C-program and $I \in AS(P)$. Then, there is a rooted computation $S_0, \ldots, S_n$ that has succeeded for P such that $P_{S_n} = P^I$ and $I_{S_n} = I$.

Proof
The claim follows immediately from Theorem 3 in case $S_0 = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. □

Note, that there are states that do not result from rooted computations, e.g., the state $\langle \{a \leftarrow b\}, \{a, b\}, \emptyset, \{\emptyset, \{a, b\}, \{b\}\} \rangle$ is not a successor of any other state. However, for stable states, we can guarantee the existence of rooted computations.

Corollary 4
Let S be a stable state. Then, there is a rooted computation $S_0, \ldots, S_n$ with $S_n = S$.

Proof
The result is a direct consequence of Corollary 3 and Definition 16. □

The next theorem lays the ground for the jumping technique that we introduce in Section 4. It allows for extending a computation by considering multiple rules of a program at once and using ASP solving itself for creating this extension.

Theorem 4
Let P be a C-program, $C = S_0, \ldots, S_n$ a computation for P, P' a set of C-rules with $P' \subseteq P$, and I an answer set of $P_{S_n} \cup P'$ with $I_{S_n} \subseteq I$ and $I \cap I^-_{S_n} = \emptyset$. Then, there is a computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ for P, such that S_m is stable, $P_{S_m} = P_{S_n} \cup P'^I$ and $I_{S_m} = I$.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

Proof
By Theorem 3, as $P_{S_n} \subseteq P_{S_n} \cup P'$, $I_{S_n} \subseteq I$, and $I \cap I^-_{S_n} = \emptyset$, there is a computation $S_n, \ldots, S_m$ that has succeeded for $P_{S_n} \cup P'$ such that $P_{S_m} = (P_{S_n} \cup P')^I$ and $I_{S_m} = I$. Then, S_m is stable and, as ${P_{S_n}}^I = P_{S_n}$, we have $P_{S_m} = P_{S_n} \cup P'^I$. As $P_{S_m} \subseteq P$, we have that $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ is a computation for P. □

The following result illustrates that the direction one chooses for building up a certain interpretation, i.e., the order of the rules considered in a computation, is irrelevant in the sense that eventually the same state will be reached.

Proposition 1
Let P be a C-program and $C = S_0, \ldots, S_n$ and $C' = S'_0, \ldots, S'_m$ computations complete for P such that $S_0 = S'_0$. Then, $I_{S_n} = I_{S'_m}$ iff $S_n = S'_m$ and $n = m$.

Proof
The 'if' direction is trivial. Let $I = I_{S_n} = I_{S'_m}$. Towards a contradiction, assume $P_{S_n} \neq P_{S'_m}$. Without loss of generality, we focus on the case that there is some $r \in P_{S_n}$ such that $r \notin P_{S'_m}$. Then, it holds that $I \models r$, $I \models \mathrm{B}(r)$, and $r \in P$. Consequently, $r \in P^I$. By completeness of C', we have $r \in P_{S'_m}$ which contradicts our assumption. Hence, we have $P_{S_n} = P_{S'_m}$.

By definition of a state, from $I_{S_n} = I_{S'_m}$ and $P_{S_n} = P_{S'_m}$, it follows that $\Upsilon_{S_n} = \Upsilon_{S'_m}$. Towards a contradiction, assume $I^-_{S_n} \neq I^-_{S'_m}$. Without loss of generality we focus on the case that there is some $a \in I^-_{S_n}$ such that $a \notin I^-_{S'_m}$. Consider the integer i where $0 < i \leqslant n$ such that $a \in I^-_{S_i}$ but $a \notin I^-_{S_{i-1}}$. Then, by definition of a successor, for $r = r_{new}(S_{i-1}, S_i)$, we have $a \in \Delta^-$ for some $\Delta^- \subseteq D_r$. As then $a \in D_r$ and, as $P_{S_n} = P_{S'_m}$, we have $r \in P_{S'_m}$, it must hold that $a \in D_{S'_m}$ by definition of a state structure. From $I \cap I^-_{S_n} = \emptyset$ we know that $a \notin I$. Therefore, since $a \in I \cup I^-_{S'_m}$, we get that $a \in I^-_{S'_m}$, being a contradiction to our assumption. As then $S_n = S'_m$, $P_{S_0} = P_{S'_0}$, and since in every step in a computation exactly one rule is added it must hold that $n = m$. □

For rooted computations, the domain of each state is determined by the atoms in the C-rules it contains.

Proposition 2
Let $C = S_0, \ldots, S_n$ be a rooted computation. Then, $I_{S_i} = I_{S_n}|_{D_{P_{S_i}}}$ and $I^-_{S_i} = I^-_{S_n}|_{D_{P_{S_i}}}$, for all $0 \leqslant i \leqslant n$.

Proof
The proof is by contradiction. Let j be the smallest index with $0 \leqslant j \leqslant n$ such that $I_{S_j} \neq I_{S_n}|_{D_{P_{S_j}}}$ or $I^-_{S_j} \neq I^-_{S_n}|_{D_{P_{S_j}}}$. Note that $0 < j$ as $I_{S_0} = I^-_{S_0} = D_{P_{S_0}} = \emptyset$. As S_j is a successor of S_{j-1}, we have $I_{S_j} = I_{S_{j-1}} \cup \Delta$ and $I^-_{S_j} = I^-_{S_{j-1}} \cup \Delta^-$, where $\Delta, \Delta^- \subseteq D_{r_{new}(S_{j-1}, S_j)}$, $D_{S_{j-1}} \cap (\Delta \cup \Delta^-) = \emptyset$, and $D_{r_{new}(S_{j-1}, S_j)} \subseteq I_{S_j} \cup I^-_{S_j}$. As we have $I_{S_{j-1}} = I_{S_n}|_{D_{P_{S_{j-1}}}}$ and $I^-_{S_{j-1}} = I^-_{S_n}|_{D_{P_{S_{j-1}}}}$, it holds that

$$I_{S_{j-1}} \cup I_{S_n}|_{D_\delta} = I_{S_n}|_{D_{P_{S_{j-1}}}} \cup I_{S_n}|_{D_\delta} = I_{S_n}|_{D_{P_{S_j}}} \text{ and}$$
$$I^-_{S_{j-1}} \cup I^-_{S_n}|_{D_\delta} = I^-_{S_n}|_{D_{P_{S_{j-1}}}} \cup I^-_{S_n}|_{D_\delta} = I^-_{S_n}|_{D_{P_{S_j}}},$$

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

where $D_\delta = D_{P_{S_j}} \setminus D_{P_{S_{j-1}}}$. For establishing the contradiction, it suffices to show that $I_{S_n}|_{D_\delta} = \Delta$ and $I^-{}_{S_n}|_{D_\delta} = \Delta^-$. Consider some $a \in \Delta$. Then, $a \in D_\delta$ because $a \in D_{r_{new}(S_{j-1},S_j)}$, $D_{S_{j-1}} \cap (\Delta \cup \Delta^-) = \emptyset$, and $D_{P_{S_{j-1}}} \subseteq D_{S_{j-1}}$. Moreover, $a \in I_{S_j}$ implies $a \in I_{S_n}$ and therefore $\Delta \subseteq I_{S_n}|_{D_\delta}$. Now, consider some $b \in I_{S_n}|_{D_\delta}$. As $D_{r_{new}(S_{j-1},S_j)} \subseteq I_{S_j} \cup I^-{}_{S_j}$, we have $b \in I_{S_j} \cup I^-{}_{S_j}$. Consider the case that $b \in I^-{}_{S_j}$. Then, also $b \in I^-{}_{S_n}$ which is a contradiction to $b \in I_{S_n}$ as S_n is a state structure. Hence, $b \in I_{S_j} = I_{S_{j-1}} \cup \Delta$. First, assume $b \in I_{S_{j-1}}$. This leads to a contradiction as then $b \in D_{P_{S_{j-1}}}$ since $I_{S_{j-1}} = I_{S_n}|_{D_{P_{S_{j-1}}}}$. It follows that $b \in \Delta$ and therefore $\Delta = I_{S_n}|_{D_\delta}$. One can show that $\Delta^- = I^-{}_{S_n}|_{D_\delta}$ analogously. □

3.4 Stable computations

In this section, we are concerned with the existence of stable computations, i.e., computations that do not involve unfounded sets. We single out an important class of C-programs for which one can solely rely on this type of computation and also give examples of C-programs that do not allow for succeeding stable computations.

Intuitively, the Σ_2^P-hardness of the semantics (cf. Pührer 2014), demands for unstable computations in the general case. This becomes obvious when considering that for a given C-program one could guess a candidate sequence C for a stable computation in polynomial time. Then, a polynomial number of checks whether each state is a successor of the previous one in the sequence suffices to establish whether C is a computation. Following Definition 17, these checks can be done in polynomial time when we are allowed to omit Condition (vi) for unfounded sets. Hence, answer-set existence for the class of C-programs for which every answer set can be built up with stable computations is in NP.

Naturally, it is interesting whether there are syntactic classes of C-programs for which we can rely on stable computations only. It turns out that many syntactically simple C-programs already require the use of unfounded sets.

Example 3
Consider C-program P_{Ex3} consisting of the C-rules

$$r_1 : a \leftarrow b \qquad \text{and}$$
$$r_2 : b \leftarrow \langle \{a\}, \{\emptyset, \{a\}\} \rangle.$$

We have that $\{a, b\}$ is the only answer set of P_{Ex3} and

$$\begin{aligned} C = &\langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle, \\ &\langle \{r_2\}, \{a, b\}, \emptyset, \{\emptyset, \{a\}\} \rangle, \\ &\langle \{r_2, r_1\}, \{a, b\}, \emptyset, \{\emptyset\} \rangle \end{aligned}$$

is the only computation that succeeds for P_{Ex3}: starting at the empty state, only rule r_2 is active, thus it must be the new rule in the successor. When, deciding the truth values for the atoms in D_{r_2}, r_2 requires b to be positive, and a must be true as well, as otherwise the computation is stuck due to violation of r_1. The second state of C contains the singleton $\{a\}$ as unfounded set.

As Example 3 shows, unstable computations are already required for a C-program without disjunction and a single monotone C-atom. Hence, also the use of weaker

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

restrictions, like convexity of C-atoms or some notion of head-cycle freeness (Ben-Eliyahu and Dechter 1994), is not sufficient.

One can observe, that the C-program from the example has cyclic positive dependencies between atoms a and b. Hence, we next explore whether such dependencies influence the need for computations that are not stable. To this end, we introduce notions of *positive dependency* in a C-program.

Definition 20
Let S be a set of C-literals. Then, the *positive normal form* of S is given by

$$S^+ = \{A \mid A \in S, A \text{ is a C-atom}\} \cup \{\bar{A} \mid not\ A \in S\},$$

where $\bar{A} = \langle D_A, 2^{D_A} \setminus C_{D_A} \rangle$ is the *complement* of A. Furthermore, the *set of positive atom occurrences* in S is given by $posOcc(S) = \bigcup_{A \in S^+, X \in C_A} X$.

Let P be a C-program. The *positive dependency graph* of P is the directed graph

$$G(P) = \langle D_P, \{\langle a, b\rangle \mid r \in P, a \in posOcc(\mathrm{H}(r)), b \in posOcc(\mathrm{B}(r))\}\rangle.$$

We next introduce the notion of *absolute tightness* for describing C-programs without cyclic positive dependencies after recalling basic notions of graph theory. For a (directed) graph $G = \langle V, \prec \rangle$, the *reachability relation* of G is the transitive closure of $\prec$. Let $\prec'$ be the reachability relation of G. Then, G is *acyclic* if there is no $v \in V$ such that $v \prec' v$.

Definition 21
Let P be a C-program. P is *absolutely tight* if $G(P)$ is acyclic.

One could assume that absolute tightness paired with convexity or monotonicity is sufficient to guarantee stable computations because absolute tightness forbids positive dependencies among disjuncts and the absence of such dependencies lowers the complexity of elementary C-programs (Ben-Eliyahu and Dechter 1994). However, as the following example illustrates, this is not the case for general C-programs.

Example 4
Consider C-program P_{Ex4} consisting of the C-rules

$$r_1 : a \vee \langle \{a, b\}, \{\{a\}, \{a, b\}\}\rangle \leftarrow$$
$$r_2 : b \vee \langle \{a, b\}, \{\{b\}, \{a, b\}\}\rangle \leftarrow$$

We have that $\{a, b\}$ is the only answer set of P_{Ex4} and

$$\begin{aligned} C_1 = &\langle \emptyset, \emptyset, \emptyset, \{\emptyset\}\rangle, \\ &\langle \{r_1\}, \{a, b\}, \emptyset, \{\emptyset, \{b\}\}\rangle, \\ &\langle \{r_1, r_2\}, \{a, b\}, \emptyset, \{\emptyset\}\rangle \qquad \text{and} \end{aligned}$$

$$\begin{aligned} C_2 = &\langle \emptyset, \emptyset, \emptyset, \{\emptyset\}\rangle, \\ &\langle \{r_2\}, \{a, b\}, \emptyset, \{\emptyset, \{a\}\}\rangle, \\ &\langle \{r_1, r_2\}, \{a, b\}, \emptyset, \{\emptyset\}\rangle \end{aligned}$$

are the only computations that succeed for P_{Ex4}. Clearly, P_{Ex4} is monotone and absolutely tight but C_1 and C_2 are not stable.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

Nevertheless, we can assure the existence of stable computations for answer sets of normal C-programs that are absolutely tight and convex. This is good news, as this class corresponds to a large subset of typical answer-set programs written for solvers like `Clasp` that do not rely on disjunction as their guessing device.

Theorem 5
Let $C = S_0, \ldots, S_n$ be a computation such that S_0 and S_n are stable and $P_\Delta = P_{S_n} \setminus P_{S_0}$ is a normal, convex, and absolutely tight C-program. Then, there is a stable computation $C' = S'_0, \ldots, S'_n$ such that $S_0 = S'_0$ and $S_n = S'_n$.

As a direct consequence of Theorem 5 and Corollary 3, we get an improved completeness result for normal convex C-programs that are absolutely tight, i.e., we can find a computation that consists of stable states only.

Corollary 5
Let P be a normal C-program that is convex and absolutely tight, and consider some $I \in AS(P)$. Then, there is a rooted stable computation $S_0, \ldots, S_n$ such that $P_{S_n} = P^I$ and $I_{S_n} = I$.

Proof
From $I \in AS(P)$, we get by Corollary 3 that there is a rooted computation $S_0, \ldots, S_n$ such that $P_{S_n} = P^I$ and $I_{S_n} = I$. Note that S_0 is the empty state. S_0 and S_n are stable according to Definition 16. From Theorem 5, we can conclude the existence of another computation $C' = S'_0, \ldots, S'_n$ such that $S_0 = S'_0$ and $S_n = S'_n$ that is stable. Clearly, C' is also rooted. □

4 Theory and practice of stepping

In this section we present our methodology for stepping answer-set programs based on the computation model introduced in the previous section.

Step-by-step execution of a program is common practice in procedural programming languages, where developers can debug and investigate the behaviour of their programs in an incremental way. The technique introduced in this work shows how this popular form of debugging can be applied to ASP, despite the genuine declarative semantics of answer-set programs that lacks a control flow. Its main application is debugging but it is also beneficial in other contexts such as improving the understanding of a given answer-set program or teaching the answer-set semantics to beginners.

For stepping to be a practical support technique for answer-set programmers rather than a purely theoretical approach, we assume the *availability of a support environment* that assists a user in a stepping session. A prototype of our stepping framework has been implemented in `SeaLion`, an IDE for ASP (Oetsch *et al.* 2013). It was developed as one of the major goals of a research project on methods and methodologies for developing answer-set programs conducted at Vienna University of Technology (2009–2013). `SeaLion` supports the ASP languages of `Gringo` and `DLV` and comes as a plugin of the Eclipse platform that is popular for Java development.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

All the features of a stepping support environment described in this section are implemented in SeaLion, if not stated otherwise.

To bridge the gap between theory and practical stepping, our examples use solver syntax rather than the abstract language of C-programs. We implicitly identify solver constructs with their abstract counterparts (cf. Section 2.3). While we use C-programs as a formal lingua franca for different solver languages, we do not require the user to know about it. Likewise, we do not expect a user to be aware of the specifics of the computation framework of Section 3 that provides the backbone of stepping. For example, the user does not need to know the properties of Definition 17. The debugging environment automatically restricts the available choices a user has when performing a step to ones that result in valid successor states.

In the following subsection, we introduce an example scenario that we use later on. In Section 4.2, we describe the general idea of stepping for ASP. There are two major ways for navigating in a computation in our framework: performing *steps*, discussed in Section 4.3, and *jumps* that we describe in Section 4.4. In Section 4.5, we describe the stepping interface of SeaLion. Building on steps and jumps, we discuss methodological aspects of stepping for debugging purposes in Section 4.6 and provide a number of use cases.

4.1 Example scenario - maze generation

Next, we introduce the problem of *maze generation* that serves as a running example in the remainder of the paper. It has been a benchmark problem of the second ASP competition (Denecker *et al.* 2009) to which it was submitted by Martin Brain. The original problem description is available on the competition's website.[2]

As the name of the problem indicates, the task we deal with is to generate a maze, i.e., a labyrinth structure in a grid that satisfies certain conditions. In particular, we deal with two-dimensional grids of cells where each cell can be assigned to be either an empty space or a wall. Moreover, there are two (distinct) empty squares on the edge of the grid, known as the entrance and the exit. A path is a finite sequence of cells, in which each distinct cell appears at most once and each cell is horizontally or vertically adjacent to the next cell in the sequence.

Such a grid is a valid maze if it meets the following criteria:

(1) Each cell is a wall or is empty.
(2) There must be a path from the entrance to every empty cell (including the exit).
(3) If a cell is on any of the edges of the grid, and is not an entrance or an exit, it must contain a wall.
(4) There must be no 2×2 blocks of empty cells or walls.
(5) No wall can be completely surrounded by empty cells.
(6) If two walls are diagonally adjacent then one or other of their common neighbours must be a wall.

[2] http://dtai.cs.kuleuven.be/events/ASP-competition/Benchmarks/MazeGeneration.shtml

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

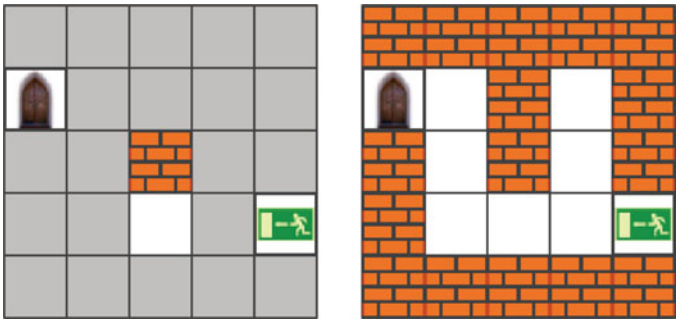

Fig. 1. Left: A grid visualising an instance of the maze generation where white squares represent empty cells, whereas grey squares are yet undefined. Right: A solution for the instance on the left.

The maze generation problem is the problem of completing a two-dimensional grid in which some cells are already decided to be empty or walls and the entrance and the exit are predefined to a valid maze. An example of a problem instance and a corresponding solution maze is depicted in Figure 1.

Next, we describe the predicate schema that we use for ASP maze generation encodings. The predicates `col/1` and `row/1` define the columns and rows in the grid, respectively. They are represented by a range of consecutive, ascending integers, starting at `1`. The positions of the entrance and the exit are determined by predicates `entrance/2` and `exit/2`, respectively, where the first argument is a column index and the second argument is a row index. In a similar manner, `empty/2` and `wall/2` determine which cells are empty or contain walls. For example, the instance of Figure 1 can be encoded by program Π_1 consisting of the following facts:

```
col(1..5). row(1..5).
entrance(1,2). exit(5,4). wall(3,3). empty(3,4).
```

Moreover, the solution in the figure could be represented by the following interpretation (projected to predicates `empty/2` and `wall/2`):

```
{wall(1,1), empty(1,2), wall(1,3), wall(1,4), wall(1,5),
 wall(2,1), empty(2,2), empty(2,3), empty(2,4), wall(2,5),
 wall(3,1), wall(3,2), wall(3,3), empty(3,4), wall(3,5),
 wall(4,1), empty(4,2), empty(4,3), empty(4,4), wall(4,5),
 wall(5,1), wall(5,2), wall(5,3), empty(5,4), wall(5,5)}
```

4.2 General idea

We introduce stepping for ASP as a strategy to identify mismatches between the intended semantics of an answer-set program under development and its actual semantics. Due to the declarativity of ASP, once one detects unintended semantics, it can be a tough problem to manually detect the reason. Stepping is a method for breaking this problem into smaller parts and structuring the search for an error. The general idea is to monotonically build up an interpretation by, in each step, adding

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

literals derived by a rule that is active with respect to the interpretation obtained in the previous step. The process is interactive in the sense that at each such step the user chooses the active rule to proceed with and decides which literals of the rule should be considered true or false in the target interpretation. Hereby, the user only adds rules he or she thinks are active in an expected or an unintended actual answer set. The interpretation grows monotonically until it is eventually guaranteed to be an answer set of the overall program, otherwise the programmer is informed why and at which step something went wrong. This way, one can in principle without any backtracking direct the computation towards the interpretation one has in mind. In debugging, having the programmer in the role of an oracle is a common scenario as it is reasonable to assume that a programmer has good intuitions on where to guide the search (Shapiro 1982). We use the computation model of Section 3 to ensure that, if the interpretation specified in this way is indeed an answer set, the process of stepping will eventually terminate with the interpretation as its result. Otherwise, the computation will fail at some step where the user gets insight why the interpretation is not an answer set, e.g., when a constraint becomes irrevocably active or no further rule is active that could derive some desired literal.

4.3 Steps

By a *step* we mean the extension of a computation by a further state. We consider a setting, where a programmer has written an answer-set program in a solver language for which C-program P it the abstraction of its grounding. Moreover, we assume that the programmer has obtained some computation for P that is neither stuck in P nor complete for P. For performing a step, one needs to find a successor state S_{n+1} for S_n such that $C' = S_0, \ldots, S_{n+1}$ is a computation for P.

We propose a sequence of three user actions to perform a step. Intuitively, for quickly finding a successor state (with the help of the debugging environment), we suggest to

(1) select a non-ground rule with active ground instances, then
(2) choose an active ground rule among the instances of the non-ground rule, and
(3) select for yet undefined atoms in the domain of the ground instance whether they are considered true or false.

First, the user selects a non-ground rule ρ. In `SeaLion`, this can be done by directly selecting ρ in the editor in which the program was written. The debugging system can support this user action by automatically determining the subset of rules in the program that have at least one instance r in their grounding that could lead to a successor state, i.e., $r = r_{new}(S_n, S)$ for some successor S of S_n.

Then, the user selects an instance r from the grounding of ρ. As the ground instances of ρ are not part of the original program, picking one requires a different approach as for choosing ρ. Here, the debugging environment can display the ground rules in a dedicated area and, as before, restrict the choice of rule groundings of ρ to ones that lead to a successor state. Filtering techniques can be used to restrict the

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

amount of the remaining instances. In `SeaLion`, the user defines partial assignments for the variables in ρ that determine a subset of the considered instances.

In the third user action for performing a step, the programmer chooses the truth values for the atoms in D_r that are neither in I_{S_n} nor in $I^-_{S_n}$. This choice must be made in a way such that there is a successor S_{n+1} of S_n with $P_{S_{n+1}} = P_{S_n} \cup \{r\}$, $I_{S_{n+1}} = I_{S_n} \cup \Delta$, and $I^-_{S_{n+1}} = I^-_{S_n} \cup \Delta^-$, where Δ contains the atoms the user chose to be true and Δ^- the atoms considered false. That is, S_n, Δ, and Δ^- must fulfil the conditions of Definition 17. Here, the user needs only to ensure that Condition (v) of Definition 17 holds, i.e., $I_{S_{n+1}} \models \mathrm{B}(r)$ and $I_{S_{n+1}} \models^{\exists} \mathrm{H}(r)$, as the other conditions automatically hold once all unassigned atoms have been assigned to Δ and Δ^-. In particular, the set of unfounded sets, $\Upsilon_{S_{n+1}}$ can always be automatically computed following Condition (vi) of Definition 17 and does not impose restrictions on the choice of Δ and Δ^-. The support system can check whether Condition (v) holds for the truth assignment specified by the user. Also, atoms are automatically assigning to Δ or Δ^- whenever their truth values are the same for all successor states that are based on adding r.

Example 5

As a first step for developing the maze-generation encoding, we want to identify border cells and guess an assignment of walls and empty cells. Our initial program is Π_2, given next.

```
maxCol(X) :- col(X), not col(X+1).
maxRow(Y) :- row(Y), not row(Y+1).
border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).

wall(X,Y) :- border(X,Y), not entrance(X,Y), not exit(X,Y).
{ wall(X,Y) : col(X), row(Y), not border(X,Y) }.
empty(X,Y) :- col(X), row(Y), not wall(X,Y).
```

The first two rules extract the numbers of columns and rows of the maze from the input facts of predicates `col/1` and `row/1`. The next four rules derive `border/2` atoms that indicate which cells form the border of the grid. The final three rules derive `wall/2` atoms for border cells except entrance and exit, guess `wall/2` atoms for the remaining cells, and derive `empty/2` atoms for non-wall cells, respectively.

We use Π_2 in conjunction with the facts in program Π_1 (defined in Section 4.1) that determine the problem instance.

We start a stepping session with the computation $C_0 = S_0$ consisting of the empty state $S_0 = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. Following the scheme of user actions described above for performing a step, we first look for a non-ground rule with instances that are active under I_{S_0}. As $I_{S_0} = \emptyset$, only the facts from Π_1 have active instances. We choose the rule `entrance(1,2).` In this case, the only (active) instance of the rule is identical to the rule, i.e., the fact:

```
entrance(1,2).
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

The only atom in the domain of the rule instance is `entrance(1,2)`. Therefore, when performing the final user action for a step one has to decide the truth value of this atom. In order to fulfil Condition (v) of Definition 17, the rule head, i.e., `entrance(1,2)`, must be true in the successor state. Thus, our first step results in the computation $C_1 = S_0, S_1$ where

$$S_1 = \langle \{\texttt{entrance(1,2).}\}, \{\texttt{entrance(1,2)}\}, \emptyset, \{\emptyset\} \rangle.$$

For the next step, we choose the rule

```
col(1..5).
```

from Π_1. The grounding[3] by `Gringo` consists of the following instances:

```
col(1). col(2). col(3). col(4). col(5).
```

We select the instance `col(5).` Since the head of the rule must be true under the successor state, as before, atom `col(5)` must be considered true in the successor state of S_1. The resulting computation after the second step is $C_2 = S_0, S_1, S_2$, where

$$S_2 = \langle \{\texttt{entrance(1,2).col(5).}\}, \{\texttt{entrance(1,2), col(5)}\}, \emptyset, \{\emptyset\} \rangle.$$

Under I_{S_2} a further rule in $\Pi_1 \cup \Pi_2$ has active instances:

```
maxCol(X) :- col(X), not col(X+1).
```

That is, it has the active instance

```
maxCol(5) :- col(5), not col(6).
```

that we choose for the next step. In order to ensure that Condition (v) of Definition 17 is satisfied, we need to ensure that head and body are satisfied under the successor state. Hence, atom `maxCol(5)` has to be considered true, whereas `col(6)` must be considered false. We obtain the computation $C_3 = S_0, S_1, S_2, S_3$, where

$$S_3 = \langle \{\texttt{entrance(1,2).col(5).maxCol(5) :- col(5), not col(6).}\}, \{\texttt{entrance(1,2)}, \texttt{col(5)}, \texttt{maxCol(5)}\}, \{\texttt{col(6)}\}, \{\emptyset\} \rangle.$$

4.4 Jumps

If one wants to simulate the computation of an answer set I in a stepping session using steps only, as many steps are necessary as there are active rules in the grounding under I. Although typically the number of active ground instances is much less than the total number of rules in the grounding, still many rules would have to be considered. In order to focus on the parts of a computation that the user is interested in, we introduce a *jumping* technique for quickly considering rules that are of minor interest, e.g., for rules that are already considered correct. We say that we *jump through* these rules. By performing a jump, we mean to find a state

[3] Remember that grounding refers to the translation performed by the grounding tool rather than mere variable elimination.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

that could be reached by a computation for the program at hand that extends the current computation by possibly multiple states. If such a state can be found, one can continue to expand a computation from that while it is ensured that the same states could be reached by using steps only. Jumps can be performed exploiting Theorem 4. In essence, jumping is done as follows.

(1) Select rules that you want to jump through (i.e., the rules you want to be considered in the state to jump to),
(2) an auxiliary answer-set program is created that contains the selected rules and the active rules of the current computations final state, and
(3) a new state is computed from an answer set of the auxiliary program.

Next, we describe the items in more detail. We assume that an answer-set program in a solver language for which C-program P it the abstraction of its grounding and a computation $S_0, \ldots, S_n$ for P are given.

The first user action is to select a subset P' of P, the rules to jump through. Hence, an implication for a stepping support environment is the necessity of means to select the ground instances that form P'. In case the user wants to consider all instances of some non-ground rules, a very user friendly way of selecting these is to simply select the non-ground rules and the environment implicitly considers P' to be their instances. `SeaLion` implements this feature such that this selection can be done in the editor in which the answer-set program is written. If the user wants to jump through only some instances of a non-ground rule, we need further user-interface features. In order to keep memory resources and the amount of rules that have to be considered by the user low, the system splits the selection of an instance in two phases. First, the user selects a non-ground rule ρ, similar as in the first user action of defining a step. Then, the system provides so far unconsidered rules of P for selection, where similar filtering techniques as sketched for the second user action for performing a step can be applied.

The auxiliary program of the second item can be automatically computed. That is a C-program P_{aux} given by $P_{aux} = P_{S_n} \cup P' \cup P_{con}$, where

$$P_{con} = \{\leftarrow \mathit{not}\ a \mid a \in I_{S_n}\} \cup \{\leftarrow a \mid a \in I^-_{S_n}\}$$

is a set of constraints that ensure that for every answer set I of P_{aux} we have $I_{S_n} \subseteq I$ and $I \cap I^-_{S_n} = \emptyset$.

After computing an answer set I of the auxiliary program, Theorem 4 ensures the existence of a computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ for P such that S_m is stable and $I_{S_m} = I$. Moreover, we have $P_{S_m} = P_{S_n} \cup P'^I$. Then, the user can proceed with further steps or jumps extending the computation $S_0, \ldots, S_m$ as if S_m had been reached by steps only.

Note that non-existence of answer sets of the auxiliary program does not imply that the overall program has no answer sets as shown next.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

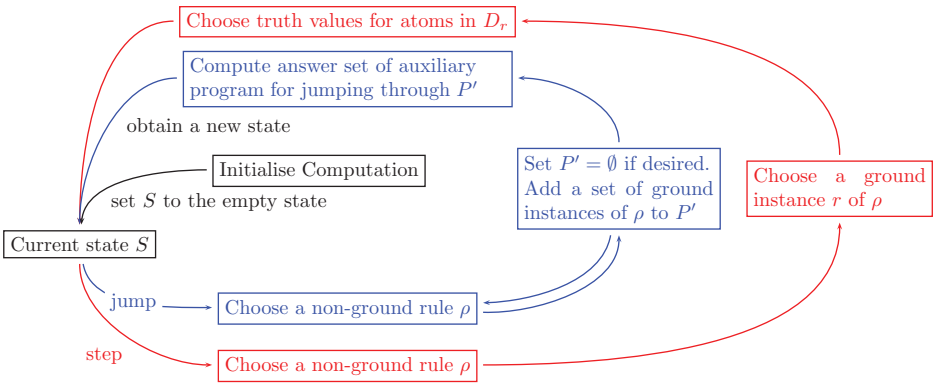

Fig. 2. Stepping cycle.

Example 6

Consider a program P consisting of the C-rules $a \leftarrow$ and $\leftarrow$ *not* a that has $\{a\}$ as its unique answer set. Assume we want to jump through the second rule starting from the computation $C = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$ consisting of the empty state. Then, $P_{aux} = \{\leftarrow \textit{not } a\}$ has no answer set.

The example shows that jumping only makes sense when the user is interested in a computation reaching an answer set of the auxiliary program. In case of multiple answer sets of the auxiliary program, the user could pick any or a stepping environment can choose one at random. For practical reasons, the second option seems more preferable. On the one hand, presenting multiple answer sets to the user can lead to a large amount of information that has to be stored and processed by the user. And on the other hand, if the user is not happy with the truth value of some atoms in an arbitrary answer set of the auxiliary program, he or she can use steps to define the truth of these atoms before performing the jump. In `SeaLion` only one answer set of the auxiliary program is computed.

The iterative extension of a computation using steps and jumps can be described as a *stepping cycle* that is depicted in Figure 2. It summarises how a user may advance a computation and thus provides a technical level representation of stepping.

Example 7

We continue computation C_3 for program $\Pi_1 \cup \Pi_2$ from Example 5. As we are interested in the final three rules of Π_2 that derive `empty/2` and `wall/2` atoms but these rules depend on atoms of predicate `border/2`, `entrance/2` and `exit/2` that are not yet considered in C_3, we jump through the facts from Π_1 and the rules

```
maxCol(X) :- col(X), not col(X+1).
maxRow(Y) :- row(Y), not row(Y+1).
border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

of program Π_2. The resulting auxiliary program Π_3 is given by the following rules (for non-ground rules, their unconsidered instances in the grounding of $\Pi_1 \cup \Pi_2$).

```
% P_S3
entrance(1,2).
col(5).
maxCol(5) :- col(5), not col(6).

% Pi':
col(1..5). row(1..5).
exit(5,4). wall(3,3). empty(3,4).
maxCol(X) :- col(X), not col(X+1).
maxRow(Y) :- row(Y), not row(Y+1).
border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).

% P_con
:- not entrance(1,2).
:- not col(5).
:- not maxCol(5).
:- col(6).
```

The program Π_3 has the single answer set I_{aux} consisting of the atoms:

```
col(1), col(2), col(3), col(4), col(5), maxCol(5),
row(1), row(2), row(3), row(4), row(5), maxRow(5),
empty(3,4), wall(3,3), entrance(1,2), exit(5,4),
border(1,1), border(2,1), border(3,1), border(4,1),
border(5,1), border(1,2), border(5,2), border(1,3),
border(5,3), border(1,4), border(5,4), border(1,5),
border(2,5), border(3,5), border(4,5), border(5,5),
```

We obtain the new state $S_4 = \langle P_{S_4}, I_{aux}, D_{P_{S_4}} \setminus I_{aux}, \{\emptyset\}\rangle$, where P_{S_4} consists of the following rules:

```
col(1). col(2). col(3). col(4). col(5).
row(1). row(2). row(3). row(4). row(5).
wall(3,3). empty(3,4). entrance(1,2). exit(5,4).
maxCol(5) :- col(5), not col(6).
maxRow(5) :- row(5), not row(6).
border(1,1) :- col(1), row(1).
border(2,1) :- col(2), row(1).
border(3,1) :- col(3), row(1).
border(4,1) :- col(4), row(1).
border(5,1) :- col(5), row(1).
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

```
border(1,2) :- col(1), row(2).
border(5,2) :- row(2), maxCol(5).
border(1,3) :- col(1), row(3).
border(5,3) :- row(3), maxCol(5).
border(1,4) :- col(1), row(4).
border(5,4) :- row(4), maxCol(5).
border(1,5) :- col(1), row(5).
border(5,1) :- row(1), maxCol(5).
border(5,5) :- row(5), maxCol(5).
border(1,5) :- col(1), maxRow(5).
border(2,5) :- col(2), maxRow(5).
border(3,5) :- col(3), maxRow(5).
border(4,5) :- col(4), maxRow(5).
border(5,5) :- col(5), maxRow(5).
```

Theorem 4 ensures the existence of a computation $C_4 = S_0, S_1, S_2, S_3, \ldots, S_4$ for program $\Pi_1 \cup \Pi_2$.

4.5 *Stepping interface of* SeaLion

In the following, we focus on the stepping functionality of SeaLion that was implemented by our former student Peter Skočovský. While it is the first implementation of the stepping technique for ASP and hence still a prototype, it is tailored for intuitive and user-friendly usage and able to cope with real-world answer-set programs. The stepping feature is integrated with the Kara plugin of SeaLion (Kloimüllner *et al.* 2013) that can create user-defined graphical representations of interpretations. Thus, besides visualising answer sets, it is also possible to visualise intermediate states of a stepping session. Visualisations in Kara are defined using ASP itself, for further information we refer to earlier work (Kloimüllner *et al.* 2013). A comprehensive discussion of other features of SeaLion is given in a related paper (Busoniu *et al.* 2013) on the IDE. SeaLion is published under the GNU General Public License and can be obtained from www.sealion.at

A stepping session in SeaLion can be started in a similar fashion as debugging Java programs in Eclipse using the launch configuration framework. SeaLion launch configurations that are used for defining which program files should be run with which solvers can be re-used as *debug configurations*.

Like many IDEs, Eclipse comes with a multiple document interface in which inner frames, in particular Eclipse editors and views, can be arranged freely by the user. Such configurations can be persisted as *perspectives*. Eclipse plugins often come with default perspectives, i.e., arrangements of views and editors that are tailored to a specific user task in the context of the plugin. Also the stepping plugin has a preconfigured perspective that is opened automatically once a stepping session has been initiated. The next subsection gives an overview of the individual stepping related subframes in this perspective.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

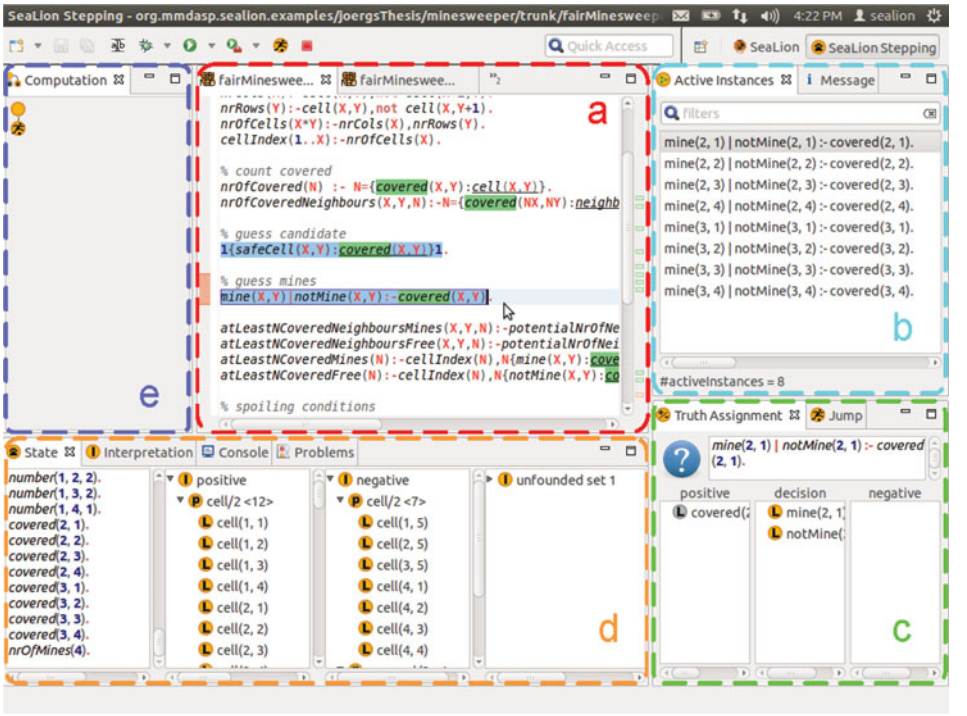

Fig. 3. `SeaLion`'s stepping view is divided into five areas (a–e).

Figure 3 shows `SeaLion` in the *stepping perspective*. The illustration distinguishes five regions (marked by supplementary dashed frames and labelled by letters) for which we give an overview in what follows.

The source code editor (Figure 3, a) is the same as used for writing answer-set programs but extended with additional functionality during stepping mode for the ASP files involved in the stepping session. In particular it indicates rules with ground instances that are active under the interpretation of the current stepping state. Constraints with active instances are highlighted by a red background (cf. Figure 9), other rules with active instances have a blue background (as, e.g., in Figure 4). The editor remains functional during stepping, i.e., the program can be modified while debugging. Note, however, that the system does not guarantee that the current computation is still a valid computation in case of a modification of the answer-set program after stepping has been initiated. The source code editor is also the starting point for performing a step or a jump as it allows for directly selecting the non-ground rule(s) to be considered in the step or jump in the source code. The choice of non-ground rules corresponds to the initial step in the stepping cycle (see Section 4.4). Selecting a single rule or consecutive rules is done by directly selecting them in the source code editor. If the rules are non-consecutive, the user must collect rules in the *jump view* located in area c of Figure 3 as the second tab.

Choosing a ground instance for performing a step is done in the *active instances view* (Figure 3, b). It contains a list with all active ground instances (with respect to conditional grounding) of the currently selected rule in the source editor. As these

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

are potentially many, the view has a textfield for filtering the list of rules. Filters are given as dot-separated list of variable assignments of the form `X=t` where `X` is a variable of the non-ground rule and `t` is the ground term that the user considers `X` to be assigned to. Only ground instances are listed that obey all variable substitutions of the entered filters.

Once a rule instance is selected in the active instances view, the atoms in the rule's domain are displayed in three lists of the *truth assignment view* (Figure 3, c). The list in the centre shows atoms whose truth value has not already been determined in the current state. The user can decide whether they should be true, respectively false, in the next step by putting them into the list on the left, respectively, on the right. These atoms can be transferred between the lists by using keyboard cursors or drag-and-drop (Figure 4). After the truth value has been decided for all the atoms of the rule instance and only in case that the truth assignment leads to a valid successor state (cf. Definition 17), a button labelled 'Step' appears. Clicking this button computes the new state.

The state view (Figure 3, d) shows the current stepping state of the debugging session. Hence, it is updated after every step or jump. It comprises four areas, corresponding to the components of the state (cf. Definition 15), the list of active rules instances, a tree-shaped representation of the atoms considered true, a tree-shaped representation of the atoms considered false, both in a similar graphical representation as that of interpretations in the interpretation view, and an area displaying the unfounded sets in a similar way. The sets of atoms displayed in this view can also be visualised using `Kara` (via options in the context menu).

Finally, the computation view (Figure 3, e) gives an overview of the steps and jumps performed so far. Importantly, the view implements an undo–redo mechanism. That is, by clicking on one of the nodes displayed in the view, representing a previous step or jump, the computation can be reset to the state after this step or jump has been performed. Moreover, after performing an undo operation, the undone computation is not lost but becomes an inactive branch of the tree-shaped representation of steps and jumps. Thus, one can immediately jump back to any state that has been reached in the stepping session by clicking on a respective node in the tree (Figure 6).

Mismatches between the users intentions (reflected in the current stepping state) and the actual semantics of the program can be detected in different parts of the stepping perspective. If the user thinks a rule instance should be active but it is not, this can already be seen in the source code editor if the non-ground version of the rule does not have any active instance. Then, the rule is not highlighted in the editor. If the non-ground version does have active instances but not the one the user has in mind, this can be detected after clicking on the non-ground rule if they are missing in the active instances view. The computation is stuck if only rules are highlighted in the source editor that are constraints (cf. Figure 9) or for all of its instances, no truth assignment can be established such that the 'Step' button appears.

Finally, if no further rule is highlighted and there is no non-empty unfounded set visible in the state view, the atoms considered positive form an answer set of the overall program. If there are further unfounded sets, the user sees that the

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

constructed interpretation is not stable. The unfounded sets indicate which atoms would need external support.

4.6 Methodology

We identify three conceptual levels of stepping as a methodology. The *technical level* corresponds to the iterative advancement of a computation covered in Sections 4.3 and 4.4 summarised in the *stepping cycle* (Figure 2). Next, we describe stepping on the *application level* as a method for *debugging* and *program analysis*. After that, we highlight how stepping is *embedded in* the greater context of *ASP development* from a *top level* perspective. Finally, we illustrate our approach in different usage scenarios. In supplementary Appendix A, we compile practical guidelines for our methodology.

4.6.1 Program analysis and debugging level methodology

The main purpose for stepping in the context of this paper is its application for debugging and analysing answer-set programs. Next, we describe how insight into a program is gained using stepping. During stepping, the user follows his or her intuitions on which rule(s) to apply next and which atoms to consider true or false. In this way, an interpretation is built up that either is or is not an answer set of the program. In both cases, stepping can be used to analyse the interplay of rules in the program in the same manner, i.e., one can see which rule instances become active or inactive after each step or jump. In the case that the targeted interpretation is an answer set of the program, the computation will never fail (in the sense of Definition 19) or get stuck and will finally succeed. It can, however, happen that intermediate states in the computation are unstable (cf. Example 3). For debugging, stepping towards an answer set is useful if the answer set is unwanted. In particular, one can see why a constraint (or a group of rules supposed to have a constraining effect) does not become active. For instance, stepping reveals other active rules that derive atoms that make some literal in the constraint false or rules that fail do derive atoms that activate the constraint. Stepping towards an actual answer set of a program is illustrated in Example 8.

In the case that there is an answer set that the user expects to be different, i.e., certain atoms are missing or unwanted, it makes sense to follow the approach that we recommend for expected but missing answer sets, i.e., stepping towards the interpretation that the user wants to be an answer set. Then, the computation is guaranteed to fail at some point, i.e., there is some state in the computation from which no more answer set of the program can be reached. In other situations, the computation can already have failed before the bug can be found, e.g., the computation can have failed from the beginning in case the program has no answer sets at all. Nevertheless, the error can be found when stepping towards the intended interpretation. In most cases, there will be either a rule instance that becomes active that the user considered inactive, or the other way around, i.e., a rule instance never becomes active or is deactivated while the computation progresses. Eventually, due

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

to our completeness results in the previous section, the computation will either get stuck or ends in an unstable state S such that no active external support for a non-empty unfounded set from Υ_S is available in the program's grounding. Stepping towards an interpretation that is not an answer set of the overall program can be seen as a form of hypothetical reasoning: the user can investigate how rules of a part of the program support each other before adding a further rule would cause an inconsistency. Example 9 illustrates stepping towards an intended but non-existing answer set for finding a bug. Another illustration of hypothetical reasoning is given in Example 10 where a user tries to understand why an interpretation that is not supposed to be an answer set is indeed no answer set.

Note that stepping does not provide explanation artefacts, analogous to stepping in imperative languages, where insight in a problem is gained by simply following the control flow and watching variable assignments. In our setting, the user only sees which remaining rule instances are active and the current state of the computation which in principle suffices to notice differences to his or her expectations. Nevertheless, it can be useful to combine this approach with query based debugging methods that provide explicit explanations as will be discussed in Section 5.

As mentioned in Section 4.2, if one has a clear idea on the interpretation one expects to be an answer set, stepping allows for building up a computation for this interpretation without backtracking. In practice, one often lacks a clear vision on the truth value of each and every atom with respect to a desired answer set. As a consequence, the user may require revising the decisions he or she has taken on the truth values of atoms as well as on which rules to add to the computation. For this reason, `SeaLion` allows for retracting a computation to a previous state, i.e., let the user select one of the states in the computation and continue stepping from there. This way a tree of states can be built up (as in Figure 9), where every path from the root node to a leaf node is a rooted computation.

4.6.2 Top-level methodology

Stepping must be understood as embedded in the programming and modelling process, i.e., the technique has to be recognised in the *context* of developing answer-set programs. A practical consequence of viewing stepping in the big picture are several possibilities for exploiting information obtained during the development of a program for doing stepping faster and more accurate.

While an answer-set program evolves, the programmer will in many cases compute answer sets of preliminary versions of the program for testing purposes. If this answer sets are persisted, they can often be used as a starting point for stepping sessions for later versions of the program. For instance, in case the grounding P of a previous version of a program is a subset of the current grounding P' it is obvious that a successful computation C for P is also a computation for P'. Hence, the user can initiate a stepping session starting from C. Also in case that $P \not\subset P'$, a stepping support system could often automatically build a computation that uses an answer set of P (or parts of it) as guidance for deciding on the rules

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

to add and the truth values to assign in consecutive states. Likewise, (parts of) computations of stepping sessions for previous versions of a program can be stored and re-used either as a computation of the current program if applicable or for computing such a computation. The idea is that previous versions of a program often constitute a part of the current version that one is already familiar with and 'trusts' in. By starting from a computation that already considers a well-known part of the program, the user can concentrate on new and often more suspicious parts of the program. Currently, there is no such feature implemented in `SeaLion`.

4.6.3 Use cases

Next, we show application scenarios of stepping using our running example.

The first scenario illustrates stepping towards an interpretation that is an answer set of the program under consideration.

Example 8

We want to step towards an answer set of our partial encoding of the maze generation problem, i.e., of the program $\Pi_1 \cup \Pi_2$. Therefore, we continue our stepping session with computation C_4, i.e., we start stepping from state S_4 that we obtained in Example 7. In particular, we want to reach an answer set that is compatible with the maze generation solution depicted in Figure 1. To this end, we start with stepping through the active instances of the rule

```
{wall(X,Y) : col(X), row(Y), not border(X,Y)}.
```

The only active instance of the rule is

```
{wall(2,2), wall(3,2), wall(4,2), wall(2,3), wall(3,3),
            wall(4,3), wall(2,4), wall(3,4), wall(4,4)}.
```

Thus, we next choose a truth assignment for the atoms appearing in the instance's choice atom. Note that we do not need to decide for the truth value of `wall(3,3)` as it is already contained in I_{S_4} and therefore already considered true. As can be observed in Figure 1, among the remaining cells the rule deals with, only the one at position $(3,2)$ is a wall in our example. Hence, we obtain a new state S_5 from S_4 by extending P_{S_4} by our rule instance, I_{S_4} by `wall(3,2)`, and $I^-_{S_4}$ by `wall(2,2)`, `wall(4,2)`, `wall(2,3)`, `wall(4,3)`, `wall(2,4)`, `wall(3,4)`, `wall(4,4)`. As in S_4, the empty set is the only unfounded set in state S_5. It remains to jump through the rules

```
wall(X,Y) :- border(X,Y), not entrance(X,Y), not exit(X,Y).
```

and

```
empty(X,Y) :- col(X), row(Y), not wall(X,Y).
```

that leads to the addition of instances

```
wall(1, 1) :- border(1, 1), not entrance(1, 1), not exit(1, 1).
wall(2, 1) :- border(2, 1), not entrance(2, 1), not exit(2, 1).
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

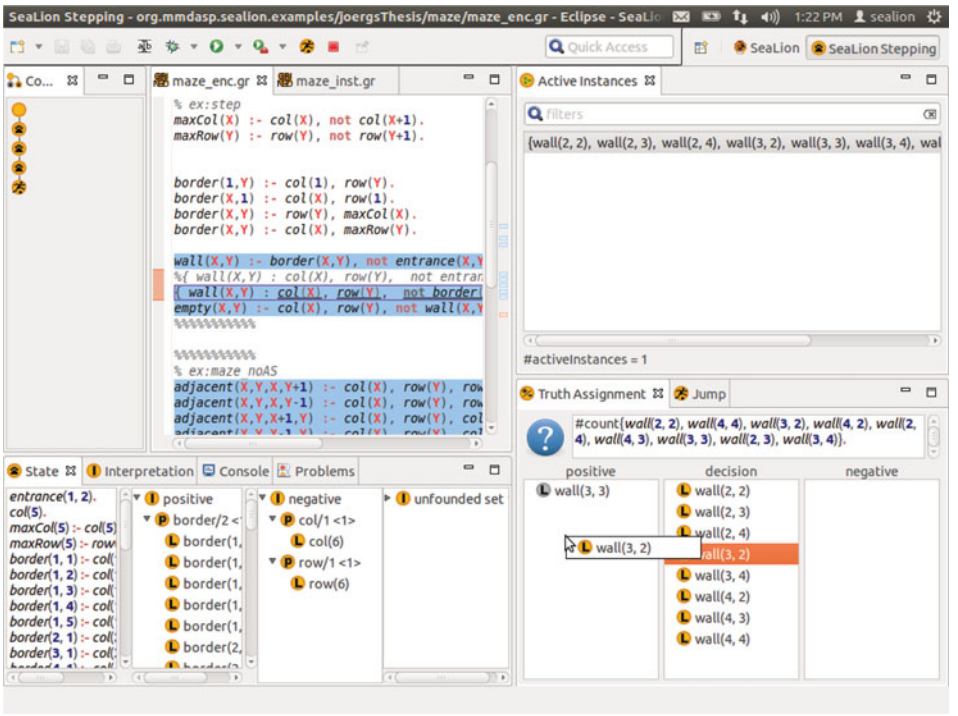

Fig. 4. Deciding to consider atom `wall(3,2)` to be true by dragging it from the middle list of atoms in the truth assignment view and dropping it in the left list. The current state of the stepping session is S_4 from Example 8 and the step prepared is that to state S_5. Note that atom `wall(3,3)` is greyed in the list of positive atoms. Greyed atoms in the positive or negative list cannot be dragged away from there again because their truth value is already considered positive, respectively negative, in the current state (here S_4). A step can only be completed once the truth value has been decided for all atoms in the rule instance.

```
wall(3, 1) :- border(3, 1), not entrance(3, 1), not exit(3, 1).
wall(4, 1) :- border(4, 1), not entrance(4, 1), not exit(4, 1).
wall(5, 1) :- border(5, 1), not entrance(5, 1), not exit(5, 1).
wall(5, 2) :- border(5, 2), not entrance(5, 2), not exit(5, 2).
wall(1, 3) :- border(1, 3), not entrance(1, 3), not exit(1, 3).
wall(5, 3) :- border(5, 3), not entrance(5, 3), not exit(5, 3).
wall(1, 4) :- border(1, 4), not entrance(1, 4), not exit(1, 4).
wall(1, 5) :- border(1, 5), not entrance(1, 5), not exit(1, 5).
wall(2, 5) :- border(2, 5), not entrance(2, 5), not exit(2, 5).
wall(3, 5) :- border(3, 5), not entrance(3, 5), not exit(3, 5).
wall(4, 5) :- border(4, 5), not entrance(4, 5), not exit(4, 5).
wall(5, 5) :- border(5, 5), not entrance(5, 5), not exit(5, 5).
empty(1, 2) :- col(1), row(2), not wall(1, 2).
empty(2, 2) :- col(2), row(2), not wall(2, 2).
empty(4, 2) :- col(4), row(2), not wall(4, 2).
empty(2, 3) :- col(2), row(3), not wall(2, 3).
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

```
empty(4, 3) :- col(4), row(3), not wall(4, 3).
empty(2, 4) :- col(2), row(4), not wall(2, 4).
empty(3, 4) :- col(3), row(4), not wall(3, 4).
empty(4, 4) :- col(4), row(4), not wall(4, 4).
empty(5, 4) :- col(5), row(4), not wall(5, 4).
```

to a new state S_6. I_{S_6} extends I_{S_5} by the head atoms of these rules that are not yet in I_{S_5}. Likewise, $I^-_{S_6}$ extends $I^-_{S_5}$ by the default negated atoms appearing in the rules that are not yet in $I^-_{S_5}$. As $\Upsilon_{S_6} = \{\emptyset\}$ and no rule in $\Pi_1 \cup \Pi_2$ has further active instances under I_{S_6}, the computation $S_0, \ldots, S_6$ has succeeded and hence I_{S_6} is an answer set of the program.

In the next example, a bug is revealed by stepping towards an intended answer set.

Example 9

As a next feature, we (incorrectly) implement rules in program Π_4 that should express that there has to be a path from the entrance to every empty cell and that 2×2 blocks of empty cells are forbidden:

```
adjacent(X,Y,X,Y+1) :- col(X), row(Y), row(Y+1).
adjacent(X,Y,X,Y-1) :- col(X), row(Y), row(Y-1).
adjacent(X,Y,X+1,Y) :- col(X), row(Y), col(X+1).
adjacent(X,Y,X-1,Y) :- col(X), row(Y), col(X-1).
reach(X,Y)   :- entrance(X,Y), not wall(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y), not wall(XX,YY).

:- empty(X,Y), not reach(X,Y).
:- empty(X,Y), empty(X+1,Y), empty(X,X+1), empty(X+1,Y+1).
```

The first six rules formalise when an empty cell is reached from the entrance, and the two constraints should ensure that every empty cell is reached and that no 2×2 blocks of empty cells exist, respectively.

Assume that we did not spot the bug in the second constraint—in the third body literal the term `Y+1` was mistaken for `X+1`. This could be the result of a typical copy–paste error. It turns out that $\Pi_1 \cup \Pi_2 \cup \Pi_4$ has no answer set. In order to find a reason, one can start stepping towards an intended answer set. We assume that the user already trusts the program $\Pi_1 \cup \Pi_2$ from Example 8. Hence, he or she can reuse the computation $S_0, \ldots, S_6$ for $\Pi_1 \cup \Pi_2$ as starting point for a stepping session because all rules in P_{S_6} are also ground instances of rules in the extended program $\Pi_1 \cup \Pi_2 \cup \Pi_4$. Then, when the user asks for rules with active ground instances a stepping support environment would present the following rules:

```
adjacent(X,Y,X,Y+1) :- col(X), row(Y), row(Y+1).
adjacent(X,Y,X,Y-1) :- col(X), row(Y), row(Y-1).
adjacent(X,Y,X+1,Y) :- col(X), row(Y), col(X+1).
adjacent(X,Y,X-1,Y) :- col(X), row(Y), col(X-1).
reach(X,Y)   :- entrance(X,Y), not wall(X,Y).
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

```
col(1..5). row(1..5).
entrance(1,2). exit(5,4).
wall(3,3). empty(3,4).

maxCol(X) :- col(X), not col(X+1).
maxRow(Y) :- row(Y), not row(Y+1).
border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).

wall(X,Y) :- border(X,Y),
             not entrance(X,Y),
             not exit(X,Y).
{ wall(X,Y) : col(X), row(Y),
              not border(X,Y) }.
empty(X,Y) :- col(X), row(Y),
              not wall(X,Y).

adjacent(X,Y,X,Y+1) :- col(X), row(Y),
                       row(Y+1).
adjacent(X,Y,X,Y-1) :- col(X), row(Y),
                       row(Y-1).
adjacent(X,Y,X+1,Y) :- col(X), row(Y),
                       col(X+1).
adjacent(X,Y,X-1,Y) :- col(X), row(Y),
                       col(X-1).
reach(X,Y)   :- entrance(X,Y),
                not wall(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY),
                reach(X,Y),
                not wall(XX,YY).

:- empty(X,Y), not reach(X,Y).
:- empty(X,Y), empty(X+1,Y),
   empty(X,Y+1), empty(X+1,Y+1).

:- exit(X,Y), wall(X,Y).
:- wall(X,Y), wall(X+1,Y),
   wall(X,Y+1), wall(X+1,Y+1).
:- wall(X,Y), empty(X+1;X-1,Y),
   empty(X,Y+1;Y-1), col(X+1;X-1),
   row(Y+1;Y-1).
:- wall(X,Y), wall(X+1,Y+1),
   not wall(X+1,Y), not wall(X,Y+1).
:- wall(X+1,Y), wall(X,Y+1),
   not wall(X,Y), not wall(X+1,Y+1).
```

Fig. 5. Program $\Pi_1 \cup \Pi_2 \cup \Pi_5 \cup \Pi_6$ used in Example 10.

```
:- empty(X,Y), not reach(X,Y).
:- empty(X,Y), empty(X+1,Y), empty(X,X+1), empty(X+1,Y+1).
```

The attentive observer will immediately notice that two constraints are currently active. There is no reason to be deeply concerned about

```
:- empty(X,Y), not reach(X,Y).
```

being active because the rule defining the `reach/2` predicate—that can potentially deactivate the constraint—has not been considered yet. However, the constraint

```
:- empty(X,Y), empty(X+1,Y), empty(X,X+1), empty(X+1,Y+1).
```

contains only atoms of predicate `empty/2` that has already been fully evaluated. Even if `empty/2` was only partially evaluated, an active instance of the constraint could not become inactive in a subsequent computation for the sole ground that it only contains monotonic literals. When the user inspects the single ground instance

```
:- empty(1,2), empty(2,2), empty(1,2), empty(2,3).
```

of the constraint the bug becomes obvious. A less attentive observer would maybe not immediately realise that the constraint will not become inactive again. In this case, he or she would in the worst case step through all the other rules before the constraint above remains as the last rule with active instances. Then, at the latest, one comes to the same conclusion that `X+1` has to be replaced by `Y+1`. Moreover, a stepping environment could give a warning when there is a constraint instance that is guaranteed to stay active in subsequent states. This feature is not implemented in `SeaLion`. We refer to the corrected version of program Π_4 by Π_5.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

Compared to traditional software, programs in ASP are typically very succinct and often authored by a single person. Nevertheless, people are sometimes confronted with ASP code written by another person, e.g., in case of joint program development, software quality inspection, legacy code maintenance, or evaluation of student assignments in a logic-programming course. As answer-set programs can model complex problems within a few lines of code, it can be pretty puzzling to understand someone else's ASP code, even if the program is short. Here, stepping can be very helpful to get insight into how a program works that was written by another programmer, as illustrated by the following example.

Example 10

Assume that the full encoding of the maze generation encoding is composed by the programs $\Pi_2 \cup \Pi_5$ and that the constraints in Π_6, given next, has been written by another author.

```
:- exit(X,Y), wall(X,Y).
:- wall(X,Y), wall(X+1,Y), wall(X,Y+1), wall(X+1,Y+1).
:- wall(X,Y), empty(X+1;X-1,Y), empty(X,Y+1;Y-1), col(X+1;X-1),
                                                   row(Y+1;Y-1).
:- wall(X,Y), wall(X+1,Y+1), not wall(X+1,Y), not wall(X,Y+1).
:- wall(X+1,Y), wall(X,Y+1), not wall(X,Y), not wall(X+1,Y+1).
```

Note that the guess whether a cell is a wall or empty in the program $\Pi_2 \cup \Pi_5 \cup \Pi_6$ is realised by guessing for each non-border cell whether it is a wall or not and deriving that a cell is empty in case we do not know that it is a wall. Moreover, observe that facts of predicate `empty/2` may be part of a valid encoding of a maze generation problem instance, i.e., they are a potential input of the program. As a consequence, it seems plausible that the encoding could guess the existence of a wall for a cell that is already defined to be empty by a respective fact in the program input. In particular, there is no constraint that explicitly forbids that a single cell can be empty and contain a wall. The encoding would be incorrect if it would allow for answer sets with cells that are empty and a wall according to the maze generation problem specification. However, it turns out that the answer sets of the program are exactly the intended ones. Let us find out why by means of stepping.

Reconsider the problem instance depicted in Figure 1 that is encoded in the program Π_1. It requires that cell $(3,4)$ is empty. If it did not, the maze shown in Figure 7 that contains a wall at cell $(3,4)$ would be a valid solution. We start a stepping session for program $\Pi_1 \cup \Pi_2 \cup \Pi_5 \cup \Pi_6$, whose code is summarised in Figure 5, and step towards an interpretation encoding the maze of Figure 7 to see what is happening if we consider $(3,4)$ to be a wall despite the presence of fact `empty(3,4)`. We can reuse the computation C_4 obtained in Example 7 whose final state S_4 considers already the facts describing the input instance and the rules needed for deriving `border/2` atoms. As in Example 8, we continue with a step for considering the ground instance of the rule

```
{wall(X,Y) : col(X), row(Y), not border(X,Y)}.
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

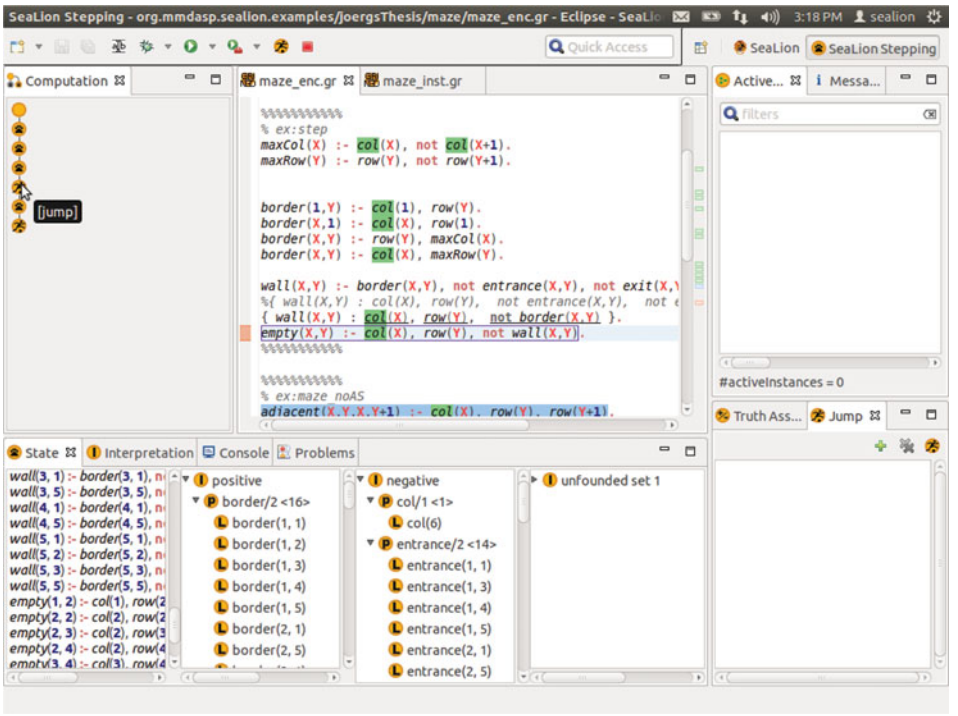

Fig. 6. In SeaLion, we can retract the computation to a previous state by clicking on the node in the computation view representing the step or jump that created the target state. The screenshot shows reverting the last two states in Example 10, where we reused a part of the computation $S_1, \ldots, S_6$ for exploring an alternative setting starting from S_4. The final state of the alternative computation presented in the example is depicted in Figure 9.

that guesses whether non-border cells are walls. This time, instead of choosing `wall(3,2)` to be true, we only add `wall(3,4)` to the atoms considered true. Then, for the resulting state S_5', both `empty(3,4)` and `wall(3,4)` are contained in $I_{S_5'}$. A visualisation of $I_{S_5'}$ if given in the centre of Figure 8. In order to derive the remaining atoms of predicates `empty/2` and `wall/2` we then jump through the rules

```
wall(X,Y) :- border(X,Y), not entrance(X,Y), not exit(X,Y).
empty(X,Y) :- col(X), row(Y), not wall(X,Y).
```

to obtain state S_6', where $I_{S_6'}$ is illustrated in the right subfigure of Figure 8.

Now, the user sees that constraint

```
:- empty(X,Y), not reach(X,Y).
```

has active instances. This comes as no surprise as the rules defining reachability between empty cells have not been considered yet. We decide to do so now and initiate a jump through the rules

```
adjacent(X,Y,X,Y+1) :- col(X), row(Y), row(Y+1).
adjacent(X,Y,X,Y-1) :- col(X), row(Y), row(Y-1).
adjacent(X,Y,X+1,Y) :- col(X), row(Y), col(X+1).
```

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

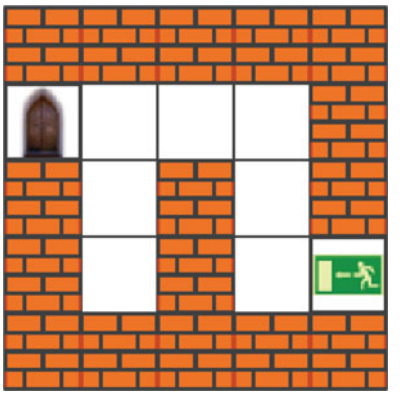

Fig. 7. A valid maze—but not a solution for the instance depicted in Figure 1 as that requires (3, 4) to be an empty cell.

```
adjacent(X,Y,X-1,Y) :- col(X), row(Y), col(X-1).
reach(X,Y)   :- entrance(X,Y), not wall(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y), not wall(XX,YY).
```

We obtain the new state S_7' and observe that under interpretation $I_{S_7'}$ the constraint still has an active instance, namely

```
:- empty(3,4), not reach(3,4).
```

Obviously, the atom `reach(3,4)` has not been derived in the computation. When inspecting the rules defining `reach/2` it becomes clear why the answer sets of the encoding are correct: the atom `reach(X,Y)` is only derived for cells that do not contain walls. Consequently, whenever there is an empty cell which was guessed to contain a wall it will be considered as not reachable from the entrance. As every empty cell has to be reachable, a respective answer-set candidate will be eliminated by an instance of the constraint

```
:- empty(X,Y), not reach(X,Y).
```

Although the encoding of the maze generation problem is correct one could consider it to be not very well designed. Conceptually, the purpose of the constraint above is forbidding empty cells to be unreachable from the entrance and not forbidding them to be walls. Moreover, if one would replace the rules

```
reach(X,Y)   :- entrance(X,Y), not wall(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y), not wall(XX,YY).
```

by

```
reach(X,Y)   :- entrance(X,Y), empty(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y), empty(XX,YY).
```

which seem to be equivalent in the terms of the problem specification, the program would not work. A more natural encoding would be to explicitly forbid empty cells to contain walls either by an explicit constraint or a modified guess where non-border cell is guessed to be either empty or contain a wall but not both.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

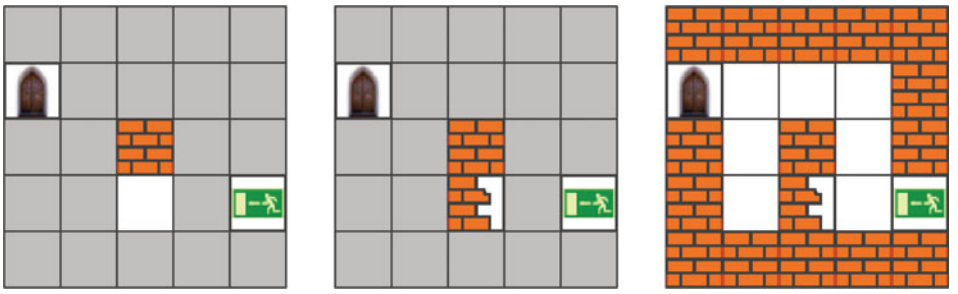

Fig. 8. The stepping session described in Example 10: Starting from the maze generation instance we step towards an interpretation encoding the wrong solution of Figure 7. After stepping through the guessing rule the resulting interpretation contains atoms `empty(3,4)` and `wall(3,4)` stating that cell (3,4) is both a wall and empty.

5 Related work

First, we describe existing approaches for debugging answer-set programs and how they are related to the method proposed in this paper.

The first work devoted to debugging of answer-set programs is a paper by Brain and De Vos (2005) in which they provide general considerations on the subject, such as the discussion of error classes in the context of ASP or implications of declarativity on debugging mentioned in the previous section. They also formulated important debugging questions in ASP, namely, why is a set of atoms subset of a specific answer set and why is a set of atoms not subset of any answer set. The authors provided pseudocode for two imperative ad-hoc algorithms for answering these questions for propositional normal answer-set programs. The algorithm addressing the first question returns answers in terms of active rules that derive atoms from the given set. The algorithm for explaining why a set of atoms is not subset of any answer set identifies different sorts of answers such as atoms with no deriving rules, inactive deriving rules, or supersets of the given set in which adding further literals would lead to some inconsistency.

The goal of the work by Pontelli *et al.* (2009) is to explain the truth values of literals with respect to a given actual answer set of a program. Explanations are provided in terms of *justifications* that are labelled graphs whose nodes are truth assignments of possibly default-negated ground atoms. The edges represent positive and negative support relations between these truth assignments such that every path ends in an assignment which is either assumed or known to hold. The authors have also introduced justifications for partial answer sets that emerge during the solving process (online justifications), being represented by three-valued interpretations. Pontelli *et al.* (2009) use sequences of three-valued interpretations (called computations) in which monotonously more atoms are considered true, respectively, false. The information carried in these interpretations corresponds to that of the second and third component of a state in a computation in our framework. The purpose of using computations in the two approaches differs, however. While computations in stepping are used for structuring debugging process in a natural way, where the choices how to proceed remains with the user, the

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

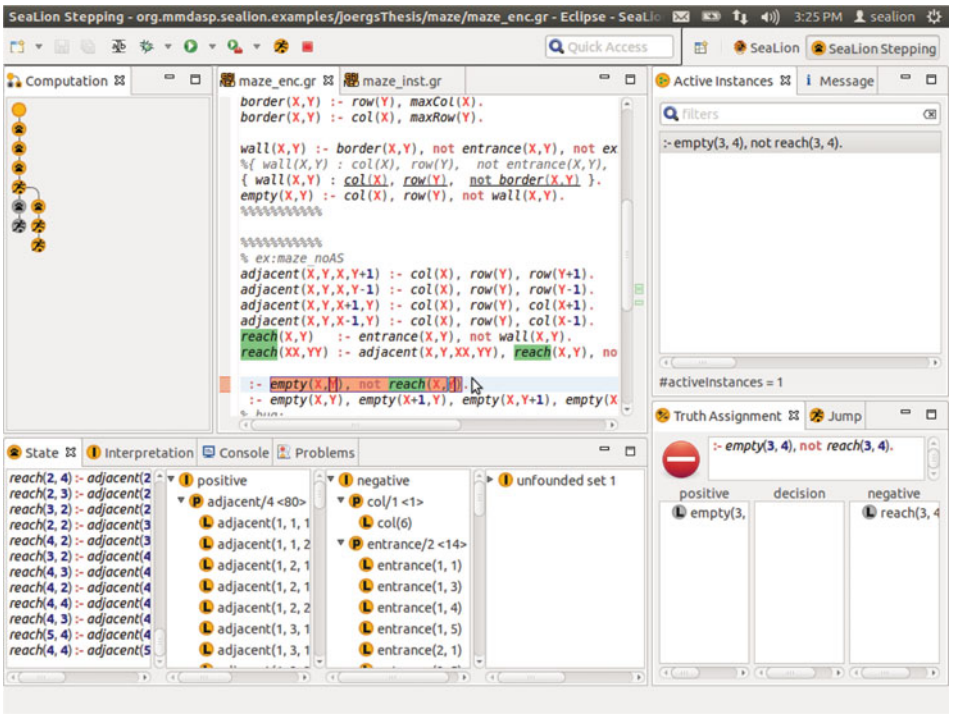

Fig. 9. The computation is stuck is state S_7' from Example 10 as only a constraint has active instances (highlighted in red in the editor). The 'No entry' symbol in the truth assignment view indicates an unsatisfied instance under the current state. Note that the tree in the computation view has an inactive branch. The current computation (nodes with yellow background) is an alternative branch for the computation in Figure 6. Clicking on the final greyed branch would set $S_1, \ldots, S_6$ to be the active computation again.

computations of Pontelli *et al.* are abstractions of the solving procedure. Their goal is to allow a solver that is compatible with their computation model to compute justifications for its intermediate results. Thus, similar to their offline versions, online justifications are non-interactive, i.e., they are computed automatically and used for post-mortem debugging. As our computation model is compatible with that for online justifications, it seems very promising to combine the two approaches in practice. While debugging information in stepping focuses on violation of rules and unfounded sets, our method leaves the reasons for an atom being true or false as implicit consequences of a user's decision. Here, online justifications could keep track of the reasons for truth values at each state of a stepping session and presented to the user during debugging on demand.

Syrjänen (2006) aimed at finding explanations why a program has no answer sets. His approach is based on finding minimal sets of constraints such that their removal yields consistency. Hereby, it is assumed that a program does not involve circular dependencies between literals through an odd number of negations which might also cause inconsistency. The author considers only a basic ASP language and hence does not take further sources of inconsistency into account, caused by program constructs of richer ASP languages, such as cardinality constraints.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

Another early approach (Brain *et al.* 2007b; Pührer 2007) is based on program rewritings using some additional control atoms, called *tags*, that allow, e.g., for switching individual rules on or off and for analysing the resulting answer sets. Debugging requests can be posed by adding further rules that can employ tags as well. That is, ASP is used itself for debugging answer-set programs. The translations needed were implemented in the command-line tool `Spock` (Brain *et al.* 2007a; Gebser *et al.* 2009) which also incorporates the translations of another approach in which also ASP is used for debugging purposes (Pührer 2007; Gebser *et al.* 2008). The technique is based on ASP meta-programming, i.e., a program over a meta-language is used to manipulate a program over an object language (in this case, both the meta-language and the object language are instances of ASP). It addresses the question why some interpretation is not an answer set of the given program. Answers are given in terms of unsatisfied rules and unfounded loops. The approach has later been extended from propositional to disjunctive logic programs with constraints, integer arithmetic, comparison predicates and strong negation (Oetsch *et al.* 2010a) and also to programs with cardinality constraints (Polleres *et al.* 2013). It has been implemented in the `Ouroboros` plugin of `SeaLion` (Frühstück *et al.* 2013). Moreover, Shchekotykhin (2015) developed a method on top of the meta-programming approaches (Gebser *et al.* 2008; Oetsch *et al.* 2010a) that poses questions to the user in order to find a desired problem diagnosis while keeping the amount of required interaction low.

In a related approach, Dodaro *et al.* (2015) use control atoms quite similar to that of the tagging approach (Brain *et al.* 2007b; Pührer 2007) to identify sets of rules that lead to inconsistency of a program under the requirement that a given set of atoms is true in some intended answer set. An implementation is provided that profits from a tight integration with the ASP solver `WASP` (Alviano *et al.* 2015). In order to reduce the possible outcomes, the debugger asks the user about the intended truth of further atoms in an interactive session.

In another paper, Li *et al.* (2015) use inductive logic programming to repair answer-set programs. The idea is that the user provides examples of desired and undesired properties of answer sets such that the debugger can semi-automatically revise a faulty program. The method requires a difference metric between logic programs in order to restrict repairs to programs that have the desired properties that minimally differ from the original program. The authors propose such a measure in terms of number of operations required for revision. These operations are rule removal and creation as well as addition or removal of individual body literals.

Caballero *et al.* (2008) developed a declarative debugging approach for datalog using a classification of error explanations similar to that of the aforementioned meta-programming technique (Gebser *et al.* 2008; Oetsch *et al.* 2010a). Their approach is tailored towards query answering and the language is restricted to stratified datalog. However, the authors provide an implementation that is based on computing a graph that reflects the execution of a query.

Wittocx *et al.* (2009) show how a calculus can be used for debugging first-order theories with inductive definitions (Denecker 2000; Denecker and Ternovska 2008) in the context of model expansion problems, i.e., problems of finding models of a

given theory that expand some given interpretation. The idea is to trace the proof of inconsistency of such an unsatisfiable model expansion problem. The authors provide a system that allows for interactively exploring the proof tree.

Besides the mentioned approaches that rely on the semantical behaviour of programs, Mikitiuk *et al.* (2007) use a translation from logic-program rules to natural language in order to detect program errors more easily. This seems to be a potentially useful feature for an IDE as well, especially for non-expert ASP programmers.

We can categorise the different methods by their setting of their debugging tasks. Here, one aspect is whether a technique works with factual or desired answer sets. Approaches that focus on actual answer sets of the program to be debugged include the algorithm by Brain and De Vos (2005) that aims at explaining the presence of atoms in an answer set. Also, justifications (Pontelli *et al.* 2009) are targeted towards explanations in a given actual answer set, with the difference that they focus on a single atom but cannot only explain their presence but also their absence. The approach by Caballero *et al.* (2008) can also be seen to target a single actual answer set. Due to their focus on actual answer sets of the debugged program, these methods cannot be applied on (erroneous) programs without any answer set. The previous meta-programming-based debugging technique (Pührer 2007; Gebser *et al.* 2008) and follow-up works (Oetsch *et al.* 2010a; Polleres *et al.* 2013) deal with a single intended but non-actual answer set of the debugged program. In the approach of Wittocx *et al.* (2009), the user can specify a class of intended semantic structures that are not preferred models of the theory at hand (corresponding to actual answer sets of the program to be debugged in ASP terminology). Syrjänen's diagnosis technique (Syrjänen 2006) is limited to the setting when a program has no answer set at all. The same holds for the work of Dodaro *et al.* (2015), however the authors demonstrate how other debugging problems can be reduced to that of inconsistency. The method requires an intended answer set but offers the means to generate that in an interactive way, building on the technique by Shchekotykhin (2015). Stepping does not require actual or intended answer sets as a prerequisite, as the user can explore the behaviour of his or her program under different interpretations that may or may not be extended to answer sets by choosing different rules instances. In the interactive setting summarised in Figure 2, where one can retract a computation to a previous state and continue stepping from there that is also implemented in `SeaLion`, a stepping session can thus be seen as an inspection across arbitrary interpretations rather than an inquiry about a concrete set of actual or non-existent answer sets. Nevertheless, if one has a concrete interpretation in mind, the user is free to focus on that. The ability to explore rule applications for partial interpretations that cannot become answer sets amounts to a form of hypothetical reasoning. A related form of this type of debugging is also available in one feature of the tagging approach (Brain *et al.* 2007b) that aims at extrapolating non-existent answer sets by switching off rules and guessing further atoms. Here, the stepping technique can be considered more focused, as the interpretation under investigation is determined by the choices of the user in stepping but is essentially arbitrary in the tagging approach if the user does not employ explicit restrictions.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

Next, we compare the ASP languages supported by different approaches. First, the language of theories with inductive definitions used in one of the debugging approaches (Wittocx *et al.* 2009) differs from the remaining approaches that are based on logic-programming rules. Many of these works deal only with the basic ASP setting of debugging ground answer-set programs, supporting only normal rules (Brain and De Vos 2005; Pontelli *et al.* 2009), disjunctive rules (Gebser *et al.* 2008) or simple choice rules (Syrjänen 2006). The work on tagging-based debugging (Brain *et al.* 2007b) sketches how to apply the approach to programs with variables by means of function symbols. The approach by Caballero *et al.* (2008) deals with non-ground normal programs which have to be stratified. Explicit support for variables is also given in an extension (Oetsch *et al.* 2010a) of the meta-programming approach for disjunctive programs. It was later extended to allow for weight constraints (Polleres *et al.* 2013) by compiling them away to normal rules. A commonality of the previous approaches is that they target ASP languages that can be considered idealised proper subsets of current solver languages. In this respect, stepping is the first debugging approach that overcomes these limitations as the use of C-programs and abstract grounding (cf. Pührer 2014) make the framework generic enough to be applied to ASP solver languages. While this does not mean that other approaches cannot be adapted to fit a solver language, it is no always immediately clear how. For our approach, instantiating our abstractions to the language constructs and the grounding method of a solver is sufficient to have a ready-to-use debugging method.

Most existing debugging approaches for ASP can be seen as declarative in the sense that a user can pose a debugging query, and receives answers in terms of different declarative definitions of the semantics of answer-set programs, e.g., in terms of active or inactive rules with respect to some interpretation. In particular, the approaches do not take the execution strategy of solvers into account. This also holds for our approach; however, stepping and online justifications (Pontelli *et al.* 2009) are exceptional as both involve a generic notion of computation that adds a procedural flavour to debugging. Nonetheless, the computation model we use can be seen as a declarative characterisation of the answer-set semantics itself as it does not apply a fix order in which to apply rules to build up an answer set.

Besides stepping, also the approaches by Wittocx *et al.* (2009) and Dodaro *et al.* (2015) as well as Shchekotykhin (2015) can be considered interactive. While in the approach of Wittocx *et al.* a fixed proof is explored interactively, the interaction in our method has influence on the direction of the computation. The other works (Dodaro *et al.* 2015; Shchekotykhin 2015) use interaction for filtering the resulting debugging information. Also in further works (Brain *et al.* 2007b; Gebser *et al.* 2008) which do not explicitly cover interleaved communication between user and system, user information can be used for filtering. The approaches mentioned in this paragraph realise declarative debugging in the sense of Shapiro (1982), where the user serves as an oracle for guiding the search for errors.

It is worth highlighting that stepping can be seen as orthogonal to the basic ideas of all the other approaches we discussed. That is, it is reasonable to have a development kit that supports stepping and other debugging methods simultaneously.

While debugging is the main focus of this paper, we also consider the computation framework for disjunctive abstract constraint programs introduced in Section 3 an interesting theoretical contribution by itself. Here, an important related work is that of Liu *et al.* (2010), who also use a notion of computation to characterise a semantics for normal C-programs. These computations are sequences of evolving interpretations. Unlike the three-valued ones used for online justifications (Pontelli *et al.* 2009), these carry only information about atoms considered true. Thus, conceptionally, they correspond to sequences $I_{S_0}, I_{S_1}, \ldots$ where $S_0, S_1, \ldots$ is a computation in our sense. The authors formulate principles for characterising different variants of computations. We will highlight differences and commonalities between the approaches along the lines of some of these properties. One main structural difference between their and our notion of computation is the granularity of steps: In the approach by Liu *et al.* it might be the case that multiple rules must be considered at once, as required by their *revision property* (R′), while in our case computation proceeds rule instance by rule instance. The purpose of property (R′) is to assure that every successive interpretation must be supported by the rules active with respect to the previous interpretation. But it also requires that every active rule in the overall program is satisfied after each step, whereas we allow rule instances that were not considered yet in the computation to be unsatisfied. For the purpose of debugging, rule-based computation granularity seems favourable as rules are our primary source code artefacts. Moreover, ignoring parts of the program that were not considered yet in a computation is essential in the stepping method, as this breaks down the amount of information that has to be considered by the user at once and allows for getting stuck and thereby detect discrepancies between his or her understanding of the program and its actual semantics. Our computations (when translated as above) meet the *persistence principle* (P′) of Liu *et al.* that ensures that a successor's interpretation is a superset of the current one. Their *convergence principle* (C′), requiring that a computation stabilises to a supported model, is not met by our computations, as we do not enforce support in general. However, when a computation has succeeded (cf. Definition 19), it meets this property. A further difference is that Liu *et al.* do not allow for non-stable computations as required by the founded persistence of reasons principle (FPr). This explains why the semantics they characterise treats non-convex atoms not in the FLP-way. Besides that, the use of non-stable computations allow us to handle disjunction. Interestingly, Liu *et al.* mention the support for disjunction in computations as an open challenging problem and suspect the necessity of a global minimality requirement on computations for this purpose. Our framework demonstrates that we can do without such a condition: As shown in Theorem 1, unfounded sets in our semantics can be computed incrementally 'on-the-fly' by considering only the rule instance added in a step as potential new external support. Finally, the *principle of persistence of reasons* (Pr′) suggests that the 'reason' for the truth value of an atom must not change in the course of a computation. Liu *et al.* identify such reasons by sets of rules that keep providing support in an ongoing computation. We have a similar principle of persistence of reasons that is however stricter as it is operates on the atom level rather than the rule level: Once a rule instance is considered in a computation in our sense, the truth value of

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

the atoms in the rule's domain is frozen, i.e., it cannot be changed or forgotten in subsequent steps. Persistence of reasons is also reflected in our definition of answer sets: The requirement $I'|_{D_A} = I|_{D_A}$ in Definition 12 that the stability of interpretation I is only spoiled by I' if the reason for $I' \models A$ is the same satisfier of C-atom A as for $I \models A$.

As argued above and in the introduction, our notion of computation is better suited for stepping than that of Liu *et al.*, yet we see potential for using the latter orthogonal to our method for debugging (for the class of programs for which the different semantics coincide). While our jumping technique allows to consider several rules, selected by the user, at once, a debugging system could provide proposals for jumping, where each proposal corresponds to a set of rules that result in a next step in the sense of Liu *et al.* Then, the system could present the atom assignments for each proposal such that the user has an alternative to choose a jump based on truth of atoms rather than rules. Moreover, this can be the basis for automated progression in a stepping session until a certain atom is assigned, analogous to watchpoints in imperative debugging. We believe that these ideas for (semi-)automated jumping deserve further investigation.

Another branch of research, that is related to our notion of computation, focuses on transition systems for analysing answer-set computation (Lierler 2011; Brochenin *et al.* 2014; Lierler and Truszczyński 2016). These works build on the ideas of a transition system for the DPLL procedure for SAT solving (Nieuwenhuis *et al.* 2006). Transition systems are graphs whose nodes represent the state of a computation whereas the edges represent possible state transitions during solving. Typically, a set of transition rules, depending on the answer-set program, determines the possible transitions. In ASP transition systems, nodes are represented by (ordered) sets of literals with annotations whether a literal is a decision literal. Moreover, there is a distinguished state *FailState* for representing when a branch of computation cannot succeed in producing an answer set. Different sets of transition rules have been proposed that correspond to different models of computations. Typical transition rules include a unit propagation rule that derives one further literal based on a rule of the program for which all other literals are defined in the previous state, a decision rule that adds one further literal (annotated as decision literal) and a transition rule for backtracking that retracts a decision literal from an inconsistent state and replace it by its negation. Existing transition systems for ASP are intended to reflect ASP solving algorithms, including failed branches of search space traversal. For instance, transition systems have been defined with transition rules for backjumping and learning as used in modern solvers (Lierler 2011). In contrast, our framework generates ideal (possibly failed) computations without backtracking. Another main difference is that all proposed transition systems have a transition rule for arbitrary assignment of decision literals whereas in our framework truth assignments are restricted to the domain of the C-rule added in the current step. Regarding supported language constructs, to the best of our knowledge, existing transition systems for ASP focus on elementary atoms, i.e., they do not cover aggregates. However, Lierler and Truszczyński (2016) also proposed transition systems for multi-logic systems including ASP. There has been work on

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

transition systems for disjunctive programs (Brochenin *et al.* 2014). These are based on integrating two sets of transition rules, one for guessing and one for checking of answer set candidates. Similarly, as in the work by Liu *et al.* (2010), states in transition systems do not keep track of ASP rules as our states do. Note that our computation framework can be turned into to a transition system for disjunctive C-programs with only two transition rules, one for propagation that is derived from the successor relation (cf. Definition 17) and another for defining the transition of unstable final states or states with remaining active rules but no successor to *FailState*.

6 Conclusion

In this paper, we introduced the stepping technique for ASP that can be used for debugging and analysis of answer-set programs. Like stepping in imperative programming, where the effects of consecutive statements are watched by the user, our stepping technique allows for monitoring the effect of rules added in a step-by-step session. In contrast to the imperative setting, stepping in our sense is interactive as a user decides in which direction to branch, by choosing which rule to consider next and which truth values its atoms should be assigned. On the one hand, this breaks a general problem of debugging in ASP, namely how to find the cause for an error, into small pieces. On the other hand, user interaction allows for focusing on interesting parts of the debugging search space from the beginning. This is in contrast to the imperative setting, where the order in which statements are considered in a debugging session is fixed. Nevertheless, also in our setting, the choice of the next rule is not entirely arbitrary, as we require the rule body to be active first. Debuggers for procedural languages often tackle the problem that many statements need to be considered before coming to an interesting step by ignoring several steps until predefined breakpoints are reached. We developed an analogous technique in our approach that we refer to as jumping which allows to consider multiple rules at once. Besides developing the technical framework for stepping, we also discussed the implementation of stepping in the `SeaLion` system and methodological aspects, thereby giving guidelines for the usage of the technique, and for setting the latter in the big picture of ASP development.

While unstable computations are often not needed, they offer great opportunities for further work. For one, the use of unfounded sets for distinguishing states in unstable computations is a natural first choice for expressing the lack of stability. Arguably, when a user arrives in a state with a non-empty unfounded set, he or she only knows that some external support has to be found for this set but there is no information which atoms of the unfounded sets are the crucial ones. It might be worthwhile to explore alternative representations for unstability such as elementary loops (Gebser *et al.* 2011) that possibly provide more pinpoint information. This would require lifting a respective notion to the full language of C-programs first.

Another issue regarding unstable computations that would deserve further attention is that in the current approach jumps can only result in stable states. Thus, unstable states in a computation can only be reached by individual steps at present.

https://doi.org/10.1017/S1471068417000217 Published online by Cambridge University Press

Here, it would be interesting to study methods and properties for computations that allow for jumping to states that are not stable.

We next discuss functionality that could be helpful for stepping which are not yet implemented in `SeaLion`. One such feature is semi-automatic stepping, i.e., the user can push a button and then the system searches for potential steps for which no further user interaction is required and applies them automatically until an answer set is reached, the computation is stuck or user interaction is required. It would also be convenient to automatically check whether the computation of a debugging session is still a computation for the debugged program after a program update. In this respect, when the computation for the old version became incompatible, a feature would be advantageous that builds up a computation for the new version that resembles the old one as much as possible. Unlike semi-automatic stepping and compatibility checks for computations that could be implemented without further studies, the latter point still requires theoretical research. Further convenient features would be functionality that highlights the truth values of atoms that cause a rule not to be active for a given substitution and methods for predicting whether a rule can become active in the future, i.e., in some continuation of the computation.

Acknowledgements

We thank the reviewers for their useful comments.

Supplementary materials

For supplementary material for this article, please visit https://doi.org/10.1017/S1471068417000217

References

Alviano, M., Dodaro, C., Leone, N. and Ricca, F. 2015. Advances in WASP. In *Proc. of Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR*, Lexington, KY, USA, September 27–30, 2015, F. Calimeri, G. Ianni and M. Truszczyński, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 40–54.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12*, 1–2, 53–87.

Brain, M. and De Vos, M. 2005. Debugging logic programs under the answer set semantics. In *Proc. of the 3rd International Workshop on Answer Set Programming* (ASP'05), Advances in Theory and Implementation, Bath, UK, September 27–29, 2005, M. De Vos and A. Provetti, Eds. CEUR Workshop Proceedings, vol. 142. CEUR-WS.org.

Brain, M., Gebser, M., Puehrer, J., Schaub, T., Tompits, H. and Woltran, S. 2007a. That is illogical Captain! The debugging support tool spock for answer-set programs – System description. In *Proc. of the 1st International Workshop on Software Engineering for Answer Set Programming* (SEA'07), Tempe, AZ, USA, May 14, 2007, M. De Vos and T. Schaub, Eds., 71–85.

Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H. and Woltran, S. 2007b. Debugging ASP programs by means of ASP. In *Proc. of the 9th International Conference*

on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), Tempe, AZ, USA, May 15–17, 2007, C. Baral, G. Brewka, and J. S. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, 31–43.

Brochenin, R., Lierler, Y. and Maratea, M. 2014. Abstract disjunctive answer set solvers. In *Proc. of the 21st European Conference on Artificial Intelligence* (ECAI'14), Prague, Czech Republic, Aug. 18–22, 2014, T. Schaub, G. Friedrich, and B. O'Sullivan, Eds. Frontiers in Artificial Intelligence and Applications, vol. 263. IOS Press, 165–170.

Busoniu, P.-A., Oetsch, J., Pührer, J., Skočovský, P. and Tompits, H. 2013. Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming 13*, 4–5, 657–673.

Caballero, R., García-Ruiz, Y. and Sáenz-Pérez, F. 2008. A theoretical framework for the declarative debugging of datalog programs. In *Revised Selected Papers of the 3rd International Workshop on Semantics in Data and Knowledge Bases* (SDKB'08), Nantes, France, March 29, 2008, K.-D. Schewe and B. Thalheim, Eds. Lecture Notes in Computer Science, vol. 4925. Springer, 143–159.

Denecker, M. 2000. Extending classical logic with inductive definitions. In *Proc. of the 1st International Conference on Computational Logic* (CL'10), London, UK, July 24–28, 2000, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer, 703–717.

Denecker, M. and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic 9*, 2.

Denecker, M., Vennekens, J., Bond, S., Gebser, M. and Truszczyński, M. 2009. The second answer set programming competition. In *Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'09), Potsdam, Germany, September 14–18, 2009, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Computer Science, vol. 5753. Springer, 637–654.

Dodaro, C., Gasteiger, P., Musitsch, B., Ricca, F. and Shchekotykhin, K. M. 2015. Interactive debugging of non-ground ASP programs. In *Proc. of Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015*, Lexington, KY, USA, September 27–30, 2015, F. Calimeri, G. Ianni and M. Truszczyński, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 279–293.

Eiter, T., Ianni, G., Schindlauer, R. and Tompits, H. 2005. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proc. of the 19th International Joint Conference on Artificial Intelligence* (IJCAI'05), Edinburgh, Scotland, UK, July 30–August 5, 2005, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 90–96.

Faber, W., Leone, N. and Pfeifer, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proc. of the 9th European Conference on Logics in Artificial Intelligence* (JELIA'04). Lecture Notes in Computer Science, vol. 3229. Springer, 200–212.

Faber, W., Pfeifer, G. and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175*, 1, 278–298.

Ferraris, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic 12*, 4, 25.

Frühstück, M., Pührer, J. and Friedrich, G. 2013. Debugging answer-set programs with Ouroboros – Extending the SeaLion plugin. In *Proc. of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'13), Corunna, Spain, September 15–19, 2013, P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 323–328.

Gebser, M., Kaminski, R., König, A. and Schaub, T. 2011. Advances in *gringo* series 3. In *Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'11), Vancouver, Canada, May 16–19, 2011, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 345–351.

Gebser, M., Kaufmann, B. and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187-188*, 52–89.

Gebser, M., Lee, J. and Lierler, Y. 2011. On elementary loops of logic programs. *Theory and Practice of Logic Programming 11*, 6, 953–988.

Gebser, M., Pührer, J., Schaub, T. and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proc. of the 23rd AAAI Conference on Artificial Intelligence* (AAAI'08), Chicago, IL, USA, July 13–17, 2008, D. Fox and C. P. Gomes, Eds. AAAI Press, 448–453.

Gebser, M., Pührer, J., Schaub, T., Tompits, H. and Woltran, S. 2009. Spock: A debugging support tool for logic programs under the answer-set semantics. In *Revised Selected Papers of the 17th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP'07) and 21st Workshop on (Constraint) Logic Programming (WLP'07), D. Seipel, M. Hanus and A. Wolf, Eds. Lecture Notes in Computer Science, vol. 5437. Springer, 247–252.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–386.

Kloimüllner, C., Oetsch, J., Pührer, J. and Tompits, H. 2013. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Revised Selected Papers of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP'11) and the 25th Workshop on Logic Programming (WLP'11), Vienna, Austria, September 28–30, 2011. Lecture Notes in Computer Science, vol. 7773. Springer, 325–344.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proc. of the 19th International Joint Conference on Artificial Intelligence* (IJCAI'05), Edinburgh, Scotland, UK, July 30–August 5, 2005, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 503–508.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

Li, T., Vos, M. D., Padget, J., Satoh, K. and Balke, T. 2015. Debugging ASP using ILP. In *Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31–September 4, 2015*, M. D. Vos, T. Eiter, Y. Lierler and F. Toni, Eds. CEUR Workshop Proceedings, vol. 1433. CEUR-WS.org.

Lierler, Y. 2011. Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming 11,* 2–3, 135–169.

Lierler, Y. and Truszczyński, M. 2016. On abstract modular inference systems and solvers. *Artificial Intelligence 236*, 65–89.

Liu, L., Pontelli, E., Son, T. C. and Truszczyński, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence 174,* 3–4, 295–315.

Marek, V. W. and Remmel, J. B. 2004. Set constraints in logic programming. In *Proc. of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'04), Fort Lauderdale, FL, USA, January 6–8, 2004, V. Lifschitz and I. Niemelä, Eds. LNCS, vol. 2923. Springer, 167–179.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczyński and D. S. Warren, Eds. Springer, 375–398.

Marek, V. W. and Truszczyński, M. 2004. Logic programs with abstract constraint atoms. In *Proc. of the 19th National Conference on Artificial Intelligence* (AAAI'04), San Jose, CA, USA, July 25–29, 2004, G. Ferguson and D. McGuinness, Eds. AAAI Press, 86–91.

Mikitiuk, A., Moseley, E. and Truszczyński, M. 2007. Towards debugging of answer-set programs in the language PSpb. In *Proc. of the 2007 International Conference on Artificial Intelligence* (ICAI'07), Volume II, Las Vegas, NV, USA, June 25–28, 2007, H. R. Arabnia, M. Q. Yang and J. Y. Yang, Eds. CSREA Press, 635–640.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3–4, 241–273.

Nieuwenhuis, R., Oliveras, A. and Tinelli, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to dpll(T). *Journal of the ACM 53,* 6, 937–977.

Oetsch, J., Pührer, J. and Tompits, H. 2010a. Catching the Ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming 10,* 4–6 (July), 513–529.

Oetsch, J., Pührer, J. and Tompits, H. 2010b. Let's break the rules: Interactive procedural-style debugging of answer-set programs. In *Proc. of the 24th Workshop on* (*Constraint*) *Logic Programming* (WLP'10), Cairo, Egypt, September 14–16, 2010, S. Abdennadher, Ed. Technical Report, Faculty of Media Engineering and Technology, German University in Cairo, 77–87.

Oetsch, J., Pührer, J. and Tompits, H. 2011. Stepping through an answer-set program. In *Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'11), Vancouver, Canada, May 16–19, 2011. Lecture Notes in Computer Science, vol. 6645. Springer, 134–147.

Oetsch, J., Pührer, J. and Tompits, H. 2012a. An FLP-style answer-set semantics for abstract-constraint programs with disjunctions. In *Technical Communications of the 28th International Conference on Logic Programming* (ICLP'12), Budapest, Hungary, A. Dovier and V. S. Costa, Eds. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 222–234.

Oetsch, J., Pührer, J. and Tompits, H. 2012b. Stepwise debugging of description-logic programs. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, E. Erdem, J. Lee, Y. Lierler and D. Pearce, Eds. Lecture Notes in Computer Science, vol. 7265. Springer, 492–508.

Oetsch, J., Pührer, J. and Tompits, H. 2013. The SeaLion has landed: An IDE for answer-set programming—Preliminary report. In *Revised Selected Papers of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP'11) and the 25th Workshop on Logic Programming (WLP'11), Vienna, Austria, September 28–30, 2011. Lecture Notes in Computer Science, vol. 7773. Springer, 305–324.

Polleres, A., Frühstück, M., Schenner, G. and Friedrich, G. 2013. Debugging non-ground ASP programs with choice rules, cardinality constraints and weight constraints. In *Proc. of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'13), Corunna, Spain, September 15–19, 2013, P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 452–464.

Pontelli, E., Son, T. C. and El-Khatib, O. 2009. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming 9,* 1, 1–56.

Pührer, J. 2007. *On debugging of propositional answer-set programs.* Master's thesis, Vienna University of Technology, Vienna, Austria.

Pührer, J. 2014. *Stepwise Debugging in Answer-Set Programming: Theoretical Foundations and Practical Realisation.* PhD thesis, Vienna University of Technology, Vienna,

Austria. Accessed 14 December 2016. URL: http://repositum.tuwien.ac.at/urn:nbn:at:at-ubtuw:1-75281 [online].

Redl, C. 2016. The DLVHEX system for knowledge representation: Recent advances (system description). *Theory and Practice of Logic Programming 16*, 5–6, 866–883.

Shapiro, E. Y. 1982. *Algorithmic Program Debugging*. PhD thesis, Yale University, New Haven, CT, USA.

Shchekotykhin, K. M. 2015. Interactive query-based debugging of ASP programs. In *Proc. of the 29th AAAI Conference on Artificial Intelligence* (AAAI'15), Austin, TX, USA, January 25–30, 2015, B. Bonet and S. Koenig, Eds. AAAI Press, 1597–1603.

Simons, P., Niemelä, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

Syrjänen, T. 2006. Debugging inconsistent answer set programs. In *Proc. of the 11th International Workshop on Non-Monotonic Reasoning* (NMR'06), Lake District, UK, May 30–June 1, 2006, J. Dix and A. Hunter, Eds. Institut für Informatik, Technische Universität Clausthal, Technical Report, 77–83.

Truszczyński, M. 2010. Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artificial Intelligence 174*, 16–17, 1285–1306.

Wittocx, J., Vlaeminck, H. and Denecker, M. 2009. Debugging for model expansion. In *Proc. of the 25th International Conference on Logic Programming* (ICLP'09), Pasadena, CA, USA, July 14–17, 2009, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 296–311.