Stepwise Debugging of Description-Logic Programs*

Johannes Oetsch, Jörg Pührer, and Hans Tompits

Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. *Description-logic programs* (or *DL-programs* for short) combine logic programs under the answer-set semantics with description logics for semantic-web reasoning. In order for a wider acceptance of the formalism among semantic-web engineers, it is vital to have adequate tools supporting the program development process. In particular, methods for debugging DL-programs are needed. In this paper, we introduce a framework for interactive stepping through a DL-program as a means for debugging which builds on recent results on stepping for standard answer-set programs. To this end, we provide a computation model for DL-programs using states based on the rules that a user considers as active in the program and the resulting intermediate interpretation. During the course of stepping, the interpretations of the subsequent states evolve towards an answer set of the overall program. Compared to the case of standard answer-set programs, we need more involved notions of states and computations in the presence of DL-atoms. In particular, if non-convex DL-atoms are involved, we have to allow for non-stable computations. Intuitively speaking, we realise this by allowing the user to assume the truth of propositional atoms which must be justified in subsequent states. To keep track of these additional atoms, we extend the well-known notion of an unfounded set for DL-programs.

1 Introduction

Description-logic programs (or *DL-programs* for short) [1] have been proposed as a powerful formalism to couple answer-set programming (ASP) [2] and description logics (DLs) [3] for semantic-web reasoning. Indeed, DL-programs realise a promising way of integrating the rules with the ontology layer in the semantic-web architecture. However, as the formalism is quite recent, it still lacks methods that support semantic-web engineers in developing DL-programs. In particular, no debugging tools for DL-programs are available.

In this paper, we introduce a stepping approach for DL-programs that allows for interactive rule-based debugging. As it is based on a sound and complete characterisation of the semantics of DL-programs, it is suited to detect all derivations from the expected to the actual semantics of a DL-program. Hence, it is not limited to detecting the source for contradictions in the case of the absence of answer sets, but it also allows for handling cases where literals are missing or are superfluous in an answer set.

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698 and by the European Commission under project IST-2009-231875 (OntoRule).

E. Erdem et al. (Eds.): Correct Reasoning, LNCS 7265, pp. 492–508, 2012.

Step-by-step execution of a program is a standard technique in procedural programming languages, where developers can debug and investigate the behaviour of their programs in an incremental way. As DL-programs have a genuine declarative semantics lacking any control flow, it is not obvious how stepping can be realised. Our approach builds on recent results on stepping of standard logic programs under the answer-set semantics [4]. Similar to that approach, we introduce a computation model for DL-programs that is based on states which represent the ground DL-rules that a user considers as active in the program and the resulting intermediate interpretation. The approach for answer-set programs was based on a simple computation model in which, at each intermediate state, the interpretation that is induced by the considered rules is guaranteed to be an answer set of these rules. As this is in general not possible for DL-programs, we have to extend the previous notions of a state and of the successor relation determining how to step from one state to another to deal with the presence of DL-atoms. For achieving this goal, we define the notions of an unfounded set and of the external support for DL-programs that allow us to keep track of literals that still need to be justified by a defining rule at a later step in a computation.

Our stepping approach is *interactive* and *incremental*, letting the semantic-web engineer choose which rules are added at each step. In our framework, states may serve as *breakpoints* from which stepping can be started. We discuss how the user can generate breakpoints that can be used to jump directly to interesting situations. We also show how ground rules that are subsequently considered active can be quickly obtained from the non-ground source code using filtering techniques. Due to the interactive nature of our proposed approach, the search for bugs can easily be guided by the intuitions of a developer about which part of the DL-program is likely to be the source of an error.

2 Preliminaries

Intuitively, a DL-program is a combination of a standard DL knowledge base and a logic program augmented with dedicated atoms realising the coupling. We first recall syntax and semantics of DLs and then introduce DL-programs based on that.

2.1 Description Logics

As our stepping approach is to a large extent independent of a specific DL, we only provide background for the basic description logic $\mathcal{ALC}$ and refer the interested reader to the literature [3] for more information on language features beyond $\mathcal{ALC}$.

By a *DL-signature* we understand a triple $\Sigma = \langle \mathcal{C}, \mathcal{R}, \mathcal{I} \rangle$, where $\mathcal{C}$, $\mathcal{R}$, and $\mathcal{I}$ are pairwise disjoint (denumerable) sets of *atomic concepts*, *role names*, and *individual names*, respectively. *Concepts* are inductively defined thus: (i) each atomic concept $A \in \mathcal{C}$, $\top$ (the *universal concept*), and $\bot$ (the *empty concept*) are concepts; (ii) if C and D are concepts and $R \in \mathcal{R}$ is a role name, then $C \sqcap D$ (the *intersection of C and D*), $C \sqcup D$ (the *union of C and D*), $\neg C$ (the *negation of C*), $\exists R.C$ (the *existential restriction of C by R*), and $\forall R.C$ (the *universal restriction of C by R*) are also concepts.

A (*DL*) *knowledge base* $\Phi = \langle \mathcal{T}, \mathcal{A} \rangle$, also referred to as a (*DL*) *ontology*, consists of a *TBox* $\mathcal{T}$, which constitutes the terminological part of the knowledge base, and its

assertional part $\mathcal{A}$, called *ABox*, which consists of assertions about actual individuals. A TBox is a finite set of *concept inclusion axioms* of the form $C \sqsubseteq D$ (expressing that the extension of C is a subset of the extension of D) or $C \equiv D$ (meaning that both $C \sqsubseteq D$ and $D \sqsubseteq C$ holds) with C and D being concepts. An ABox is a finite set of *concept assertions* of the form $C(a)$ and *role assertions* of the form $R(a, b)$, where $a, b \in \mathcal{I}$ are individual names, C is a concept, and $R \in \mathcal{R}$ is a role name.

An *interpretation* $I = \langle \Delta^I, \cdot^I \rangle$ consists of a nonempty *domain* Δ^I and a mapping $\cdot^I$ that assigns to each atomic concept $C \in \mathcal{C}$ a subset of Δ^I, to each individual $o \in \mathcal{I}$ an element of Δ^I, and to each role $R \in \mathcal{R}$ a subset of $\Delta^I \times \Delta^I$. The mapping $\cdot^I$ is inductively defined as follows, where C and D are concepts and $R \in \mathcal{R}$ is a role name:

- $\top^I = \Delta^I$ and $\bot^I = \emptyset$;
- $(C \sqcap D)^I = C^I \cap D^I$;
- $(C \sqcup D)^I = C^I \cup D^I$;
- $(\neg C)^I = \Delta^I \setminus C^I$;
- $(\forall R.C)^I = \{x \in \Delta^I \mid \forall y \colon \langle x, y \rangle \in R^I \rightarrow y \in C^I\}$; and
- $(\exists R.C)^I = \{x \in \Delta^I \mid \exists y \colon \langle x, y \rangle \in R^I \wedge y \in C^I\}$.

The *satisfaction relation* $\models$ between an interpretation I and a concept inclusion axiom $C \sqsubseteq D$, a concept assertion $C(a)$, or a role assertion $R(a, b)$ is defined as follows: (i) $I \models C \sqsubseteq D$ iff $C^I \subseteq D^I$; (ii) $I \models C(a)$ iff $a^I \in C^I$; and (iii) $I \models R(a, b)$ iff $\langle a^I, b^I \rangle \in R^I$. An interpretation I is a *model* of a TBox $\mathcal{T}$, symbolically $I \models \mathcal{T}$, iff $I \models t$ for all $t \in \mathcal{T}$. Moreover, I is a model of an ABox $\mathcal{A}$, symbolically $I \models \mathcal{A}$, iff $I \models a$ for all $a \in \mathcal{A}$. Finally, I is a model of an $\mathcal{ALC}$ knowledge base $\Phi = \langle \mathcal{T}, \mathcal{A} \rangle$ iff $I \models \mathcal{T}$ and $I \models \mathcal{A}$. An axiom or assertion F is a *logical consequence* of Φ, denoted by $\Phi \models F$, iff every model of Φ satisfies F.

2.2 DL-Programs

In the following, we briefly summarise syntax and semantics of DL-programs.

A *signature* $\Sigma = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{I} \rangle$ for DL-programs consists of pairwise disjoint (denumerable) sets $\mathcal{C}$, $\mathcal{R}$, $\mathcal{P}$, and $\mathcal{I}$, where $\langle \mathcal{C}, \mathcal{R}, \mathcal{I} \rangle$ is a DL-signature and $\mathcal{P}$ is a set of predicate symbols. By a *term* we understand an individual name from $\mathcal{I}$ or a variable. A (*classical*) *literal* is an atom a or its *strong negation* $\sim a$. For a literal l, we define $\mathrm{Lit}_l = \{l\}$. A *query*, $Q(\mathbf{t})$, is either (i) a concept inclusion axiom F or its negation $\neg F$, (ii) an expression of form $C(t)$ or $\neg C(t)$, where C is a concept and t is a *term*, or (iii) an expression of form $R(t_1, t_2)$ or $\neg R(t_1, t_2)$, where R is a role and t_1, t_2 are terms.

Informally, a DL-program over $\Sigma = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{I} \rangle$ consists of an ontology Φ over $\Sigma_o = \langle \mathcal{C}, \mathcal{R}, \mathcal{I} \rangle$ and a normal logic program Π over Σ possibly containing queries to Φ. In formal terms, a *DL-atom* $a(\mathbf{t})$ *over* Σ is defined as an expression of form

$$\mathrm{DL}[S_1 \; op_1 \; p_1, \ldots, S_m \; op_m \; p_m; \, Q](\mathbf{t}) \,, \qquad m \geq 0 \,, \tag{1}$$

where each S_i is either a concept from $\mathcal{C}$ or a role predicate from $\mathcal{R}$, $op_i \in \{\uplus, \cup, \cap\}$, p_i is a unary or binary predicate symbol from $\mathcal{P}$, respectively, and $Q(\mathbf{t})$ is a query. We call $S_1 \; op_1 \; p_1, \ldots, S_m \; op_m \; p_m$ the *input signature* and $p_1, \ldots, p_m$ the *input predicate*

symbols of $a(\mathbf{t})$. Moreover, literals over input predicate symbols are *input literals*. We denote the set of input literals of a DL-atom A by Lit_A. Intuitively, $\uplus$ (resp., $\cup$) increases S_i (resp., $\neg S_i$) by the extension of p_i, while $\cap$ constrains S_i to p_i. A *DL-rule* r *over* Σ has the form

$$a \leftarrow b_1, \ldots, b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m\,, \qquad m \geq k \geq 0\,, \tag{2}$$

where a is a literal and any $b_1, \ldots, b_m$ is a literal or a DL-atom. We call $B(r) = \{b_1, \ldots, b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m\}$ the *body* of r and $H(r) = a$ the *head* of r. Moreover, we distinguish between the *positive body* $B^+(r) = \{b_1, \ldots, b_k\}$ and the *negative body* $B^-(r) = \{b_{k+1}, \ldots, b_m\}$ of r. By Lit_r we denote the set $\{a\} \cup \bigcup_{1 \leq i \leq m} \mathrm{Lit}_{b_i}$. A DL-rule with $B(r) = \emptyset$ is called a *fact*. DL-rules without a head are also allowed and are called *constraints*. These are used to filter out every answer-set candidate that satisfies their bodies. Finally, a *description-logic program*, or a *DL-program*, *over* $\Sigma = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{I} \rangle$ is a pair $\mathcal{KB} = \langle \Phi, \Pi \rangle$ consisting of a DL ontology Φ over $\Sigma_o = \langle \mathcal{C}, \mathcal{R}, \mathcal{I} \rangle$ and a finite set of DL-rules Π over Σ.

For defining the semantics of DL-programs, let in what follows $\mathcal{KB} = \langle \Phi, \Pi \rangle$ be a DL-program over $\Sigma = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{I} \rangle$, where $\Phi = \langle \mathcal{T}, \mathcal{A} \rangle$, and $\Sigma_{ASP} = \langle \mathcal{P}, \mathcal{I} \rangle$. By $gr(\Pi)$ we denote the *grounding* of Π with respect to $\mathcal{I}$, i.e., the set of all ground rules originating from DL-rules in Π by uniformly replacing, per DL-rule, all variables by each possible combination of constants in $\mathcal{I}$.

An *interpretation* I *over* Σ_{ASP} is a consistent subset of literals over Σ_{ASP}. We say that I *satisfies* a literal l under Φ, denoted by $I \models^{\Phi} l$, iff $l \in I$. Furthermore, I satisfies a ground DL-atom $a = DL[S_1 op_1\ p_1, \ldots, S_m op_m p_m; Q](\mathbf{c})$ under Φ, denoted by $I \models^{\Phi} a$, if $\langle \mathcal{T}, \mathcal{A} \cup \tau^I(a) \rangle \models Q(\mathbf{c})$, where $\tau^I(a) = \bigcup_{i=1}^{m} A_i(I)$ is the extension of a under I and

- $A_i(I) = \{S_i(\mathbf{t}) \mid p_i(\mathbf{t}) \in I\}$, for $op_i = \uplus$;
- $A_i(I) = \{\neg S_i(\mathbf{t}) \mid p_i(\mathbf{t}) \in I\}$, for $op_i = \cup$; and
- $A_i(I) = \{\neg S_i(\mathbf{t}) \mid p_i(\mathbf{t}) \notin I\}$, for $op_i = \cap$.

An interpretation I satisfies $\mathit{not}\ b$ under Φ, where b is a literal or a DL-atom, symbolically $I \models^{\Phi} \mathit{not}\ b$, if $I \not\models^{\Phi} b$. Let S be a set of literals and DL-atoms, each of which possibly default negated. Then, I satisfies S under Φ, symbolically $I \models^{\Phi} S$, if $I \models^{\Phi} l$ for each $l \in S$. A DL-rule r is *active under* I *and* Φ iff $I \models^{\Phi} B(r)$. For a ground DL-rule r under Φ not being a constraint, I satisfies r, symbolically $I \models^{\Phi} r$, if $I \models^{\Phi} B(r)$ implies $I \models^{\Phi} H(r)$. Furthermore, I is a model of a DL-program $\mathcal{KB} = \langle \Phi, \Pi \rangle$, denoted by $I \models \mathcal{KB}$, if $I \models^{\Phi} r$ for all $r \in gr(\Pi)$.

We base the semantics of DL-programs on a reduct construction introduced by Faber, Leone, and Pfeifer [5], which we sometimes refer to as the *FLP-semantics*. We consider this semantics rather than the weak or strong semantics of Eiter et al. [1] because it is the one implemented in DLVHEX[1], the state-of-the-art solver for HEX-programs [6] which generalise DL-programs.

Definition 1. *Let* $\Sigma = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{I} \rangle$ *be a signature for DL-programs,* Φ *a DL ontology over* $\langle \mathcal{C}, \mathcal{R}, \mathcal{I} \rangle$*,* Π *a set of ground DL-rules over* Σ*, and* I *an interpretation over*

[1] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

$\Sigma_{ASP} = \langle \mathcal{P}, \mathcal{I} \rangle$. *Then, the* FLP-reduct of Π under Φ relative to I *is the set*

$$\Pi_{\Phi}^{I} = \{r \in \Pi \mid I \models^{\Phi} B(r)\}.$$

We first define answer sets of sets of DL-rules with respect to a DL knowledge base.

Definition 2. *Let $\Sigma = \langle \mathcal{C}, \mathcal{R}, \mathcal{P}, \mathcal{I} \rangle$ be a signature for DL-programs, Φ a DL ontology over $\langle \mathcal{C}, \mathcal{R}, \mathcal{I} \rangle$, Π a set of DL-rules over Σ, and I an interpretation over $\Sigma_{ASP} = \langle \mathcal{P}, \mathcal{I} \rangle$. Then, I is an* answer set of Π with respect to Φ *if it is a minimal model of $gr(\Pi)_{\Phi}^{I}$. The set of all answer sets of Π with respect to Φ is denoted by* $\mathrm{AS}(\Pi)^{\Phi}$.

Based on that, we define answer sets for DL-programs as follows.

Definition 3. *Let $\mathcal{KB} = \langle \Phi, \Pi \rangle$ be a DL-program. An interpretation I is an* answer set of $\mathcal{KB}$ *if it is is an answer set of Π with respect to Φ. The set of all answer sets of $\mathcal{KB}$ is denoted by* $\mathrm{AS}(\mathcal{KB})$.

In the absence of DL-atoms in a DL-program $\mathcal{KB} = \langle \Phi, \Pi \rangle$, Π corresponds to a non-disjunctive extended logic program whose answer sets as defined by Gelfond and Lifschitz [7,8] coincide with the answer sets of $\mathcal{KB}$. If $\mathcal{KB}$ contains neither DL-atoms nor strong negation, Π is a normal logic program and the answer sets of $\mathcal{KB}$ are the stable models of Π as defined earlier by Gelfond and Lifschitz [9]. Also note that the semantics we use coincides with the strong answer-set semantics when all DL-atoms are *monotonic* [6]:

Definition 4. *For a DL-program $\mathcal{KB} = \langle \Phi, \Pi \rangle$, a ground DL-atom a is* monotonic relative to $\mathcal{KB}$ *if for all interpretations I and J with $I \subseteq J$, $I \models^{\Phi} a$ implies $J \models^{\Phi} a$.*

Moreover, we need the related notion of *convex* DL-atoms:

Definition 5. *For a DL-program $\mathcal{KB} = \langle \Phi, \Pi \rangle$, a ground DL-atom a is* convex relative to $\mathcal{KB}$ *if for all interpretations I, J, and K, if $I \subset J \subset K$, then, whenever $I \models^{\Phi} a$ and $K \models^{\Phi} a$ jointly hold, then also $J \models^{\Phi} a$ holds.*

3 A Stepping Framework for DL-Programs

In this section, we introduce our framework for stepping through DL-programs. As noted in the introduction, our approach builds on ideas of previous work on stepping for normal logic programs under the answer-set semantics [4]. To illustrate the basic idea of stepping, we first briefly discuss the intuitions of the previous stepping approach.

The general idea is to first take a part of a program and an answer set of this part. Then, step by step, rules are added by the user such that, at every step, the literal derived by the new rule is added to the interpretation which remains to be an answer set of the evolving program part. Hereby, the user only adds rules he or she thinks are active in the final 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 an expected or an unintended actual answer set. The individual steps of a computation, referred to as *states* of the program, are represented by a set of ground rules which the user considers as active along with an interpretation that constitutes a partial answer set of the program.

Example 1. Let Π be the normal logic program consisting of the rules

$$r_1 : p_1(c) \leftarrow not\ p_2(c), \quad r_2 : p_2(c) \leftarrow not\ p_1(c), \quad \text{and} \quad r_3 : p_3(c) \leftarrow p_1(c).$$

Following the intuitions above, we express states by pairs $\langle \Pi', I' \rangle$, where Π' are the rules considered active and I' is the interpretation derived by those rules. First, we consider no rule to be active, and hence no atom is derived—the corresponding state is $\langle \emptyset, \emptyset \rangle$. Under the current interpretation $\emptyset$, two unconsidered rules are active: r_1 and r_2. We choose to consider r_1 first and arrive at state $\langle \{r_1\}, \{p_1(c)\} \rangle$, as $p_1(c)$ is derived when r_1 is assumed to be active. At this point, only r_3 is active under $\{p_1(c)\}$. Hence, we finally reach state $\langle \{r_1, r_3\}, \{p_1(c), p_3(c)\} \rangle$ whose interpretation $\{p_1(c), p_3(c)\}$ is an answer set of Π. □

Note that in the example, and generally in the case of normal logic programs, stepping can be done in such a way that every intermediate state in a computation is *stable*. This means that the atoms I' derived by the currently considered set Π' of rules form an answer set of Π'. This is in general not possible for DL-programs in the presence of non-convex DL-atoms. Thus, the previous method for standard logic programs cannot be applied straightforwardly when DL-atoms are involved, as illustrated next.

Example 2.
Consider the DL-program $\mathcal{KB} = \langle \Phi, \Pi \rangle$ where $\Phi = \langle \{A \sqcap B \sqsubseteq Q, \neg C \sqcap \neg D \sqsubseteq Q\}, \emptyset \rangle$ and Π consists of the DL-rules

$$\begin{aligned} &r_1 : p_1(c) \leftarrow \mathrm{DL}[A \uplus p_1, B \uplus p_2, C \cap p_3, D \cap p_2;\ Q](c), \\ &r_2 : p_2(c) \leftarrow p_1(c), \quad \text{and} \\ &r_3 : p_1(c) \leftarrow p_2(c), \end{aligned}$$

having unique answer set $\{p_1(c), p_2(c)\}$. The DL-atom involved is non-convex relative to Φ as it is true under $\emptyset$ and $\{p_1(c), p_2(c)\}$ but not under $\{p_1(c)\}$ or $\{p_2(c)\}$. Now, assume we want to start stepping from the empty interpretation. At the beginning, only r_1 is applicable under $\emptyset$, which derives $p_1(c)$. We arrive at a state which is not stable as $\{p_1(c)\}$ is not an answer set of $\langle \Phi, \{r_1\} \rangle$. Next, we can only choose r_2 as next DL-rule to be considered active. The two active DL-rules already derive the answer set $\{p_1(c), p_2(c)\}$ of Π, but $\{p_1(c), p_2(c)\}$ is no answer set of $\langle \Phi, \{r_1, r_2\} \rangle$ because $\{p_2(c)\}$ is a smaller model of $\langle \Phi, \{r_1, r_2\} \rangle$. However, the computation becomes stable again when we add r_3. □

The example shows that a stepping approach for DL-programs under the FLP-semantics must allow for non-stable computations. Intuitively, we realise this by allowing the user to assume the truth of propositional literals which must be justified in subsequent states. Technically, we use the theory of unfounded sets for guaranteeing stability at a later point in the computation. To this end, we next introduce the notions of external support and unfounded sets for DL-programs.

3.1 External Support and Unfounded Sets for DL-Programs

Intuitively, each set of literals in an answer set must be "supported" by an active DL-rule deriving one of the literals in a non-cyclic way, i.e., the reason for the DL-rule to be active does not depend on the literal it derives. We call such DL-rules *external supports*.

Definition 6. *Let r be a ground DL-rule, Φ a DL knowledge base, I an interpretation, and X a set of literals. Then, r is an* external support for X with respect to I and Φ *if* (*i*) $I \models^{\Phi} B(r)$, (*ii*) $I \setminus X \models^{\Phi} B(r)$, *and* (*iii*) $H(r) \in (X \cap I)$.

Next, we show how answer sets can be characterised in terms of external supports.

Theorem 1. *Let $\mathcal{KB} = \langle \Phi, \Pi \rangle$ be a DL-program and I an interpretation. Then, I is an answer set of $\mathcal{KB}$ iff $I \models^{\Phi} gr(\Pi)$ and every X with $\emptyset \subset X \subseteq I$ has an external support $r \in gr(\Pi)$ with respect to I and Φ.*

We express the absence of an external support in an interpretation by adapting the concept of an *unfounded set* [10,11] to DL-programs.

Definition 7. *Let X be a set of literals, Π a set of DL-rules, Φ a DL knowledge base, and I an interpretation. Then, X is* unfounded in Π with respect to I and Φ *if there is no DL-rule $r \in \Pi$ that is an external support for X with respect to I and Φ.*

Note that $\emptyset$ is an unfounded set independent of which DL-rules, interpretations, or DL knowledge base is chosen. Theorem 1 immediately implies the following result:

Corollary 1. *Let $\mathcal{KB} = \langle \Phi, \Pi \rangle$ be a DL-program and I an interpretation. Then, I is an answer set of $\mathcal{KB}$ iff $I \models^{\Phi} gr(\Pi)$ and there is no X with $\emptyset \subset X \subseteq I$ that is unfounded in Π with respect to I.*

3.2 States and Computations

Our stepping framework is based on sequences of states, reassembling computations, in which an increasing number of ground DL-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 8. *A* state structure *is a tuple $\langle \Pi, I, I^-, \Upsilon \rangle$, where Π is a set of ground DL-rules, I and I^- are disjoint interpretations, and Υ is a set of sets of literals.*

Given a DL knowledge base Φ, a state structure $\langle \Pi, I, I^-, \Upsilon \rangle$ is a state with respect to Φ *if* (*i*) $I \models^{\Phi} B(r) \cup H(r)$ *for every $r \in \Pi$,* (*ii*) $\mathrm{Lit}_r \subseteq I \cup I^-$ *for every $r \in \Pi$, and* (*iii*) $\Upsilon = \{X \subseteq I \mid X \text{ is unfounded in } \Pi \text{ with respect to } I \text{ and } \Phi\}$.

Now we are ready to formally state what we understand by the stability of a state.

Definition 9. *A state $\langle \Pi, I, I^-, \Upsilon \rangle$ with respect to an DL knowledge base Φ is* Φ-stable *if $I \in \mathrm{AS}(\Pi)^{\Phi}$.*

Note that a state is Φ-stable exactly when $\Upsilon = \{\emptyset\}$.

In what follows, we show how we can proceed forward in a computation, i.e., which states might follow a given state. This is expressed in the successor relation defined next.

Definition 10. *For a state $S = \langle \Pi, I, I^-, \Upsilon \rangle$ with respect to DL knowledge base Φ and a state structure $S' = \langle \Pi', I', I'^-, \Upsilon' \rangle$, S' is a* Φ-successor of S *if there is a DL-rule $r \in \Pi' \setminus \Pi$ and sets $\Delta, \Delta^- \subseteq \mathrm{Lit}_r$ such that* (*i*) $\Pi' = \Pi \cup \{r\}$, (*ii*) $I' = I \cup \Delta$, $I'^- = I^- \cup \Delta^-$, *and* $(I \cup I^-) \cap (\Delta \cup \Delta^-) = \emptyset$, (*iii*) $\mathrm{Lit}_r \subseteq (I' \cup I'^-)$, (*iv*) $I \models^{\Phi} B(r)$, (*v*) $I' \models^{\Phi} B(r) \cup H(r)$, *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' and Φ.*

Condition (i) ensures that a successor state considers exactly one DL-rule more to be active. Conditions (ii) and (iii) express that the interpretations I and I^- are extended by the so far unconsidered literals in Δ and Δ^- appearing in the new DL-rule r. Note that from S' being a state structure we get that Δ and Δ^- are distinct. A requirement for considering r as next DL-rule is that it is active under the current interpretation I, expressed by Condition (iv). Moreover, r 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 DL-rule.

Here, it is interesting that only extended previous unfounded sets can be unfounded sets in the extended program Π' and that r is the only rule which could provide external support for them in Π' with respect to the new interpretation I' and Φ, as seen next.

Theorem 2. *Let $S = \langle \Pi, I, I^-, \Upsilon \rangle$ be a state with respect to DL knowledge base Φ and $S' = \langle \Pi \cup \{r\}, I', I'^-, \Upsilon' \rangle$ a Φ-successor of S, where $\Delta = I' \setminus I$. Moreover, let X' be a set of literals with $\emptyset \subset X' \subseteq I'$. Then, the following statements are equivalent:*
(i) X' is unfounded in $\Pi \cup \{r\}$ with respect to I' and Φ.
(ii) $X' = \Delta' \cup X$, where $\Delta' \subseteq \Delta$, $X \in \Upsilon$, and r is not an external support for X' with respect to I' and Φ.

The result shows that determining the unfounded sets in a computation after adding a further DL-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 with respect to DL knowledge base Φ and S' a Φ-successor of S. Then, S' is a state with respect to Φ.*

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

Definition 11. *Let Φ be a DL ontology. A* Φ-computation *is a sequence $C = S_0, \ldots, S_n$ of states with respect to Φ such that S_{i+1} is a Φ-successor of S_i, for all $0 \leq i < n$.*

The following result guarantees the soundness of our stepping framework.

Theorem 3. *Let $\mathcal{KB} = \langle \Phi, \Pi \rangle$ be a DL-program and $C = S_0, \ldots, S_n$ a Φ-computation such that $S_n = \langle gr(\Pi)^I_\Phi, I, I^-, \{\emptyset\} \rangle$. Then, I is an answer set of $\mathcal{KB}$.*

The computation model is also complete in the following sense:

Theorem 4. *Let $S_0 = \langle \Pi, I, I^-, \Upsilon \rangle$ be a state with respect to Φ, $\mathcal{KB} = \langle \Phi, \Pi' \rangle$ a DL-program with $\Pi \subseteq gr(\Pi')$, and I' an answer set of $\mathcal{KB}$ with $I \subseteq I'$ and $I' \cap I^- = \emptyset$. Then, there is a Φ-computation $S_0, \ldots, S_n$ such that $S_n = \langle gr(\Pi)^{I'}_\Phi, I', I'^-, \{\emptyset\} \rangle$.*

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

Corollary 3. *Let $\mathcal{KB} = \langle \Phi, \Pi \rangle$ be a DL-program and $I \in \mathrm{AS}(\mathcal{KB})$. Then, there is a Φ-computation $S_0, \ldots, S_n$ with $S_0 = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$ and $S_n = \langle gr(\Pi)^{I'}_\Phi, I', I'^-, \{\emptyset\} \rangle$.*

Example 3. Consider $\mathcal{KB} = \langle \Phi, \Pi \rangle$ from Example 2. Then, the sequence

$$\langle \emptyset, \emptyset, \emptyset, \{\emptyset\}\rangle,\ \langle \{r_1\}, \{p_1(c), p_2(c)\}, \emptyset, \{\emptyset, \{p_1(c)\}, \{p_2(c)\}\}\rangle,$$
$$\langle \{r_1, r_2\}, \{p_1(c), p_2(c)\}, \emptyset, \{\emptyset, \{p_1(c)\}\}\rangle,\ \langle \{r_1, r_2, r_3\}, \{p_1(c), p_2(c)\}, \emptyset, \{\emptyset\}\rangle$$

is a Φ-computation, constituting a derivation for the answer set $\{p_1(c), p_2(c)\}$ of $\mathcal{KB}$. The first and the last state in this computation are Φ-stable whereas the other two are not, as indicated by the presence of non-empty unfounded sets. □

Indeed, the DL-program of Example 2 can be seen as an unlikely worst-case scenario in which computations are required to be unstable. In fact, whenever a DL-program does not involve recursion through DL-atoms or contains only convex DL-atoms then computations with stable states only are sufficient to compute all answer sets. That is, in such a setting, it is sufficient to add at most a single classical literal to the emerging interpretation in every subsequent state. Note that DL-atoms that do not involve the rarely used $\uplus$-operator are always convex.

4 Applying Stepping

In this section, we outline how the stepping framework can be applied in practice. As it allows for stepwise constructing interpretations following a user's intuition on which DL-rule instances to become active next, one may also reach states where there is no answer set of the overall DL-program extending the state's interpretation under which the DL-rules considered active are indeed active. Then, every continuation would reach a sate where adding a further DL-rule would require to add literals that are inconsistent with previously chosen active rules. The possibility of such dead-ends is intentional, as reaching such a point—which can be automatically detected—indicates that and why the DL-program's semantics differs from the semantics expected by the user.

In our approach, the user always has two options how to proceed: (i) (re-)initialise stepping and start a computation with a new state as breakpoint, or (ii) extend the current computation by adding a further active DL-rule. We first describe the technical aspects of how to obtain a breakpoint and how ground DL-rule instances can be chosen. Then, we discuss how stepping can be applied for debugging.

As an example, we consider a situation where a car model should undergo multiple crash tests under different conditions. We assume that the manufacturer has an ontology Φ_{ex} about possible test conditions regarding the car which also contains information on which conditions cannot hold at the same time. The knowledge base is given as follows:

$$\begin{aligned}\Phi_{ex} = \{\{&\mathit{TestableCond} \equiv \mathit{Testable} \sqcap \mathit{Cond}\},\\ \{&\mathit{Cond}(\mathit{engine\_running}),\ \mathit{Cond}(\mathit{engine\_off}),\ \mathit{Cond}(\mathit{battery\_on}),\\ &\mathit{Cond}(\mathit{battery\_off}),\ \mathit{Cond}(\mathit{front\_seat\_adult}),\\ &\mathit{Cond}(\mathit{front\_seat\_child}),\ \mathit{Cond}(\mathit{front\_seat\_empty}),\\ &\mathit{Cond}(\mathit{extreme\_temperature}),\ \mathit{Cond}(\mathit{low\_temperature}),\\ &\mathit{Incompatible}(\mathit{engine\_running}, \mathit{engine\_off}),\\ &\mathit{Incompatible}(\mathit{battery\_on}, \mathit{battery\_off}),\\ &\mathit{Incompatible}(\mathit{engine\_running}, \mathit{battery\_off}),\\ &\mathit{Incompatible}(\mathit{front\_seat\_adult}, \mathit{front\_seat\_child}),\\ &\mathit{Incompatible}(\mathit{front\_seat\_adult}, \mathit{front\_seat\_empty}),\end{aligned}$$

$$Incompatible(front\_seat\_child, front\_seat\_empty),$$
$$Incompatible(extreme\_temperature, low\_temperature)\}\}.$$

The axiom states that a testable condition is both in the extension of $Testable$ and $Cond$, where $Testable$ is a concept for which the ontology does not assert any individuals. The remaining assertions list the conditions that can be set for testing as well as specify which conditions are incompatible in the sense that they cannot hold at the same time.

The task now is to write DL-rules such that the resulting DL-program creates a fixed number n of different test configurations in which every compatible pair of testable conditions is tested at least once in some configuration. Moreover, it should also be expressed by the DL-rules which conditions are considered testable. However, only those conditions which are also known in the ontology should be used for testing. Assume the set Π_{ex}, comprising the following DL-rules, realises this task:

$$\begin{aligned}
r_1 &: testNo(1..n) \leftarrow,\\
r_2 &: testable(engine_running) \leftarrow,\\
r_3 &: testable(engine_off) \leftarrow,\\
r_4 &: testable(battery_on) \leftarrow,\\
r_5 &: testable(battery_off) \leftarrow,\\
r_6 &: testable(front_seat_adult) \leftarrow,\\
r_7 &: testable(front_seat_child) \leftarrow,\\
r_8 &: testable(front_seat_empty) \leftarrow,\\
r_9 &: testable(roof_opened) \leftarrow,\\
r_{10} &: testable(roof_closed) \leftarrow,\\
r_{11} &: testcond(M) \leftarrow \mathrm{DL}[Testable \uplus testable;\ TestableCond](M),\\
r_{12} &: incompatible(X, Y) \leftarrow \mathrm{DL}[;\ Incompatible](X, Y),\\
r_{13} &: incompatible(X, Y) \leftarrow incompatible(Y, X),\\
r_{14} &: test(T, S) \leftarrow testcond(S), testNo(T), not\ {\sim}test(T, S),\\
r_{15} &: {\sim}test(T, S) \leftarrow testcond(S), testNo(T), not\ test(T, S),\\
r_{16} &: combination(S_1, S_2) \leftarrow testcond(S_1), testcond(S_2),\\
&\qquad\qquad not\ incompatible(S_1, S_2), S_1 < S_2,\\
r_{17} &: combinationTested(S_1, S_2) \leftarrow combination(S_1, S_2),\\
&\qquad\qquad test(T, S_1), test(T, S_2),\\
r_{18} &: \leftarrow combination(S_1, S_2), not\ combinationTested(S_1, S_2),\\
r_{19} &: \leftarrow test(T, S_1), test(T, S_2), not\ combination(S_1, S_2), S_1 < S_2.
\end{aligned}$$

Intuitively, r_1 assigns the numbers 1 to n as indices of single tests in any solution while r_2 to r_{10} define testable conditions. Note that not all conditions in Φ_{ex} are testable, and conversely not all testable conditions are conditions in Φ_{ex}. The DL-rule r_{11} states which conditions of the ontology are testable. For that, we send information about which conditions we consider testable to the ontology and query for the extension of the $TestableCond$ concept. Due to the axiom in the ontology, we collect the intersection of conditions in Φ_{ex} and testable conditions of Π_{ex} in the $testcond$ predicate. The DL-rule r_{12} imports incompatible conditions from Φ_{ex} and r_{13} ensures symmetry of the $incompatible$ relation. The DL-rules r_{14} and r_{15} non-deterministically choose whether a given condition holds with respect to a given test case. The $combination$ predicate collects pairs of testable conditions that may occur in the same test case, as realised by

r_{16}. The combinations of testable conditions covered by some test case are derived by r_{17}. This information is used in constraint r_{18} which eliminates answer-set candidates with a combination of testable conditions not tested by any test case. Finally, constraint r_{19} filters out candidates in which two incompatible conditions are jointly tested.

Obtaining a Breakpoint. Every state may serve as a potential starting point for a stepping session. Hence, analogous to stepwise debugging in procedural programming languages, we can consider a state as a breakpoint from which stepping is started. Having a suitable breakpoint at hand will often allow for finding a bug in just a few steps. As mentioned earlier, the empty state $\langle \emptyset, \emptyset, \emptyset, \{\emptyset\}\rangle$ is a trivial state. Besides that, $\langle F, F, \emptyset, \{\emptyset\}\rangle$, where F is the set of all facts in a DL-program, is also ensured to be a state (except for the practically irrelevant case when a literal and its strong negation are jointly asserted).

Example 4. Let us consider the case $n = 7$ in our running example, and let F be the set of facts in Π_{ex}, given thus:

$$F = \{testNo(1), testNo(2), testNo(3), testNo(4), testNo(5), testNo(6), testNo(7),\\ testable(engine\_running), testable(engine\_off), testable(battery\_on),\\ testable(battery\_off), testable(front\_seat\_adult), testable(front\_seat\_child),\\ testable(front\_seat\_empty), testable(roof\_opened), testable(roof\_closed)\}.$$

Note that $S_0 = \langle F, F, \emptyset, \{\emptyset\}\rangle$ is a state. From here, we can start stepping by choosing, e.g., the ground DL-rule $r = state(engine\_running) \leftarrow \mathrm{DL}[Testable \uplus testable; TestableCond](engine\_running)$, being an instance of r_{11} and active under F and Φ_{ex}, as next rule to be added. We obtain the Φ_{ex}-computation $C = S_0, S_1$ with $S_1 = \langle F \cup \{r\}, F \cup \{state(engine\_running)\}, \emptyset, \{\emptyset\}\rangle$. □

Often, it is useful to have states other than the empty or the fact-based state as starting points for stepping, since to reach an answer set I of a DL-program, the minimum length of a computation starting from the empty state is $|I|$. We now discuss how to generate states that may serve as breakpoints using conditions the user finds relevant.

Stable states can be obtained by computing an answer set I of a trusted subset of the DL-rules (or their grounding) and selecting rule instances active under I.

Proposition 1. *Let $\mathcal{KB} = \langle \Phi, \Pi\rangle$ be a DL-program and $\Pi' \subseteq \Pi \cup gr(\Pi)$ such that $I \in \mathrm{AS}(\Pi')^{\Phi}$. Then, $\langle gr(\Pi')^{I}_{\Phi}, I, \bigcup_{r \in gr(\Pi')} \mathrm{Lit}_r \setminus I, \{\emptyset\}\rangle$ is a state.*

Hence, it suffices to find an appropriate Π' in order to get breakpoints. One option for doing so is to let the user manually specify Π' as a subset of Π (including facts).

Example 5. Assume we want to step through the DL-rules that derive the $test/2$ atoms. The respective definitions rely on the available testable conditions of the car. Hence, we use a breakpoint where all instances of DL-rule r_{11}, deriving $testcond/1$-atoms, were already applied. Following Proposition 1, we calculate an answer set of program $\Pi'_{ex} = F \cup \{r_{11}\}$ with respect to Φ_{ex}. The unique answer set of Π'_{ex} is

$$I_3 = F \cup \{testcond(front\_seat\_empty), testcond(front\_seat\_child),\\ testcond(front\_seat\_adult), testcond(battery\_off),\\ testcond(battery\_on), testcond(engine\_off), testcond(engine\_running)\}.$$

The desired breakpoint for subsequent stepping is $S_3 = \langle gr(\Pi'_{ex})^{I_3}_{\Phi_{ex}}, I_3, \emptyset, \{\emptyset\}\rangle$. □

Note that if the subprogram Π' for breakpoint generation has more than one answer set, the selection of the set $I \in \mathrm{AS}(\Pi')^{\Phi}$ is based on the programmer's intuition, similar to selecting the next DL-rule in stepping.

Another use of Proposition 1 is *jumping* from one state to another by considering further non-ground DL-rules. This makes sense, e.g., in a debugging situation where the user initially started with a breakpoint S that is considered as an early state in a computation. After few steps, reaching state $S' = \langle \Pi_{S'}, I_{S'}, I_{S'}{}^{-}, \Upsilon_{S'} \rangle$, the user realises that the computation from S to S' is as intended and wants to proceed to a point where more literals have already been derived, i.e., after applying a selection Π'' of non-ground DL-rules from Π on top of the interpretation $I_{S'}$. Then, Π' from Proposition 1 is given by $\Pi' = \Pi_{S'} \cup \Pi''$. Note that, for an arbitrary answer set I of $\mathrm{AS}(\Pi')^{\Phi}$, it is not ensured that there is a computation starting from S' that ends with the state $gr(\Pi')^{I}_{\Phi}$ because there might be DL-rules in $\Pi_{S'}$ that are not active under I. For assuring that there is a computation of Π starting from S' and ending with $gr(\Pi')^{I}_{\Phi}$, Π' can be joined with $Con_{S'} = \{\leftarrow not\ \ l \mid l \in B^{+}(r), r \in \Pi_{S'}\} \cup \{\leftarrow l \mid l \in B^{-}(r), r \in \Pi_{S'}\}$.

Example 6. Starting from state S_3 obtained in Example 5, we start stepping through the DL-rules that derive the $test/2$ atoms, i.e., instances of DL-rule r_{14}. Hence, as next DL-rule to be added, we choose

$$r' : test(1, engine\_running) \leftarrow testcond(engine\_running), testNo(1), \\ not\ \sim test(engine\_running, 1)$$

and obtain as succeeding state

$$S_4 = \langle \Pi_4, I_3 \cup \{test(1, engine\_running)\}, \{\sim test(1, engine\_running)\}, \{\emptyset\} \rangle,$$

where $\Pi_4 = gr(\Pi'_{ex})^{I_3}_{\Phi_{ex}} \cup \{r'\}$. As the DL-rule instance works as expected, we now want to apply the full non-ground DL-rule r_{14}. We use Proposition 1 by first computing an answer set of $\Pi_5 = \Pi_4 \cup \{r_{14}\}$, e.g.,

$$\begin{aligned} I_5 = I_3 \cup \{ & test(1, engine\_off), test(1, engine\_running), test(1, front\_seat\_adult), \\ & test(2, battery\_on), test(2, engine\_running), \\ & test(2, front\_seat\_empty), test(3, battery\_off), \\ & test(4, battery\_on), test(4, engine\_running), test(4, front\_seat\_child), \\ & test(5, battery\_off), test(5, engine\_off), test(5, front\_seat\_child), \\ & test(6, battery\_off), test(6, front\_seat\_adult), \\ & test(7, battery\_off), test(7, engine\_off), test(7, front\_seat\_empty)\}. \end{aligned}$$

The new state is $S_5 = \langle gr(\Pi_5)^{I_5}_{\Phi_{ex}}, I_5, I_5{}^{-}, \{\emptyset\} \rangle$, where

$$\begin{aligned} I_5{}^{-} = \{ & \sim test(1, engine\_off), \sim test(1, engine\_running), \\ & \sim test(1, front\_seat\_adult), \sim test(2, battery\_on), \\ & \sim test(2, engine\_running), \sim test(2, front\_seat\_empty), \\ & \sim test(3, battery\_off), \sim test(4, battery\_on), \sim test(4, engine\_running), \\ & \sim test(4, front\_seat\_child), \sim test(5, battery\_off), \sim test(5, engine\_off), \\ & \sim test(5, front\_seat\_child), \sim test(6, battery\_off), \\ & \sim test(6, front\_seat\_adult), \sim test(7, battery\_off), \sim test(7, engine\_off), \\ & \sim test(7, front\_seat\_empty)\}. \end{aligned}$$

□

Assisted Stepping. To obtain a Φ-successor of a given state $S = \langle \Pi_S, I, I^-, \Upsilon \rangle$ in the context of a DL-program $\mathcal{KB} = \langle \Phi, \Pi \rangle$, by Definition 10 we need a DL-rule $r \in gr(\Pi) \setminus \Pi_S$ such that $S' = \langle \Pi_S \cup \{r\}, I \cup \Delta, I^- \cup \Delta^-, \Upsilon' \rangle$ is also a state, where $\Delta \cup \Delta^- \subseteq \mathrm{Lit}_r$. One can proceed in the following fashion: First, a non-ground DL-rule $r \in \Pi$ with $H(r) \neq \emptyset$ is selected for instantiation. Then, the user assigns constants to the variables occurring in r. Both steps can be assisted by filtering techniques. Information which non-ground DL-rules in Π have instances that are active under I but not contained in Π_S can be done, e.g., by using the formalism of DL-programs itself, through meta-programming techniques like tagging transformations [12,13].

Example 7. When a computation reached state S_5 of our running example, only ground instances of r_{12}, r_{15}, r_{16}, and r_{19} are active under I_5 but not contained in Π_5. Thus, they can be pre-filtered for the user's disposal. □

Assistence can also be given when the variables in r are assigned one after the other. Then, the domains of the remaining ones can be accordingly restricted such that there is still a compatible ground instance of r that is active under I. Consider a partial substitution ϑ assigning constants in Π to some variables in r. When fixing the assignment of a further variable X occurring in $B(r)$, where $\vartheta(X)$ is yet undefined, we may choose only a constant c such that there is a substitution ϑ' with $\vartheta'(X') = \vartheta(X')$, where $\vartheta(X')$ is defined, $\vartheta'(X) = c$, and $I \models^{\Phi} B(r)\vartheta'$. A simple meta-DL-program can be used to compute potential values of $\vartheta'(X)$, given r, ϑ, and I, checking the above conditions and whether $r\vartheta' \notin \Pi_S$. This is computationally not harder than evaluating the DL-program to debug and in practice often easier, as no guessing is needed in the meta-DL-program.

Once a substitution ϑ for all variables in r is found, Δ and Δ^- must be determined. Again, a system assisting the stepping process can identify respective subsets of $\mathrm{Lit}_r \setminus (I \cup I^-)$ such that the obtained state satisfies the final condition of Definition 10, i.e., that all DL-rules in the potential successor state of S are active.

Example 8. For obtaining a successor of state S_5, we like to apply an instance of

$$r_{12} = \mathit{incompatible}(X, Y) \leftarrow \mathrm{DL}[; \mathit{Incompatible}](X, Y).$$

Two variables are contained in r_{12}, X and Y. Assume we already assigned the constant $\mathit{front\_seat\_adult}$ to variable X. Then, a filtering system can help to find a substitution for Y where $I_5 \models^{\Phi} B(r_{12})\vartheta'$. This amounts to querying for $\mathit{Incompatible}$-successors in Φ_{ex}. The resulting choices for Y are $\mathit{front\_seat\_child}$ and $\mathit{front\_seat\_empty}$. □

Stepping-Based Debugging. Besides getting insights into the interplay of DL-rules of a DL-program, stepping is beneficial when it comes to detecting the reason for an error, i.e., an unexpected outcome, of a DL-program. After a user detected unintended semantics of his or her program, e.g., answer sets that are not expected or missing answer sets, stepping can be started from a state that considers only a trusted subset of the current DL-program. In particular, during development, states obtained in previous stepping sessions can often be reused for extended versions of the evolving DL-program.

Example 9. Let us assume that Π'_{ex} is identical to Π_{ex} except that it does not contain DL-rule r_{13} that ensures the symmetry of the $\mathit{incompatible}$ predicate. Forgetting this

DL-rule can be considered a typical programming mistake. It turns out that Π'_{ex} has an answer set $I_{\Pi'_{ex}}$ such that $I_5 \subseteq I_{\Pi'_{ex}}$. This is in contradiction to our expectations as I_5 contains both $test(1, \mathit{engine\_off})$ and $test(1, \mathit{engine\_running})$, stating that in test 1 the engine of the car is off and running at the same time. Constraint

$$r_{19} = \leftarrow test(T, S_1), test(T, S_2), not\ \ combination(S_1, S_2), S_1 < S_2$$

is meant to eliminate such answer sets. To find the bug, we start stepping at state S_5 and instantiate r_{19}, replacing T by 1, S_1 by $\mathit{engine\_off}$, and S_2 by $\mathit{engine\_running}$. Now, the user sees that the grounded DL-rule is active under I_5 and Φ_{ex}. We have an indication that the atom $combination(\mathit{engine\_off}, \mathit{engine\_running})$ must become true in subsequent steps, as we know that this rule is not active under $I_{\Pi'_{ex}}$ and continue with

$$\begin{aligned} r_{16} = combination(S_1, S_2) \leftarrow\ & testcond(S_1), testcond(S_2), \\ & not\ \ incompatible(S_1, S_2), S_1 < S_2\,, \end{aligned}$$

the only DL-rule in Π'_{ex} that derives atoms of the $combination$ predicate, and substitute S_1 and S_2 like before. Indeed, the resulting ground DL-rule is applicable under I_5 and DL_{ex} as $incompatible(\mathit{engine\_off}, \mathit{engine\_running})$ is not in I_5 which should not be the case in an answer set. Finally, we check the instance

$$\begin{aligned} & incompatible(\mathit{engine\_off}, \mathit{engine\_running}) \leftarrow \\ & \qquad \mathrm{DL}[; \mathit{Incompatible}](\mathit{engine\_off}, \mathit{engine\_running}) \end{aligned}$$

of DL-rule r_{12} being the only one in the grounding of Π'_{ex} that might derive the atom $incompatible(\mathit{engine\_off}, \mathit{engine\_running})$. Here, a query to the ontology reveals that $\mathit{Incompatible}(\mathit{engine\_off}, \mathit{engine\_running})$ is not a consequence of Φ_{ex}. Instead, only $\mathit{Incompatible}(\mathit{engine\_running}, \mathit{engine\_off})$ is asserted in Φ_{ex}. Now it is obvious that the encoding does not enforce the expected symmetry of predicate $incompatible$. □

5 Related Work

There has been little work on debugging DL-programs. In fact, we are only aware of one approach that aims for diagnosing minimal sets of ground DL-atoms in an inconsistent DL-program that would restore consistency when the Boolean results of the query of these DL-atoms are inverted [14]. Besides that, a refined semantics for DL-programs to overcome counter-intuitive results that may emerge when input literals of DL-atoms cause inconsistency in the ontology was presented by Pührer et al. [15].

Within the last years, there has been a considerable amount of work on debugging answer-set programs. The idea of stepping, adapted in this paper for DL-programs, was initially introduced for normal logic programs [4]. Other debugging methods related to stepping are, on the one hand, finding reasons why some interpretation is not an answer set of a program by identifying unfounded loops or unsatisfied rules [12] and, on the other hand, explaining why a program yields no answer sets at all by means of pinpointing unintentionally active constraints [16]. Brain and De Vos [17] used a simple algorithm to recursively find active rules that explain why an atom is in some answer

set or why no such rules exist. Similar debugging questions have been considered by Brain et al. [13] using a meta-programming approach based on ASP itself. While our approach is independent of a concrete solver implementation, a method to directly trace derivations for atoms was realised for the ASP solver DLV [18].

The notion of computations for stepping is related to work by Pontelli et al. [19] who used justifications for similar purposes. Justifications are labelled directed graphs that explain the truth value of a literal l in some answer set in terms of truth values of literals l depends on. Computations in our sense, however, are more related to progressions of active rules, which can be identified in a program following a programmer's intuition that explain how partial interpretations evolve towards answer sets. Quite in the spirit of our notion of computation is that of Liu et al. [20] introduced for characterising the answer sets of logic programs with arbitrary abstract constraints as a sequence of evolving interpretations. As DL-programs can be seen as abstract constraint programs [21], the framework of Liu et al. [20] is, in turn, relevant for our work. Besides differences in the semantics, our notion of computations explicitly takes the rules that are active in some state into account since our motivation is debugging and program analysis.

There exists work focussing on efficient evaluation of restricted classes of DL-programs by rewriting them to datalog with negation [22]. In principle, for this type of DL-programs, the result of this transformation can be used for stepping also through the DL-part of the DL-program. Clearly, this requires the user's familiarity with the translation. In general, for a debugging approach that covers also the ontology part, our approach for stepping of DL-rules can be combined with work on finding faults in DLs. Such methods include explaining concept subsumption [23,24,25], i.e., giving reasons why for given concepts C and D, $C^I \subseteq D^I$ hold for every model I of a TBox $\mathcal{T}$. Subsumption checking can be used to identify incoherent concepts, i.e., concepts that can be proven to have no satisfying instances. Moreover, it can be used to check whether two concepts C and D are equivalent and thus one of them can be regarded as redundant. Another approach is axiom pinpointing [26], where axioms causing a concept to be unsatisfiable with respect to a TBox are detected.

6 Conclusion

We presented a framework for stepping through DL-programs that can be used for debugging based on the intuitions of the user on which DL-rules to apply next. It rests on a computation model where rules are subsequently added to a state. Moreover, we introduced unfounded sets and the notion of external support for DL-programs. We discussed how to obtain states that may serve as breakpoints from which stepping is started. By keeping these breakpoints during development, stepping sessions can be quickly initiated from situations the semantic-web engineer is already familiar with.

As a debugging methodology, our approach focuses on the rules part of a DL-program as the DL ontology is treated as a black box. For future work, it would be interesting to explore how our method can be combined with existing explanation techniques for DLs. That is, when during stepping a DL-atom is or is not satisfied although the opposite is expected, a hybrid debugging approach could provide reasons in terms of axioms and assertions in the DL knowledge base.

References

1. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artificial Intelligence 172(12-13), 1495–1539 (2008)
2. Lifschitz, V.: What is answer set programming? In: Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008), pp. 1594–1597. AAAI Press (2008)
3. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
4. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an Answer-Set Program. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 134–147. Springer, Heidelberg (2011)
5. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence 175(1), 278–298 (2011)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), Professional Book Center, pp. 90–96 (2005)
7. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Proceedings of the 7th International Conference on Logic Programming (ICLP 1990), pp. 579–597 (1990)
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3/4), 365–386 (1991)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K. (eds.) Proceedings of the 5th International Conference on Logic Programming (ICLP 1988), Seattle, WA, USA, pp. 1070–1080. MIT Press (1988)
10. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. Information and Computation 135(2), 69–112 (1997)
11. Faber, W.: Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 40–52. Springer, Heidelberg (2005)
12. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: On debugging non-ground answer-set programs. Theory and Practice of Logic Programming 10(4-5), 513–529 (2010)
13. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP Programs by Means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
14. Pührer, J., El Ghali, A., Chniti, A., Korf, R., Schwichtenberg, A., Lévy, F., Heymans, S., Xiao, G., Eiter, T.: D2.3 Consistency maintenance. Intermediate report. Technical report, ONTORULE IST-2009-231875 Project (2010)
15. Pührer, J., Heymans, S., Eiter, T.: Dealing with Inconsistency When Combining Ontologies and Rules Using DL-Programs. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 183–197. Springer, Heidelberg (2010)
16. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR 2006), Clausthal, Germany, Institut für Informatik, Technische Universität Clausthal, Technical Report, pp. 77–83 (2006)
17. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proceedings of the 3rd Workshop on Answer Set Programming: Advances in Theory and Implementation, ASP 2005. CEUR Workshop Proceedings, CEUR-WS.org, vol. 142, pp. 140–152 (2005)

18. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proceedings of the 2nd Workshop on Software Engineering for Answer Set Programming (SEA 2009), Technical Report 2009-20, University of Bath, pp. 79–93 (2009)
19. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming 9(1), 1–56 (2009)
20. Liu, L., Pontelli, E., Son, T.C., Truszczyski, M.: Logic programs with abstract constraint atoms: The role of computations. Artificial Intelligence 174(3-4), 295–315 (2010)
21. Wang, Y., You, J.H., Yuan, L.Y., Shen, Y.D., Zhang, M.: The loop formula based semantics of description logic programs. Theoretical Computer Science 415, 60–85 (2012)
22. Heymans, S., Eiter, T., Xiao, G.: Tractable reasoning with dl-programs over datalog-rewritable description logics. In: Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010). Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 35–40. IOS Press (2010)
23. McGuinness, D.L., Borgida, A.: Explaining subsumption in description logics. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), pp. 816–821. Morgan Kaufmann (1995)
24. Borgida, A., Franconi, E., Horrocks, I.: Explaining ALC subsumption. In: Proceedings of the 1999 International Workshop on Description Logics (DL 1999). CEUR Workshop Proceedings, CEUR-WS.org, vol. 22, pp. 33–36 (1999)
25. Deng, X., Haarslev, V., Shiri, N.: A resolution based framework to explain reasoning in description logics. In: Proceedings of the 2005 International Workshop on Description Logics (DL 2005). CEUR Workshop Proceedings, CEUR-WS.org, vol. 147 (2005)
26. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: Gottlob, G., Walsh, T. (eds.) Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 355–362. Morgan Kaufmann (2003)