Interactive Answer Set Programming — Preliminary Report —

Martin Gebser[1,3], Philipp Obermeier[3], and Torsten Schaub[2,3⋆]

[1] Aalto University, HIIT
[2] INRIA Rennes
[3] University of Potsdam

Abstract. Traditional Answer Set Programming (ASP) rests upon one-shot solving. A logic program is fed into an ASP system and its stable models are computed. The high practical relevance of dynamic applications led to the development of multi-shot solving systems. An operative system solves continuously changing logic programs. Although this was primarily aiming at dynamic applications in assisted living, robotics, or stream reasoning, where solvers interact with an environment, it also opened up the opportunity of interactive ASP, where a solver interacts with a user. We begin with a formal characterization of interactive ASP in terms of states and operations on them. In turn, we describe the interactive ASP shell *aspic* along with its basic functionalities.

1 Introduction

Traditional logic programming [1, 2] is based upon query answering. Unlike this, logic programming under the stable model semantics [3] is commonly implemented by model generation based systems, viz. answer set solvers [4]. Although the latter also allows for checking whether a (ground) query is entailed by some or all stable models, respectively, there is no easy way to explore a domain at hand by posing consecutive queries without relaunching the solver. The same applies to the interactive addition and/or deletion of temporary program parts that come in handy during theory exploration, for instance, when dealing with hypotheses.

An exemplary area where such exploration capacities would be of great benefit is bio-informatics (cf. [5–10]). Here, we usually encounter problems with large amounts of data, resulting in runs having substantial grounding and solving times. Furthermore, problems are often under-constrained, thus yielding numerous alternative solutions. In such a setting, it would be highly beneficial to explore a domain via successive queries and/or under certain hypotheses. For instance, for determining nutritional requirements for sustaining maintenance or growth of an organism, it is important to indicate seed compounds needed for the synthesis of other compounds [11]. Now, rather than continuously analyzing several thousand stable models (or their intersection or union), a biologist may rather perform interactive "in-silico" experiments by temporarily adding compounds and subsequently exploring the resulting models by posing successive queries.

⋆ Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

We address this shortcoming and show how the multi-shot solving capacities of *clingo* 4 [12] can be harnessed to provide query answering and theory exploration functionalities. In fact, multi-shot ASP was conceived for incorporating online information into operative ASP solving processes. Although this technology was originally devised for dealing with data streams in dynamic environments, like assisted living or cognitive robotics, it can likewise be used to incorporate facts, rules, or queries provided by a user. As a result, we present the semantics, design, and implementation of a system for interactive query answering and theory exploration with ASP. Our ASP shell *aspic* is based on the the answer set solver *clingo* 4 and is implemented as a dedicated front-end.

Our approach along with the resulting system greatly supersede the one of the earlier prototype *quontroller* [13], which itself was based on the archetypal reactive solver *oclingo*. The *clingo* 4 series goes beyond its predecessors by providing control capacities that can be used either through scripting inlets in ASP encodings or corresponding APIs in Lua and Python.[4] Given this, *aspic* does not only provide a much more general framework than *quontroller*, but is moreover based on firm semantic underpinnings that we develop in Section 2. In Section 3, we illustrate the functionality and implementation of *aspic*, and Section 4 concludes the paper.

2 Operational Semantics

This section defines an operational semantics in terms of system states and operations modifying such states. We confine ourselves to a propositional setting but discuss query answering with variables in Section 2.3. To begin with, we introduce the major ingredients of our semantics.

We consider logic programs and assignments over an alphabet $\mathcal{A}$ of ground atoms. A rule r is of the form $h \leftarrow \ell_1, \ldots, \ell_n$ with head $h = a$, $h = \{a\}$, or $h = \bot$, where $a \in \mathcal{A}$ and $\bot$ is a constant denoting falsity, and body $\ell_1, \ldots, \ell_n$ such that $\ell_i = a_i$ or $\ell_i = \sim a_i$ for $1 \leq i \leq n$, where $a_i \in \mathcal{A}$ and $\sim$ stands for default negation. We denote the head atom a or $\bot$, respectively, by $head(r)$, and $body(r) = \{\ell_1, \ldots, \ell_n\}$ is the set of body literals of r. Given a literal ℓ, we define $\overline{\ell}$ as $\sim a$, if $\ell = a$, and as a, if $\ell = \sim a$, for $a \in \mathcal{A}$. For a set L of literals, let $L^+ = L \cap \mathcal{A}$ and $L^- = \{\overline{\ell} \mid \ell \in L \setminus \mathcal{A}\}$ denote the sets of atoms occurring positively or negatively, respectively, in L. A logic program P is a finite set of rules. The set of head atoms of P is $head(P) = \{head(r) \in \mathcal{A} \mid r \in P\}$; note that $\bot \notin head(P)$ even when $head(r) = \bot$ for some $r \in P$, and below we skip $\bot$ when writing rules without head atom. Moreover, we use $atom(\phi)$ to refer to all atoms occurring in ϕ, no matter whether ϕ is a program or any other type of logical expression. Finally, we let $AS(P)$ stand for all stable models of P.

An assignment i over a set $A \subseteq \mathcal{A}$ of atoms is a function $i : A \to \{t, f, u\}$. We often refer to an assignment i in terms of three characteristic functions: $i^t = \{a \in \mathcal{A} \mid i(a) = t\}$, $i^f = \{a \in \mathcal{A} \mid i(a) = f\}$, and $i^u = \{a \in \mathcal{A} \mid i(a) = u\}$.

Given this, a *system state* over $\mathcal{A}$ is defined as a quadruple

$$(R, I, i, j)$$

[4] Details can be found in [12]. Similarly, we refer the reader to the literature for introductions to ASP [14, 4] and the input language of *clingo* 4 [15].

where

- R is a ground program over $\mathcal{A}$,
- $I \subseteq \mathcal{A} \setminus head(R)$ is a set of input atoms,
- i is an assignment over I, and
- j is an assignment over $\mathcal{A}$.

A system state (R, I, i, j) induces the following logic program

$$P(R, I, i, j) = R \cup \{a \leftarrow \mid a \in i^t\} \cup \{\{a\} \leftarrow \mid a \in i^u\}$$
$$\cup \{ \leftarrow \sim a \mid a \in j^t\} \cup \{ \leftarrow a \mid a \in j^f\}$$

Note the different effects of assignments i and j. Changing truth values in i amounts to data manipulation, while changing the ones in j boils down to a data filter. In fact, atoms true in i are exempt from the unfoundedness criterion, "undefined"[5] atoms give rise to two alternatives, and false atoms are left to the semantics and thus set to false. Unlike this, assignment j acts as a mere filter on the set of stable models; in particular, atoms true in j are subject to the unfoundedness criterion, while an atom "undefined" in j imposes no constraint. Accordingly, i and j have also a different default behavior. While i defaults to false, j defaults to "undefined", and we thus leave $a \mapsto f$ or $a \mapsto u$, respectively, implicit when specifying assignments i and j.

2.1 State Changing Operations

We now start by fixing the operational semantics of our approach in terms of state changing operations. A state changing operation is a function mapping a system state to another, depending on an extra input parameter.

Assumption and Cancellation. Operator $assume$ takes a ground literal ℓ and transforms a system state as follows

$$assume : \langle \ell, (R, I, i, j_1)\rangle \mapsto (R, I, i, j_2)$$

where

- $j_2^t = j_1^t \cup \{\ell\}$ if $\ell \in \mathcal{A}$, and $j_2^t = j_1^t \setminus \{\overline{\ell}\}$ otherwise
- $j_2^f = j_1^f \cup \{\overline{\ell}\}$ if $\overline{\ell} \in \mathcal{A}$, and $j_2^f = j_1^f \setminus \{\ell\}$ otherwise
- $j_2^u = \mathcal{A} \setminus (j_2^t \cup j_2^f)$

For example, $assume(a, (\{\{a\} \leftarrow\}, \emptyset, \emptyset, \emptyset))$ is associated with program

$$P(\{\{a\} \leftarrow\}, \emptyset, \emptyset, \{a \mapsto t\}) = \{\{a\} \leftarrow\} \cup \{ \leftarrow \sim a\}$$

having the stable model $\{a\}$ only. Unlike this, $assume(a, (\emptyset, \emptyset, \emptyset, \emptyset))$ induces program $P(\emptyset, \emptyset, \emptyset, \{a \mapsto t\}) = \{ \leftarrow \sim a\}$, which has no stable model.

Operator $cancel$ takes a ground literal ℓ and transforms a system state as follows

$$cancel : \langle \ell, (R, I, i, j_1)\rangle \mapsto (R, I, i, j_2)$$

where

[5] Strictly speaking, such atoms are not undefined but assigned value u.

- $j_2^t = j_1^t \setminus \{\ell\}$
- $j_2^f = j_1^f \setminus \{\overline{\ell}\}$
- $j_2^u = \mathcal{A} \setminus (j_2^t \cup j_2^f)$

For example, $cancel(a, assume(a, (\{\{a\} \leftarrow\}, \emptyset, \emptyset, \emptyset)))$ yields the two stable models of $\{\{a\} \leftarrow\}$. More generally, we observe the following interrelationships between the above operations. For a system state $S = (R, I, i, j)$ and a literal ℓ:

$$\begin{aligned} cancel(\ell, assume(\ell, S)) &= S \text{ if } \ell \notin j^t \cup j^f \text{ and } \overline{\ell} \notin j^t \cup j^f \\ cancel(\ell, cancel(\ell, S)) &= cancel(\ell, S) \\ assume(\ell, cancel(\ell, S)) &= assume(\ell, S) \\ assume(\ell, assume(\ell, S)) &= assume(\ell, S) \end{aligned}$$

Assertion, Opening, and Retraction. Operator $assert$ takes a ground atom a and transforms a system state as follows

$$assert : \langle a, (R, I, i_1, j)\rangle \mapsto (R, I, i_2, j)$$

where

- $i_2^t = i_1^t \cup \{a\}$ if $a \in I$, and $i_2^t = i_1^t$ otherwise
- $i_2^u = i_1^u \setminus \{a\}$
- $i_2^f = I \setminus (i_2^t \cup i_2^u)$

Note that, unlike with $assume$, the atom of an asserted literal must belong to the input atoms.

For example, $assert(a, (\emptyset, \{a\}, \emptyset, \emptyset))$ is associated with program

$$P(\emptyset, \{a\}, \{a \mapsto t\}, \emptyset) = \{a \leftarrow\}$$

having the only stable model $\{a\}$.

Operator $open$ takes a ground atom a and transforms a system state as follows

$$open : \langle a, (R, I, i_1, j)\rangle \mapsto (R, I, i_2, j)$$

where

- $i_2^t = i_1^t \setminus \{a\}$
- $i_2^u = i_1^u \cup \{a\}$ if $a \in I$, and $i_2^u = i_1^u$ otherwise
- $i_2^f = I \setminus (i_2^t \cup i_2^u)$

Continuing the above example, $open(a, assert(a, (\emptyset, \{a\}, \emptyset, \emptyset))) = open(a, (\emptyset, \{a\}, \{a \mapsto t\}, \emptyset))$ leads to

$$P(\emptyset, \{a\}, \{a \mapsto u\}, \emptyset) = \{\{a\} \leftarrow\}$$

having the empty stable model in addition to $\{a\}$.

Operator $retract$ takes a ground atom a and transforms a system state as follows

$$retract : \langle a, (R, I, i_1, j)\rangle \mapsto (R, I, i_2, j)$$

where

- $i_2^t = i_1^t \setminus \{a\}$
- $i_2^u = i_1^u \setminus \{a\}$
- $i_2^f = I \setminus (i_2^t \cup i_2^u)$

For example, $retract(a, (\emptyset, \{a\}, \{a \mapsto t\}, \emptyset))$ yields $P(\emptyset, \{a\}, \emptyset, \emptyset) = \emptyset$, whose stable model is empty as well. More generally, we observe the following interrelationships between the above operations. For a system state $S = (R, I, i, j)$ and an atom a:

$$\begin{aligned} assert(a, S) &= S \text{ if } a \notin I \\ open(a, S) &= S \text{ if } a \notin I \\ retract(a, assert(a, S)) &= S \text{ if } a \notin i^t \cup i^u \\ retract(a, open(a, S)) &= S \text{ if } a \notin i^t \cup i^u \\ o(a, o'(a, S)) &= o(a, S) \text{ for } o, o' \in \{assert, open, retract\} \end{aligned}$$

Definition, External, and Release. Operator $define$ takes a set R of ground rules and transforms a system state as follows

$$define : \langle R, (R_1, I_1, i, j)\rangle \mapsto (R_2, I_2, i, j)$$

where

- $R_2 = C_{I_1}(R_1 \cup R)$ if R_1 and R are compositional, and $R_2 = R_1$ otherwise
- $I_2 = I_1 \setminus head(R_2)$

Function C confines a program to the atom base of the current system state. More precisely, a program P and a set I of input atoms induce the atom base $B = I \cup head(P)$ with which we define $C_I(P)$ as:

$$\{head(r) \leftarrow body(r)^+ \cup \{\sim a \mid a \in body(r)^- \cap B\} \mid r \in P, body(r)^+ \subseteq B\}$$

This restricts the rules in P to all atoms available as heads in P or as input atoms in I.

Given that *clingo*'s program composition follows the one of module theory [16], the same criteria apply to the implementation of $define$ in *aspic*: Two programs P and Q are *compositional*, if

- $head(P) \cap head(Q) = \emptyset$ and
- for every strongly connected component C in the positive dependency graph of $P \cup Q$, we have $head(P) \cap C = \emptyset$ or $head(Q) \cap C = \emptyset$.

In words, atoms must not be redefined and no mutual positive recursion is permissible in the joint program. More liberal definitions are possible but not supported by the current framework of *aspic*.

As an example, consider $define(\{a \leftarrow \sim b\}, (\emptyset, \emptyset, \emptyset, \emptyset))$. The state and added program induce atom base $\{a\}$. With it, the resulting state is captured by the simplified program $\{a \leftarrow\}$, having a single stable model containing a. Adding afterwards b via $define(\{b \leftarrow\}, define(\{a \leftarrow \sim b\}, (\emptyset, \emptyset, \emptyset, \emptyset))$ results in program $\{a \leftarrow, b \leftarrow\}$ along with a stable model containing a and b.

For another example, consider the operation $\mathit{define}(\{a \leftarrow b, b \leftarrow a\}, (\emptyset, \emptyset, \emptyset, \emptyset))$. This state results in an empty stable model, whereas the operator composition $\mathit{define}(\{a \leftarrow b\}, \mathit{define}(\{b \leftarrow a\}, (\emptyset, \{a\}, \emptyset, \emptyset)))$ is noneffective because a positive cycle would be obtained in the joined program.

Similarly, operation $\mathit{define}(\{a \leftarrow b, a \leftarrow \sim c\}, (\emptyset, \{b, c\}, \emptyset, \emptyset))$ is well-defined, while $\mathit{define}(\{a \leftarrow b\}, \mathit{define}(\{a \leftarrow \sim c\}, (\emptyset, \{b, c\}, \emptyset, \emptyset)))$ is not, and the second define operation is thus vacuous.

Operator $\mathit{external}$ takes a ground atom a and transforms a system state as follows

$$\mathit{external} : \langle a, (R, I_1, i, j)\rangle \mapsto (R, I_2, i, j)$$

where

- $I_2 = I_1 \cup (\{a\} \setminus \mathit{head}(R))$

The $\mathit{external}$ operator allows for introducing atoms whose truth value can be controlled externally (via assert, open, and $\mathit{retract}$) or defined by rules later on. Note that this is only possible if an atom is not already defined.

Let us reconsider the above example in conjunction with $\mathit{external}$ operations. At first, consider $\mathit{define}(\{a \leftarrow \sim b\}, \mathit{external}(b, (\emptyset, \emptyset, \emptyset, \emptyset)))$. The state and added program induce now atom base $\{a, b\}$. With it, the resulting state is captured by the program $\{a \leftarrow \sim b\}$ having a single stable model containing a. Adding afterwards b via

$$\mathit{define}(\{b \leftarrow\}, \mathit{define}(\{a \leftarrow \sim b\}, \mathit{external}(b, (\emptyset, \emptyset, \emptyset, \emptyset))))$$

results in program $\{a \leftarrow \sim b, b \leftarrow\}$, which now has a stable model containing only b (but not a).

Operator $\mathit{release}$ takes a ground atom a and transforms a system state as follows

$$\mathit{release} : \langle a, (R_1, I_1, i_1, j)\rangle \mapsto (R_2, I_2, i_2, j)$$

where

- $R_2 = R_1 \cup \{a \leftarrow a\}$ if $a \in I_1$, and $R_2 = R_1$ otherwise
- $I_2 = I_1 \setminus \{a\}$
- $i_2^v = i_1^v \setminus \{a\}$ for $v \in \{t, f, u\}$

The purpose of adding rule $a \leftarrow a$ is two-fold. First, it prevents new rules defining a to be added and, second, assures that a remains false.[6] To see this, observe that the compositionality criterion of define prohibits the re-definition of an atom, and that atoms must be among the input atoms to be manipulated by assert, open, or $\mathit{retract}$.

As before, we observe some interrelationships between the above operations. For a system state $S = (R_1, I_1, i, j)$ and an atom a:

$$\begin{aligned}
\mathit{release}(a, S) &= S \text{ if } a \notin I_1 \\
\mathit{external}(a, S) &= S \text{ if } a \in I_1 \cup \mathit{head}(R_1) \\
\mathit{external}(a, \mathit{release}(a, S)) &= \mathit{release}(a, S) \text{ if } a \in I_1 \cup \mathit{head}(R_1) \\
\mathit{define}(R, \mathit{release}(a, S)) &= \mathit{release}(a, S) \text{ if } a \in \mathit{head}(R) \cap (I_1 \cup \mathit{head}(R_1)) \\
o(a, \mathit{release}(a, S)) &= \mathit{release}(a, S) \text{ for } o \in \{\mathit{assert}, \mathit{open}, \mathit{retract}\}
\end{aligned}$$

[6] In terms of module theory, this amounts to adding a to the output atoms of the module capturing the system state.

2.2 Ground Queries

In analogy to database systems, we map query answering to testing membership in a (single) set. This set is obtained by consolidating a program's stable models according to two criteria, namely a model *filter* and an entailment *mode*. A filter maps a collection of sets of ground atoms to a subset of the collection. Examples include the identity function and functions selecting optimal models according to some objective function. A mode maps a collection of sets of ground atoms to a subset of the union of the collection. Although we confine ourselves to intersection and union, many other modes are possible, like the intersection of a specific number of models or the complement of the union, etc. Combining a filter with a mode lets us turn the semantics of a logic program, given by its set of stable models, into a single set of atoms of interest, for instance, the atoms true in all optimal stable models (relative to `#minimize` statements).

The operator $query$ maps an atomic query $q \in \mathcal{A}$, an entailment mode μ, a model filter ν, and a system state $S = (R, I, i, j)$ to the set $\{yes, no\}$:

$$query(q, (\mu, \nu), S) = \begin{cases} yes & \text{if } \; q \in \mu \circ \nu(AS(P(S))) \\ no & \text{otherwise} \end{cases}$$

Taking $\mu = \cap$ and $\nu = id$ (the identity function) amounts to testing whether a is a skeptical consequence of program $P(S)$; accordingly, taking $\cup$ instead of $\cap$ checks for credulous consequence.

A *Boolean query* ϕ is an expression over $\mathcal{A}$ (in negation normal form) obtained from connectives $\sim$, $\wedge$, and $\vee$ in the standard way. To handle such queries, we define a function Q as follows:

$$\begin{aligned} Q(\phi) &= \{q_\phi \leftarrow \phi\} \text{ if } \phi \in \mathcal{A} \\ Q(\sim\phi) &= \{q_{\sim\phi} \leftarrow \sim q_\phi\} \cup Q(\phi) \\ Q(\phi_1 \wedge \phi_2) &= \{q_{\phi_1 \wedge \phi_2} \leftarrow q_{\phi_1}, q_{\phi_2}\} \cup Q(\phi_1) \cup Q(\phi_2) \\ Q(\phi_1 \vee \phi_2) &= \left\{ \begin{array}{l} q_{\phi_1 \vee \phi_2} \leftarrow q_{\phi_1} \\ q_{\phi_1 \vee \phi_2} \leftarrow q_{\phi_2} \end{array} \right\} \cup Q(\phi_1) \cup Q(\phi_2) \end{aligned}$$

where $head(Q(\phi)) \cap (atom(R) \cup I \cup atom(\phi)) = \emptyset$ for a system state (R, I, i, j).

With this, the above $query$ operation can be extended to Boolean queries as follows:

$$query(\phi, (\mu, \nu), S) \quad \text{iff} \quad query(q_\phi, (\mu, \nu), define(Q(\phi), S))$$

A more sophisticated way of applying the $query$ operator to a Boolean query ϕ in a system state $S = (R, I, i, j)$ is to execute the following sequence of operations:

$$\begin{aligned} S_1 &= external(e, S) \\ S_2 &= define(ext(Q(\phi), e), S_1) \\ S_3 &= assert(e, S_2) \\ v &= query(q_\phi, (\mu, \nu), S_3) \\ S' &= release(e, S_3) \end{aligned}$$

where $e \in \mathcal{A} \setminus (atom(R \cup Q(\phi)) \cup I)$ is a fresh atom and $ext(Q(\phi), e) = \{head(r) \leftarrow body(r) \cup \{e\} \mid r \in Q(\phi)\}$. As with $query(q_\phi, (\mu, \nu), \mathit{define}(Q(\phi), S))$, we have that $query(q_\phi, (\mu, \nu), S_3)$ yields the value v of $query(\phi, (\mu, \nu), S)$. The respective system state S_3, however, differs from $\mathit{define}(Q(\phi), S)$: Rules from $Q(\phi)$ are in $ext(Q(\phi), e)$ annotated with a dedicated input atom e. Hence, releasing e leads to a state S' such that rules as well as head atoms of $ext(Q(\phi), e)$ are obsolete and can be deleted (cf. [17]).

2.3 Non-Ground Operations

So far, we considered system states and operations exclusively in the ground case. Let us now lay out how this lifts to the non-ground case.

As usual, a program with variables is regarded as an abbreviation for its ground instantiation. Accordingly, the program R in a state (R, I, i, j) can be non-ground and its instantiation is left to *clingo* 4 (or *gringo* 4, to be precise). Likewise, the state-changing operations define and $external$ accept respective non-ground expressions. However, here safety of the defined rules and external declarations is an additional requirement (cf. [15]). All other operations still deal with ground atoms.

Analogously, we provide means for asking non-ground queries. To this end, we extend the $query$ operator to support non-ground conjunctive queries (but refrain from general Boolean queries for the sake of using standard safety conditions). Specifically, a conjunctive query ϕ is a conjunction of (possibly non-ground) literals, which can be represented by a rule $q_\phi \leftarrow \phi$, provided that variable occurrences in ϕ are safe. With this, we can extend the $query$ operation to non-ground conjunctive queries as follows:

$$query(\phi, (\mu, \nu), S) \quad \text{iff} \quad query(q_\phi, (\mu, \nu), \mathit{define}(\{q_\phi \leftarrow \phi\}, S))$$

Note that this definition treats variables in ϕ existentially, and different instances of $q_\phi \leftarrow \phi$ provide alternatives for deriving the target q_ϕ of an associated atomic query.

3 The *aspic* System

Following our initial motivation to design a human-oriented interface, we implemented the foregoing concepts and primitives in form of an interactive ASP shell, dubbed *aspic*.[7] With this system, users can continuously enter state-changing operations to dynamically load, define, and change an ASP knowledge base as well as pose queries to explore the induced stable models. Technically, *aspic* is implemented in Python using *clingo* 4 as back-end through its Python API. Subsequently, we illustrate the functionality and typical workflow of *aspic* based on an encoding for graph coloring. After that, we give a more detailed account of the system implementation.

Initially, Listing 1 provides an ASP encoding for n-coloring. The encoding expects a graph, that is, a set of facts over predicate `edge`/2 representing its edges. Given such facts, the nodes are extracted in Line 7–8. Then, Line 11 makes sure that each node is marked with exactly one of the n colors, and Line 12 forbids that adjacent nodes have the same color. Finally, the resulting stable models are projected to predicate `mark`/2

[7] Available at `http://potassco.sourceforge.net/labs.html`.

```
1  % n-coloring
2  #const n = 3.
3  #const m = 10.
4  color(1..n).

6  % Extract nodes from edges
7  node(X) :- edge(X,_).
8  node(X) :- edge(_,X).

10 % Generate n-coloring
11 1 { mark(X,C) : color(C) } 1 :- node(X).
12 :- edge(X,Y), mark(X,C), mark(Y,C).

14 % Allow aspic user to add and remove edges via 'external'
15 #external edge(X,Y) : X = 1..m-1, Y = X+1..m.

17 % Display n-coloring
18 #show mark/2.
```

Listing 1. ASP encoding of n-coloring (`ncoloring.lp`)

```
1  ?- assert edge(1,2)
2  ?- assert edge(1,4)
3  ?- assert edge(2,3)
4  ?- assert edge(3,4)
```

Listing 2. Building up an initial graph by asserting edges

via Line 18. However, other than in classical one-shot solving, the encoding in Listing 1 is not only combinable with a fixed graph, but also allows an *aspic* user to dynamically provide and change graphs. To achieve this, we consider instances of `edge`/2 as input atoms, which is reflected by declaring them as external in Line 15.

To use *aspic* on Listing 1, we run '`aspic`' from the terminal, upon which it greets us with its prompt prefix '`?-`', waiting for input. Now, we enter '`?- load ncoloring.lp`' to load the file `ncoloring.lp`. Alternatively, we could have loaded `ncoloring.lp` immediately at invocation time by starting *aspic* with '`aspic ncoloring.lp`'. An overview of all supported commands can be obtained by entering '`?- help`'. In the following, we illustrate a typical *aspic* session, i.e., a sequence of *aspic* command interactions, documented throughout Listing 2–6.

First, we add an initial graph to the system by asserting the instances of `edge`/2 given in Listing 2. Then, we ask an atomic query over the atom `mark(1,1)`, leading to the stable model displayed in Listing 3. By default, the **`query`** command returns a single model matching its query along with a corresponding satisfiability status, as shown in Line 6–7. Specifically, for an atomic query q in the current system state S, the model returned by **`query`** is the first one found by *clingo* 4 run on the program $P(S)$, using q as an assumption (cf. [17]); in case of a Boolean query ϕ, the program and the assumption for solving change to $P(S) \cup Q(\phi)$ and q_ϕ. In either case, the satisfiability

```
5  ?- query mark(1,1)
6  Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,3)]
7  SAT

9  ?- option -n 0

11 ?- query mark(1,1)
12 Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,3)]
13 Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,2)]
14 Model: [mark(1,1), mark(2,3), mark(3,2), mark(4,3)]
15 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
16 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,2)]
17 Model: [mark(1,1), mark(2,2), mark(3,3), mark(4,2)]
18 SAT

20 ?- option -e brave

22 ?- query mark(1,1)
23 Model: [mark(1,1), mark(2,2), mark(2,3), mark(3,1),
24         mark(3,2), mark(3,3), mark(4,2), mark(4,3)]
25 SAT

27 ?- option -e cautious

29 ?- query mark(1,1)
30 Model: [mark(1,1)]
31 SAT

33 ?- option -e auto
```

Listing 3. Querying with different entailment modes

status, `SAT` or `UNSAT`, indicates whether some stable model matching the assumption could be found. The number of models returned by **`query`** can be changed via the **`option`** command[8] in Line 9, taking *clingo* 4 command line options as argument and forwarding them unaltered to the underlying *clingo* 4 process. Here, argument '`-n 0`' instructs *clingo* 4 to enumerate all models rather than stopping at the first one. With that, querying again for `mark(1,1)` in Line 11 returns the models listed in Line 12–17. Similarly, the entailment modes $\cup$ and $\cap$ can be activated via the **`option`** commands in Line 20 and 27, where the arguments '`-e brave`' and '`-e cautious`' instruct *clingo* 4 to return the union or intersection of all stable models matching a query. Regarding stable models including `mark(1,1)`, the union and intersection are obtained as results in Line 23–24 or Line 30, respectively. To switch the behavior of *clingo* 4 back to model enumeration, we then use '**`option`** `-e auto`' in Line 33. Note that, in the current *aspic* version, only identity is supported as model filter and, hence, always used.[9]

[8] Keywords are preliminary and may be subject to change in future releases.

[9] The addition of optimization as filter is planned for an upcoming release.

```
34 ?- query mark(1,1) & [ mark(3,2) | not mark(4,2) ]
35 Model: [mark(1,1), mark(2,3), mark(3,2), mark(4,3)]
36 Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,3)]
37 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
38 SAT

40 ?- assert edge(2,4)

42 ?- query mark(1,1) & [ mark(3,2) | not mark(4,2) ]
43 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
44 SAT

46 ?- open edge(2,4)

48 ?- query mark(1,1) & [ mark(3,2) | not mark(4,2) ]
49 Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,3)]
50 Model: [mark(1,1), mark(2,3), mark(3,2), mark(4,3)]
51 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
52 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
53 SAT

55 ?- retract edge(2,4)

57 ?- query mark(1,1) & [ mark(3,2) | not mark(4,2) ]
58 Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,3)]
59 Model: [mark(1,1), mark(2,3), mark(3,2), mark(4,3)]
60 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
61 SAT
```

Listing 4. Interplay of a Boolean query with *assert*, *open*, and *retract*

As next step, we enter the Boolean query displayed in Listing 4, asking for stable models such that node 1 is marked with color 1 and node 3 with color 2 or node 4 with another color than 2. Note that the connectives $\sim$, $\wedge$, and $\vee$ are represented by '`not`', '`&`', and '`|`' and that subexpressions are parenthesized by square brackets. The resulting models are shown in Line 35–37, followed by the corresponding satisfiability status. Asserting an additional edge between node 2 and 4 in Line 40 restricts outcomes to the single model in Line 43. On the other hand, opening `edge(2,4)` in Line 46 reintroduces stable models obtained before (found in different order due to inherent non-determinisms in search), and also duplicates the one in which node 2 and 4 have different colors because `edge(2,4)` can be made true or false in this case, while neither alternative is visible in the projected output. After retracting `edge(2,4)` in Line 55, the duplication disappears, and we receive the same (number of) stable models as in the beginning. Again note that the order of models can vary, given that dynamic conflict information and search heuristics depend on previous runs in multi-shot solving.

Listing 5 illustrates how assumptions, essentially corresponding to atomic queries, can be incorporated. The effect of assuming '`not mark(2,3)`' in Line 62 is that stable

```
62 ?- assume not mark(2,3)

64 ?- query mark(1,1)
65 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
66 Model: [mark(1,1), mark(2,2), mark(3,3), mark(4,2)]
67 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,2)]
68 SAT

70 ?- query mark(1,1) & [ mark(3,2) | not mark(4,2) ]
71 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,3)]
72 SAT

74 ?- cancel not mark(2,3)
```

Listing 5. Querying under an assumption

```
75 ?- external elim(X,C) : X = 1..m, color(C)

77 ?- define :- edge(X,Y), elim(X,C), mark(Y,C).
78          :- edge(X,Y), mark(X,C), elim(Y,C). ?

80 ?- assert elim(2,3)
81 ?- assert elim(4,2)

83 ?- query mark(X,1) & elim(X,C)
84 UNSAT

86 ?- query mark(X,C) & elim(X,C)
87 Model: [mark(1,1), mark(2,2), mark(3,1), mark(4,2)]
88 Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,2)]
89 Model: [mark(1,1), mark(2,3), mark(3,1), mark(4,3)]
90 SAT
```

Listing 6. Performing non-ground operations

models such that node 2 is marked with color 3 are disregarded until the assumption is canceled in Line 74. As a consequence, the atomic or Boolean, respectively, queries in Line 64 and 70 yield proper subsets of the stable models that had been obtained before.

Finally, in Listing 6, we make use of non-ground expressions to which *define*, *external*, and *query* operations are extended. In Line 75, instances of `elim`/2 are declared as external, where the arguments `X` and `C` range over all potential nodes or available colors, respectively. Integrity constraints forbidding that nodes adjacent to `X` are marked with `C` are then defined in Line 77–78.[10] The assertions in Line 80–81 thus exclude that neighbors of node 2 and 4 are marked with color 3 or 2, respectively. As indicated by the outcome `UNSAT` of asking the non-ground query in Line 83, it is now

[10] Symbol '`?`' at the end of Line 78 is an *aspic* token indicating the end of a collection of rules serving as argument of a *define* operation, so that such rules can be written in multiple lines.

impossible to mark node 2 or 4 with color 1. The second non-ground query in Line 86 admits stable models such that node 2 is marked with color 3 or node 4 with color 2, which yields three stable models matching some instance of the query.

As mentioned above, *aspic* is built on top of the Python API of *clingo* 4, accessible via the `gringo` module. Throughout a user session, the system state (R, I, i, j) is captured by a `gringo.Control` object. In particular, both **`define`** and **`external`** commands rely on the API functions `add` and `ground` to incorporate rules and external declarations into the system state, thus extending R or I, respectively. The "inverse" command **`release`** uses the API function `release_external` to remove an external atom from I; this includes the deletion of rules containing a released atom in their body. The assignment i as well as corresponding **`assert`**, **`open`** and **`retract`** commands rely on the API function `assign_external` to manipulate (external) input atoms in I. Unlike this, assignment j is expressed by assumptions, managed via **`assume`** and **`cancel`** and then passed to the API function `solve`, which is invoked upon **`query`** commands. The **`query`** command for an atomic query q is implemented by taking the atom q as an additional (non-persistent) assumption. More sophisticated Boolean or non-ground queries ϕ are processed via $ext(Q(\phi), e)$ or $ext(\{q_\phi \leftarrow \phi\}, e)$, respectively, and the corresponding sequence of state-changing operations given in Section 2.2, which are in turn mapped to API functions as explained above. That is, after processing a query, related rules and assumptions are withdrawn, so that the system state remains largely the same apart from internal data structures of *clingo* 4 that are updated during search.

4 Discussion

We presented a simple yet effective application of multi-shot ASP solving that allows for interactive query answering and theory exploration. This was accomplished by means of the multi-shot solving capacities of *clingo* 4. The possibility of reusing (ground) rules as well as recorded conflict information over a sequence of queries distinguishes multi-shot ASP solving from the basic one-shot approach. With it, program parts can be temporarily added to the solving process until they are withdrawn again. A typical use case are queries that automatically vanish after having been processed. Moreover, assertions allow for exploring a domain under user-defined hypotheses, included in all subsequent solving processes until the asserted information is retracted.

Queries and query answering play a key role in knowledge representation and the design of intelligent agents [14]. Similar to our work, *ASP-Prolog* [18] and *XASP* [19] provide frameworks to interactively explore ASP programs. In contrast to *aspic*, however, both systems repeatedly invoke the stand-alone ASP solver *smodels* [20] to compute stable models. While the ASP system *dlv* [21] supports (safe) conjunctive queries along with skeptical or credulous reasoning, it also needs to be relaunched upon each query, whereas *aspic* processes queries within an operative ASP solving process.

As future work, we plan to support non-trivial filters, e.g., optimization, and will improve *aspic*'s functionality based on user feedback and experience. In particular, we aim at generalizing the use of first-order expressions within queries as well as state-changing operations.

Acknowledgments This work was partially funded by DFG grant SCHA 550/9-1+2.

References

1. Clocksin, W., Mellish, C.: Programming in Prolog. Springer-Verlag (1981)
2. Lloyd, J.: Foundations of Logic Programming. Springer-Verlag (1987)
3. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88). MIT Press (1988) 1070–1080
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
5. Baral, C., Chancellor, K., Tran, N., Tran, N., Joy, A., Berens, M.: A knowledge based approach for representing and reasoning about signaling networks. In: Proceedings of the Twelfth International Conference on Intelligent Systems for Molecular Biology (ISMB'04) and the Third European Conference on Computational Biology (ECCB'04). Oxford University Press (2004) 15–22
6. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. In Fox, D., Gomes, C., eds.: Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08). AAAI Press (2008) 436–441
7. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. Theory and Practice of Logic Programming **11**(2-3) (2011) 323–360
8. Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., Veber, P.: Repair and prediction (under inconsistency) in large biological networks with answer set programming. In Lin, F., Sattler, U., eds.: Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10). AAAI Press (2010) 497–507
9. Ray, O., Whelan, K., King, R.: Logic-based steady-state analysis and revision of metabolic networks with inhibition. In Barolli, L., Xhafa, F., Vitabile, S., Hsu, H., eds.: Proceedings of the Fourth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'10). IEEE Computer Society (2010) 661–666
10. Videla, S., Guziolowski, C., Eduati, F., Thiele, S., Grabe, N., Saez-Rodriguez, J., Siegel, A.: Revisiting the training of logic models of protein signaling networks with ASP. In Gilbert, D., Heiner, M., eds.: Proceedings of the Tenth International Conference on Computational Methods in Systems Biology (CMSB'12). Springer-Verlag (2012) 342–361
11. Schaub, T., Thiele, S.: Metabolic network expansion with ASP. In Hill, P., Warren, D., eds.: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). Springer-Verlag (2009) 312–326
12. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Preliminary report. In Leuschel, M., Schrijvers, T., eds.: Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14). Theory and Practice of Logic Programming **14**(4-5) (2014) Online supplement
13. Gebser, M., Obermeier, P., Schaub, T.: A system for interactive query answering with answer set programming. In Fink, M., Lierler, Y., eds.: Proceedings of the Sixth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'13). CoRR (2013)
14. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
15. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S.: Potassco User Guide. `http://potassco.sourceforge.net` (2015)
16. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In Brewka, G., Coradeschi, S., Perini, A., Traverso, P., eds.: Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06). IOS Press (2006) 412–416

17. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Springer-Verlag (2008) 190–205
18. El-Khatib, O., Pontelli, E., Son, T.: ASP-PROLOG: A system for reasoning about answer set programs in Prolog. In Jayaraman, B., ed.: Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04). Springer-Verlag (2004) 148–162
19. Castro, L., Warren, D.: An environment for the exploration of non monotonic logic programs. In Kusalik, A., ed.: Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE'01). (2001)
20. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
21. Alviano, M., Faber, W., Leone, N., Perri, S., Pfeifer, G., Terracina, G.: The disjunctive Datalog system DLV. In de Moor, O., Gottlob, G., Furche, T., Sellers, A., eds.: Datalog Reloaded. Springer-Verlag (2011) 282–301