A system for interactive query answering with answer set programming

Martin Gebser, Philipp Obermeier, and Torsten Schaub[⋆]

Universität Potsdam, Institut für Informatik

Abstract. Reactive answer set programming has paved the way for incorporating online information into operative solving processes. Although this technology was originally devised for dealing with data streams in dynamic environments, like assisted living and cognitive robotics, it can likewise be used to incorporate facts, rules, or queries provided by a user. As a result, we present the design and implementation of a system for interactive query answering with reactive answer set programming. Our system *quontroller* is based on the reactive solver *oclingo* and implemented as a dedicated front-end. We describe its functionality and implementation, and we illustrate its features by some selected use cases.

1 Introduction

Traditional logic programming [1, 2] is based upon query answering. Unlike this, logic programs under the stable model semantics [3] are implemented by model generation based systems, viz. answer set solvers [4]. Although the latter also allows for checking whether a query is entailed by some stable model, there is so far no 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. 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.

We address this shortcoming and show how recently developed systems for *reactive* answer set programming (ASP) [11, 12] can be harnessed to provide query answering

[⋆] 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.

and theory exploration capacities. In fact, reactive 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 and cognitive robotics, it can likewise be used to incorporate facts, rules, or queries provided by a user. As a result, we present the design and implementation of a system for interactive query answering and theory exploration with ASP. Our system *quontroller*[1] is based on the reactive answer set solver *oclingo* and implemented as a dedicated front-end. We describe its functionality and implementation, and we illustrate its features on a running example.

2 Approach

In order to provide dedicated support for query answering, the *quontroller* encapsulates *oclingo* along with its basic front-end for entering online progressions. The basic idea is to condition the stable models of an underlying logic program via *query programs*, temporarily asserting atoms to be contained in stable models of interest. For circumventing restrictions due to the modularity requirement of reactive ASP (cf. [11, 12]) and enabling repeated assertions of an atom in a series of queries, the *quontroller* associates query programs with sequence numbers and exploits *oclingo*'s step counter to automatically map their contents. In the following, we detail this idea on the well-known example of n-coloring.

To begin with, Listing 1 provides an ASP encoding of n-coloring. The encoding applies to graphs represented by facts over predicate `edge`/2. Given such facts, the nodes of the graph are extracted in Line 6–7, each node is marked with exactly one of `n` colors in Line 10, and Line 11 forbids that connected nodes are marked with the same color. In Line 14–15, stable models are projected onto atoms over predicate `mark`/2.

Unlike with one-shot ASP solving, we do not combine the encoding in Listing 1 with fixed facts, but rather aim at a selective addition as well as withdrawal of atoms over `edge`/2. In order to prepare reactive ASP rules for this purpose, the instructions in Listing 2 can be fed into the *quontroller*. They are then mapped to the language of *oclingo* as shown in Listing 3. The resulting reactive ASP rules are divided into a static **`#base`** part (Line 1–13), a stepwise **`#cumulative`** part (Line 15–25), and a stepwise **`#volatile`** part (Line 27–34). The stepwise parts are instantiated for successive step numbers replacing the constant `t`, where instances of **`#cumulative`** rules are gathered over steps and **`#volatile`** ones are discarded when progressing to the next step.

In more detail, the **`#domain`** instructions in Line 2 and 3 of Listing 2 are mapped to the static rules in Line 4 and 7 of Listing 3. They define the (*quontroller*-internal) predicates `_domain_edge`/2 and `_domain_mark`/2, which provide the domains of instances of `edge`/2 and `mark`/2 that can be asserted by query programs. In particular, the rule in Line 4 expresses that edges may connect distinct nodes with labels running from `1` to `4`, and the rule in Line 7 declares that each node can be marked with the colors provided by `color`/1. Furthermore, the instruction in Line 4 of Listing 2 is mapped to the static choice rule in Line 10 of Listing 3. This rule compensates for the lack of facts over

[1] To be pronounced ‘cointreau’-ler; URL: `potassco.sourceforge.net/labs.html`.

```
% n colors
#const n = 3.
color(1..n).

% extract nodes from edges
node(X) :- edge(X,_).
node(X) :- edge(_,X).

% generate n-coloring
1 { mark(X,C) : color(C) } 1 :- node(X).
:- edge(X,Y), mark(X;Y,C).

% display n-coloring
#hide.
#show mark/2.
```

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

```
1  #setup.
2  #domain edge(X,Y) : X := 1..4, Y := 1..4, X != Y.
3  #domain mark(X,C) : X := 1..4, color(C).
4  #choose edge/2.
5  #define edge/2.
6  #query mark/2.
7  #show edge/2.
8  #endsetup.
```

Listing 2: *quontroller* setup instructions (`setup.ini`)

`edge`/2 and allows for instantiating the original encoding in Listing 1 relative to the domain given by `_domain_edge`/2. Again relying on `_domain_edge`/2, the instruction in Line 7 of Listing 2 is mapped to the **`#show`** statement in Line 13 of Listing 3 for including atoms over `edge`/2 (in addition to those over `mark`/2) in output projections.

The **`#cumulative`** and **`#volatile`** parts in Listing 3 deal with the instructions in Line 5 and 6 of Listing 2. In particular, the **`#external`** statements in Line 18 and 23 of Listing 3 declare (*quontroller*-internal) instances of `_assert_mark(X1,X2,t)` and `_assert_edge(X1,X2,t)` as potential online inputs from query programs, where `X1` and `X2` are instantiated relative to domains given by static rules and the constant `t` is added as an argument to distinguish separate queries. The rules in Line 19–20 and 24–25 of Listing 3 further define the (*quontroller*-internal) predicates `_derive_mark`/3 and `_derive_edge`/3 for indicating "active" assertions from query programs. Given that such assertions may remain active over several queries, the rules defining `_derive_mark`/3 and `_derive_edge`/3 include one case for reflecting current assertions (Line 19 and 24) and another for passing on former assertions (Line 20 and 25). As a consequence, instances of `_derive_mark(X1,X2,t)` and `_derive_edge(X1,X2,t)` capture active assertions regarding the original predicates

```
 1  #base.

 3  % #domain edge(X,Y) : X := 1..4, Y := 1..4, X != Y.
 4  _domain_edge(X,Y) :- X := 1..4, Y := 1..4, X != Y.

 6  % #domain mark(X,C) : X := 1..4, color(C).
 7  _domain_mark(X,C) :- X := 1..4, color(C).

 9  % #choose edge/2.
10  { edge(X1,X2) : _domain_edge(X1,X2) }.

12  % #show edge/2.
13  #show edge(X1,X2) : _domain_edge(X1,X2).

15  #cumulative t.

17  % #query mark/2.
18  #external _assert_mark(X1,X2,t) : _domain_mark(X1,X2).
19  _derive_mark(X1,X2,t) :- _domain_mark(X1,X2), _assert_mark(X1,X2,t).
20  _derive_mark(X1,X2,t) :- _domain_mark(X1,X2), _derive_mark(X1,X2,t-1).

22  % #define edge/2.
23  #external _assert_edge(X1,X2,t) : _domain_edge(X1,X2).
24  _derive_edge(X1,X2,t) :- _domain_edge(X1,X2), _assert_edge(X1,X2,t).
25  _derive_edge(X1,X2,t) :- _domain_edge(X1,X2), _derive_edge(X1,X2,t-1).

27  #volatile t.

29  % #query mark/2.
30  :- _domain_mark(X1,X2), _derive_mark(X1,X2,t), not mark(X1,X2).

32  % #define edge/2.
33  :- _domain_edge(X1,X2), _derive_edge(X1,X2,t), not edge(X1,X2).
34  :- _domain_edge(X1,X2), edge(X1,X2), not _derive_edge(X1,X2,t).
```

Listing 3: Mapping of *quontroller* setup instructions in `setup.ini` to reactive ASP rules

`mark`/2 and `edge`/2, and corresponding matches are established via **`#volatile`** integrity constraints. For one, the **`#query`** instruction in Line 6 of Listing 2 is mapped to the integrity constraint in Line 30 of Listing 3, thus requiring `mark(X1,X2)` to hold at any step where `_derive_mark(X1,X2,t)` indicates an active assertion, and the analogous integrity constraint in Line 33 is obtained in view of the **`#define`** instruction in Line 5 of Listing 2. The latter is complemented by another integrity constraint in Line 34, denying `edge(X1,X2)` to hold when `_derive_edge(X1,X2,t)` does not indicate any active assertion. That is, a **`#query`** instruction expresses that assertions may require atoms to belong to stable models of interest, and a **`#define`** instruction is stronger by additionally claiming some active assertion for atoms to hold.

After launching *oclingo* with the encoding in Listing 1 and the reactive ASP rules in Listing 3 (via '`quontroller.py -o encoding.lp -c setup.ini`'), the *quontroller* is ready to process query programs provided by a user. An exemplary stream of query programs is shown in Figure 1(a), and Figure 1(b) provides its counterpart in the syntax of *oclingo*'s basic front-end. In fact, the *quontroller* maps query programs to available stream constructs and automatically performs replacements for interacting with reactive ASP rules. To begin with, the keywords '**`#query`**`.`' and '**`#endquery`**`.`', which encapsulate individual query programs, are mapped to '**`#step`** `q : 0.` **`#forget`** `q-1.`' and '**`#endstep`**`.`', where `q` is the sequence number

```
 1  #query.
 2  #assert : e(1).
 3  edge(1,2).
 4  edge(1,3).
 5  edge(2,3).
 6  edge(2,4).
 7  edge(3,4).
 8  #endquery.

10  #query.
11  #assert.
12  mark(1,1).
13  #endquery.

15  #query.
16  #assert : e(2).
17  edge(1,4).
18  #endquery.

20  #query.
21  #retract : e(2).
22  #assert.
23  mark(1,1).
24  mark(2,2).
25  #endquery.

27  #query.
28  #retract : e(1).
29  #endquery.

31  #stop.
```

(a) *quontroller* query stream

```
 1  #step 1 : 0.  #forget 0.
 2  #assert : e(1).
 3  _assert_edge(1,2,1).
 4  _assert_edge(1,3,1).
 5  _assert_edge(2,3,1).
 6  _assert_edge(2,4,1).
 7  _assert_edge(3,4,1).
 8  #endstep.

10  #step 2 : 0.  #forget 1.
11  #volatile : 1.
12  _assert_mark(1,1,2).
13  #endstep.

15  #step 3 : 0.  #forget 2.
16  #assert : e(2).
17  _assert_edge(1,4,3).
18  #endstep.

20  #step 4 : 0.  #forget 3.
21  #retract : e(2).
22  #volatile : 1.
23  _assert_mark(1,1,4).
24  _assert_mark(2,2,4).
25  #endstep.

27  #step 5 : 0.  #forget 4.
28  #retract : e(1).
29  #endstep.

31  #stop.
```

(b) Mapping of *quontroller* query stream in (a)

Fig. 1: A *quontroller* query stream and its mapping to a reactive ASP online progression

of a query program and the **`#forget`** directive enables simplifications of reactive ASP rules for yet undefined **`#external`** atoms introduced at step `q-1`. Also note that '`: 0`' in **`#step`** directives tells *oclingo* not to increment the step counter on unsatisfiability.

Each query program may include labeled assertions, as declared via '**`#assert`** `: e(1).`' in Line 2 of Figure 1(a) and 1(b). Such a construct expresses that subsequently provided rules remain active until the labeled assertion is explicitly retracted. In view of its reactive ASP rules, the *quontroller* however replaces each head atom `p(...)` of a rule (or fact) to assert by its internal representation `_assert_p(...,q)`, where `q` is again the sequence number of the query program at hand. For instance, '`edge(1,2).`' is mapped to '`_assert_edge(1,2,1).`' in Line 3 of Figure 1(a) and 1(b). By means of reactive ASP rules capturing the **`#define`** instruction in Line 5 of Listing 2, the instances of `_assert_edge`/3 provided by facts in Line 3–7 of Figure 1(b) are matched with the original atoms over `edge`/2. As a consequence, the first query program yields six stable models, in which the unconnected nodes `1` and `4` share one of the colors `1`, `2`, or `3` and the nodes `2` and `3` are marked with distinct remaining colors.

The second query program in Line 10–13 of Figure 1(a) includes '`mark(1,1).`' as an unlabeled assertion, indicated by the keyword '**`#assert.`**' The latter is mapped to the stream construct '**`#volatile`** `: 1.`' (cf. Line 11 of Figure 1(b)), meaning that the internal representation '`_assert_mark(1,1,2).`' of the assertion expires "automatically" in the next step. Technically, such expiration is implemented by adding assumption literals to the bodies of transient rules, i.e. '`_assert_mark(1,1,2).`' is internally turned into '`_assert_mark(1,1,2) :- _expire(3).`' and `_expire(3)` holds up to step `3` where it is then permanently falsified. However, in the second step, the assertion of color `1` for node `1` leads to two stable models of interest among the six obtained in the first step. Also note that the *quontroller* language includes '**`#assert.`**' in order to indicate the beginning of query parts in which head atoms are replaced by internal representations, so that any rules to be left untouched can still be provided beforehand.

Summarizing the remaining query programs, the labeled assertion `e(2)` in the third query program turns the graph represented by atoms over `edge`/2 into a clique of four nodes, so that no stable model is obtained in the third step. Hence, `e(2)` is retracted in the fourth step (by discharging an assumption literal associated with `e(2)`), and the additional unlabeled assertion of colors for the nodes `1` and `2` leads to a single stable model of interest. Note that '`mark(1,1).`' is turned into '`_assert_mark(1,1,4).`' in Line 23 of Figure 1(b), while '`_assert_mark(1,1,2).`' has been used in Line 12. The rewriting by the *quontroller* thus avoids a clash with *oclingo*'s modularity requirement and enables repeated assertions of the "same" atom (in separate query programs). Finally, the empty (projection of a) stable model is obtained after retracting all instances of `_assert_edge`/3 in the last query program, and '**`#stop.`**' afterwards signals the end of the query stream to the *quontroller*.

3 Discussion

We presented a simple yet effective extension of reactive ASP that allows for interactive query answering and theory exploration with ASP. This was accomplished by means of a mapping scheme between queries and reactive ASP rules along with the assumption-based solving capacities of *oclingo*. With it, programs can be temporarily added to the solving process, either for an initially limited number of interactions or until they are interactively withdrawn again. A typical use case of limited program parts are integrity constraints, representing queries automatically vanishing after having been posed. Unlike this, an assertion allows, for instance, for exploring the underlying domain under user-defined hypotheses. All subsequent solving processes then include the asserted information until it is retracted by the user. The possibility of reusing ground rules as well as recorded conflict information over a sequence of queries distinguishes reactive ASP from ordinary one-shot reasoning methods. As future work, we want to study the performance of query answering with the *quontroller* on challenging benchmark problems.

Acknowledgments This work was partially funded by DFG grant SCHA 550/9-1. We are grateful to the anonymous reviewers for their suggestions.

References

1. Clocksin, W., Mellish, C.: Programming in Prolog. Springer (1981)
2. Lloyd, J.: Foundations of Logic Programming. Springer (1987)
3. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. Proc. ICLP, MIT (1988) 1070–1080
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool (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. Proc. ISMB, (2004) 15–22
6. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. Proc. AAAI, AAAI (2008) 436–441
7. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. TPLP Journal **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. Proc. KR, AAAI (2010) 497–507
9. Ray, O., Whelan, K., King, R.: Logic-based steady-state analysis and revision of metabolic networks with inhibition. Proc. CISIS, IEEE (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. Proc. CMSB, Springer (2012) 342–361
11. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. Proc. LPNMR, Springer (2011) 54–66
12. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Preliminary report. Proc. KR, AAAI (2012) 613–617