A Tool for Advanced Correspondence Checking in Answer-Set Programming: Preliminary Experimental Results*

Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran

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

1 Introduction

The class of nonmonotonic logic programs under the answer-set semantics [5], with which we are dealing with in this paper, represents the canonical and, due to the availability of efficient answer-set solvers, arguably most widely used approach to answer-set programming (ASP). The latter is based on the idea that problems are encoded in terms of theories such that the solutions of a given problem are determined by the models ("answer sets") of the corresponding theory.

In previous work [4], a general framework for specifying correspondences between logic programs under the answer-set semantics has been introduced. Hereby, the correspondence of two programs is determined in terms of a class $\mathcal{C}$ of *context programs* and a comparison relation ρ such that correspondence between two programs, P and Q, holds iff the answer sets of $P \cup R$ and $Q \cup R$ satisfy ρ, for any program $R \in \mathcal{C}$. The framework includes, as special instances, the well-known notions of *strong equivalence* [8], *uniform equivalence* [3], and the practicably important case of program comparison under *projected* answer sets.

For the case of propositional disjunctive logic programs, correspondence checking in the above framework under projected answer sets is surprisingly hard, viz. Π_4^P-complete in general [4], i.e., lying on the fourth level of the polynomial hierarchy. For computing such program correspondences, efficient reductions to *quantified propositional logic* have been developed [11]. More specifically, two kinds of reductions, $\mathcal{S}[\cdot]$ and $\mathcal{T}[\cdot]$, are introduced, where $\mathcal{T}[\cdot]$ can be seen as an explicit optimization of $\mathcal{S}[\cdot]$. In this paper, we report about an implementation of these reductions, which we refer to as the system `eqcheck`, and about initial experimental results.

We recall that quantified propositional logic is an extension of classical propositional logic characterised by the condition that its sentences, usually referred to as *quantified Boolean formulas* (QBFs), are permitted to contain quantifications over atomic formulas.

The rationale to consider a reduction approach to QBFs is twofold: (i) complexity results about quantified propositional logic imply that decision problems from the polynomial hierarchy can be efficiently represented in terms of QBFs, and (ii) several practicably efficient solvers for quantified propositional logic are currently available. Indeed, `eqcheck` uses such solvers as back-end inference engines.

2 System Description

The tool `eqcheck` [9] implements linear-time reductions from inclusion problems $(P, Q, \mathcal{P}_A, \subseteq_B)$ and equivalence problems $(P, Q, \mathcal{P}_A, =_B)$ to corresponding QBFs. It takes as input (i) two programs, say P and Q, as well as (ii) two sets of atoms, say A and B, where A specifies the context and B the set of atoms for projection on the correspondence relation. The reduction ($\mathcal{S}[\cdot]$ or $\mathcal{T}[\cdot]$) and the type of correspondence problem ($\subseteq_B$, $=_B$) are specified via command line arguments: `-T, -S` to select the kind of reduction, `-e, -i` to check for program equivalence or program inclusion.

In general the syntax to specify answer-set programs for `eqcheck` corresponds to the basic `DLV` [**?**] syntax. Propositional `DLV` programs can be passed to `eqcheck` and programs processible for `eqcheck` can be handled by DLV. As usual white-spaces are skipped during the parsing procedure. Rules, facts, and

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

constrains are separated by dots. The symbol `v` used to indicate disjunctions is expressed with a lower case `v`. The symbol $\leftarrow$ is represented with the two characters `:-`. Default negation *not* is expressed with the reserved word `not`. Atoms start with a lower case character followed by an arbitrary number of lower case characters, upper case characters, numbers or the underline character. To enrich expressibility even grounded predicates can be expressed as an atom followed by a list of atoms within parentheses. What follows is a distinct description of how to apply the tool.

Considering the running example the two programs would be expressed in our syntax as follows:

```
sel(b) :- b, not out(b).
sel(a) :- a, not out(a).
out(a) v out(b) :- a, b.
```

for program P and

```
sel(a) v sel(b) :- a.
sel(a) v sel(b) :- b.
fail :- sel(a), not a, not fail.
fail :- sel(b), not b, not fail.
```

for program Q. We suppose that file `P.dl` contains program P and file `Q.dl` contains program Q.

Now we want to check if P is equivalent to Q w.r.t. projection on the output predicate sel. We also want to restrict the context to programs over $\{a, b\}$. Therefore we need to specify a context set

```
(a, b)
```

stored in file `A` and a projection set

```
(sel(a), sel(b))
```

stored in file `B`. Now invocation syntax would be:

```
eqcheck -e P.dl Q.dl A B
```

By default the encoding $\mathcal{T}[\cdot]$ is chosen. Notice that the order of the arguments is important: first the programs, then the context set, and at last the projection set. An alternative syntax would have been:

```
eqcheck -e -A "(a, b)" -B "(sel(a), sel(b))" P.dl Q.dl
```

,i.e., specifying A and B without creating extra files for them. For the complete syntax invoke `eqcheck`with option `-h`.

After invocation the resulting QBF is written to the standard output device and can be processed further by QBF-solvers. The output could be piped, e.g., directly to the BDD-based non-normalform QBF-solver boole [10]:

```
eqcheck -e P.dl Q.dl A B | boole
```

which yields `0` or `1` as answer for the correspondence problem (in our case the correspondence holds and the output is `1`).

If the set A (resp. B) is omitted in invocation, then the whole universe (set of all variables that occur in program P or Q) is assumed for the set A (resp. B). If `0` is passed instead of a filename, then the empty set is assumed for set A (resp. B). Thus checking for strong equivalence or ordinary equivalence (with or without projection) as special cases of the general framework can be specified easily with `eqcheck`.

We developed `eqcheck` entirely in *ANSI C*; hence, it is highly portable. The parser for the input data was written using *LEX* and *YACC*. The complete package in its current version consists of more than 2000 lines of code. For further information, cf. [9].

3 Experimental Results

Our experiments were conducted to determine the behaviour of different QBF-solvers in combination with different prenexing strategies (whenever prenexing is necessary). To this end, we implemented a generator, `eqtest`, providing correspondence problems (as detailed above) which emanate from the proof of the Π_4^P-hardness of checking such problems, in order to have a class of benchmark problems capturing the intrinsic complexity of correspondence checking.

The strategy to generate a test case is to (i) generate a QBF Φ by random which is hard for the at most fourth level of the polynomial hierarchy; (ii) reduce Φ to an inclusion problems $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ according to the complexity result such that Π holds iff Φ is valid; iii) apply eqcheck to generate a corresponding QBF Ψ out of Π; (iv) evaluate Ψ; (v) and check if Ψ is equivalent to Φ.

Notice that step (iv) is used to compare the different QBF-solvers whereas step (v) is essential to validate `eqcheck`. Concerning step (ii) the reduction from the QBF to the corresponding inclusion problem is obtained as follows:

Let $\Phi = \exists X \forall Y \exists Z \phi$ a QBF with $\phi = \bigwedge_{i=1}^{n} C_i$ a CNF over $V = (W \cup X \cup Y \cup Z)$ and $C_i = c_{i,1} \vee \cdots \vee c_{i,k_i}$; $\bar{V} = \{\bar{v} \mid v \in V\}$ new atoms; $C_i^* = c_{i,1}^*, \ldots, c_{i,k_i}^*$, $v^* = \bar{v}$, and $(\neg v)^* = v$. We define P as:

$$\begin{aligned} P = \{&v \vee \bar{v} \leftarrow \mid v \in V\} \cup \\ &\{v \leftarrow u, \bar{u};\ \bar{v} \leftarrow u, \bar{u} \mid v, u \in V \backslash W\} \cup \\ &\{\leftarrow \mathit{not}\, v;\ \leftarrow \mathit{not}\, \bar{v} \mid v \in V \backslash W\} \cup \\ &\{v \leftarrow C_i^*;\ \bar{v} \leftarrow C_i^* \mid v \in V \backslash W;\ 1 \leq i \leq n\}. \end{aligned}$$

For Q we use further atoms $X' = \{x' \mid x \in X\}$, $\bar{X}' = \{\bar{x}' \mid x \in X\}$ and define:

$$\begin{aligned} Q = \{&v \vee \bar{v} \leftarrow \mid v \in X \cup Y\} \cup \\ &\{v \leftarrow u, \bar{u};\ \bar{v} \leftarrow u, \bar{u} \mid v, u \in X \cup Y\} \cup \\ &\{\leftarrow x', \bar{x}';\ \leftarrow \mathit{not}\, x', \mathit{not}\, \bar{x}' \mid x \in X\} \cup \\ &\{v \leftarrow x';\ \bar{v} \leftarrow x';\ v \leftarrow \bar{x}';\ \bar{v} \leftarrow \bar{x}' \mid v \in X \cup Y, x \in X\} \cup \\ &\{x' \leftarrow \bar{x}, \mathit{not}\, \bar{x}';\ \bar{x}' \leftarrow x, \mathit{not}\, x' \mid x \in X\}. \end{aligned}$$

Set A and B are defined as:

$$A = B = \{X \cup \bar{X} \cup Y \cup \bar{Y}\}.$$

For further information on `eqtest`, cf. [9].

We have set up a test series comprising 1000 instances of inclusion problems (465 of them evaluating to true), where the first program P has 620 rules, the second program Q has 280 rules, using a total of 40 atoms, and the sets A, B of atoms are chosen such that $A = B$, containing 16 atoms. After employing `eqcheck`, the resulting QBFs possess, in case of translation $\mathcal{S}$, 200 atoms and, in case of translation $\mathcal{T}$, 152 atoms. The additional prenexing step (together with a translation of the propositional part into conjunctive normal form) yields, in case of $\mathcal{S}$, QBFs with 6575 clauses over 2851 atoms and, in case of $\mathcal{T}$, QBFs with 6216 clauses over 2555 atoms.

The prenexing step in normal form transformation is not a deterministic process. Although certain dependencies have to be respected, when combining the quantifiers of different subformulas to one linear prefix, the arrangements can be done in different manners. Consider for example the formula $\forall w(\exists x \forall y \exists z \phi \wedge \exists a \psi)$. Then there are two ways to construct the prefix without changing the truth value of the formula: $\forall w \exists x \exists a \forall y \exists z$ and $\forall w \exists x \forall y \exists z \exists a$. Obviously with the growth of the formula size, the number of possible arrangements increases. For a more detailed discussion of this issue see [2]. This indeterminism has severe consequences: the impact of the chosen shifting strategy has an enormous impact on the running time of the solvers (again see [2]). The structure of the formulas w.r.t. to the quantifiers in our benchmark set is very similar to the formula's structure in the example above. Since this structure is very simple, only two different prenexing strategies can be applied (UP and DOWN).

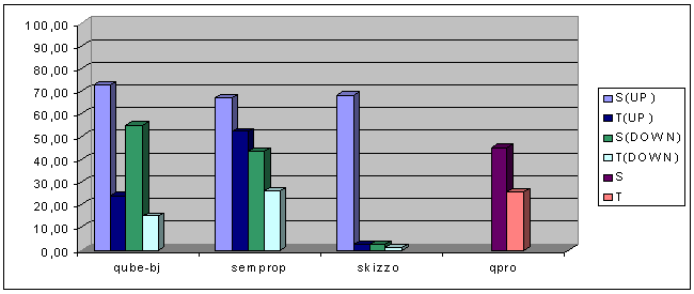

Fig. 1. Results for true problem instances subdivided by solvers, encodings $\mathcal{S}$,$\mathcal{T}$, and shifting strategies DOWN, UP.

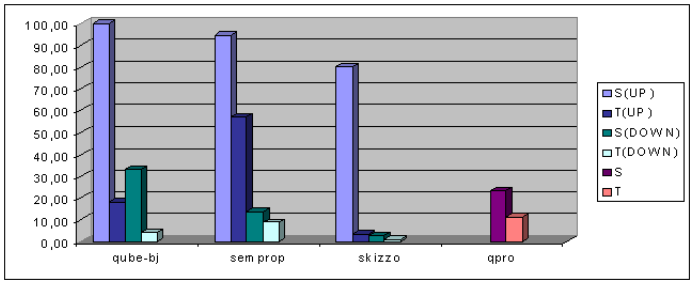

Fig. 2. Results for false problem instances subdivided by solvers, encodings $\mathcal{S}$,$\mathcal{T}$, and shifting strategies DOWN, UP.

We compared four QBF-solvers, viz. `qube` [6], `semprop` [7], `skizzo` [1], and `qpro`. The former three require input in prenex form (thus, we test them using both strategies DOWN and UP), and `qpro` is a new solver, currently under development at our department, which admits arbitrary QBFs as input.

Our results are depicted in Figures 1 and 2, referring to the true and false instances of our series, respectively. The y-axis shows the (arithmetically) average running time in seconds for each solver (with respect to the chosen translation and prenexing strategy). We set a time-out of 100 seconds.

As expected, for all solvers, the more compact encoding $\mathcal{T}$ was evaluated faster than the QBFs stemming from $\mathcal{S}$. The performance of the normal-form solvers `qube`, `semprop`, and `skizzo` is highly dependent on the shifting strategy. For our test set, DOWN dominates UP. Moreover, analysing the results for `qpro`, compared to the other solvers, there is an indication that the normal-form approach of QBF evaluation is not particularly appropriate for finding simplifications in formulas, which is an interesting issue for future work.

References

1. M. Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *Proc. CADE-05*, 2005.
2. U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In *Proc. SAT-03. Selected Revised Papers*, volume 2919 of *LNCS*, pages 214–228, 2004.
3. T. Eiter and M. Fink. Uniform Equivalence of Logic Programs under the Stable Model Semantics. In *Proc. ICLP-03*, number 2916 in LNCS, pages 224–238, 2003.
4. T. Eiter, H. Tompits, and S. Woltran. On Solution Correspondences in Answer Set Programming. In *Proc. IJCAI-05*, pages 97–102, 2005.
5. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
6. E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. *Artificial Intelligence*, 145:99–120, 2003.
7. R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Proc. TABLEAUX 2002*, volume 2381 of *LNCS*, pages 160–175, 2002.
8. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

9. J. Oetsch. eqcheck homepage. http://www.kr.tuwien.ac.at/research/eq/.
10. boole homepage. http://www.cs.cmu.edu/~modelcheck/bdd.html.
11. H. Tompits and S. Woltran. Towards Implementations for Advanced Equivalence Checking in Answer-Set Programming. In *Proc. ICLP-05*, volume 3668 of *LNCS*, pages 189–203, 2005.