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

Abstract. In recent work, a general framework for specifying program correspondences under the answer-set semantics has been defined. The framework allows to define different notions of equivalence, including the well-known notions of *strong* and *uniform equivalence*, as well as refined equivalence notions based on the *projection* of answer sets, where not all parts of an answer set are of relevance (like, e.g., removal of auxiliary letters). In the general case, deciding the correspondence of two programs lies on the fourth level of the polynomial hierarchy and therefore this task can (presumably) not be efficiently reduced to answer-set programming. In this paper, we give an overview about an implementation to compute program correspondences in this general framework. The system, called `eqcheck`, relies on linear-time constructible reductions to *quantified propositional logic* using extant solvers for the latter language as back-end inference engines. We provide some preliminary performance evaluation, which shed light on some crucial design issues.

1 Introduction

The class of nonmonotonic logic programs under the answer-set semantics [6], 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 paradigm 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. Logic programming under the answer-set semantics has become an important host for solving many AI problems, including planning, diagnosis, information integration, and inheritance reasoning (cf. [5] for an overview).

To support engineering tasks of ASP solutions, an important issue is to determine the equivalence of different problem encodings. To this end, various notions of equivalence between programs under the answer-set semantics have been studied in the literature, including the recently proposed framework by Eiter *et al.* [4], which subsumes most of the previously introduced notions. Within this framework, correspondence between two programs, P and Q, holds iff the answer sets of $P \cup R$ and $Q \cup R$ satisfy certain specified criteria, for any program R in a specified class, called the *context*. We shall focus here on correspondence problems where both the context and the comparison between answer sets is specified by alphabets. Note that this kind of program correspondence includes, as special instances, the well-known notions of *strong equivalence* [10], *uniform equivalence* [3], and the practicably important case of program comparison under *projected* answer sets.

For illustration, consider the following two programs which both express the selection of at most one of the atoms a, b, but an atom is only selected if it can be derived together with the context:

$$P = \{\ sel(b) \leftarrow b, not\ \ out(b);\quad sel(a) \leftarrow a, not\ \ out(a);\quad out(a) \vee out(b) \leftarrow a, b\ \}.$$

$$Q = \{\ fail \leftarrow sel(a), not\, a, not\, fail;\quad fail \leftarrow sel(b), not\, b, not\, fail;\quad sel(a) \vee sel(b) \leftarrow a;\quad sel(a) \vee sel(b) \leftarrow b\ \}.$$

Both programs use "local" atoms, $out(\cdot)$ and $fail$, respectively, which are expected not to appear in the context. In order to compare the programs, we could specify an alphabet A for the context, for instance,

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

$A = \{a, b\}$ or, more generally, any set A of atoms not containing the local atoms $out(a)$, $out(b)$, and $fail$. On the other hand, we want to check whether, for each addition of a context program over A, the answer sets correspond when taking only atoms from $B = \{sel(a),\ sel(b)\}$ into account.

In this paper, we report about an implementation of such correspondence problems. The system is called `eqcheck`, and we also present some initial experimental results. The overall approach of this implementation is to reduce the problem of correspondence checking to the satisfiability problem of *quantified propositional logic*, 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 motivation to use such a reduction approach is as follows: First, complexity results [4] show that correspondence checking within this framework is hard, lying on the fourth level of the polynomial hierarchy. This indicates that implementations of such checks cannot be straightforwardly realised using ASP-systems themselves. In turn, it is well known that decision problems from the polynomial hierarchy can be efficiently represented in terms of QBFs in such a way that determining the validity of the resultant QBFs is not computationally harder than checking the original problem. In previous work [11], such translations from correspondence checking to QBFs being constructible in *linear time and space* have been developed. Second, various practicably efficient solvers for quantified propositional logic are currently available (see, e.g., [8]). Hence, such tools are used as back-end inference engines in our system to compute the correspondence problems under consideration.

2 Theoretical Background

We deal here with propositional disjunctive logic programs, which are finite sets of rules of form

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, not\; a_{m+1}, \ldots, not\; a_n,$$

$n \geq m \geq l \geq 0$, where all a_i are propositional atoms and not denotes default negation. If all atoms occurring in a program P are from a given set A of atoms, we say that P is a program *over* A. The set of all programs over A is denoted by $\mathcal{P}_A$.

Following Gelfond and Lifschitz [6], an interpretation I, i.e., a set of atoms, is an *answer set* of a program P iff it is a minimal model of the *reduct* P^I, resulting from P by

(i) deleting all rules containing default negated atoms $not\; a$ such that $a \in I$, and
(ii) deleting all default negated atoms in the remaining rules.

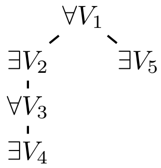
The set of all answer sets of a program P is denoted by $\mathcal{AS}(P)$.

For a set of atoms B, and families $\mathcal{S}$, $\mathcal{S}'$ of interpretations, we write $\mathcal{S}|_B = \{Y \cap B \mid Y \in \mathcal{S}\}$, and we define projections of the standard subset and set-equality relations as follows: $\mathcal{S} \subseteq_B \mathcal{S}'$ iff $\mathcal{S}|_B \subseteq \mathcal{S}'|_B$, and $\mathcal{S} =_B \mathcal{S}'$ iff $\mathcal{S}|_B = \mathcal{S}'|_B$.

Following Eiter *et al.* [4], given sets of atoms A and B, we consider *inclusion problems* $(P, Q, \mathcal{P}_A, \subseteq_B)$ and *equivalence problems* $(P, Q, \mathcal{P}_A, =_B)$. Formally, we say that $(P, Q, \mathcal{P}_A, \subseteq_B)$ holds iff, for each $R \in \mathcal{P}_A$, $\mathcal{AS}(P \cup R) \subseteq_B \mathcal{AS}(Q \cup R)$. As well, $(P, Q, \mathcal{P}_A, =_B)$ holds iff, for each $R \in \mathcal{P}_A$, $\mathcal{AS}(P \cup R) =_B \mathcal{AS}(Q \cup R)$. Note that $(P, Q, \mathcal{P}_A, =_B)$ thus holds iff $(P, Q, \mathcal{P}_A, \subseteq_B)$ and $(Q, P, \mathcal{P}_A, \subseteq_B)$ jointly hold.

In [4], the concept of a *spoiler* to an inclusion problem has been defined, having the property that a spoiler exists iff the respective inclusion problem does *not* hold. As shown in [11], the conditions of deciding whether a spoiler of an inclusion problem exists can efficiently be represented in terms of quantified propositional logic such that, given an inclusion problem, the resulting QBF is true iff no spoiler to that inclusion problem exists. As mentioned above, having a procedure to decide inclusion problems, we can decide the respective equivalence problems as well.

We consider two different reductions from inclusion problems to QBFs, $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$, where $\mathsf{T}[\cdot]$ can be seen as an explicit optimization of $\mathsf{S}[\cdot]$ (see [11] for the details about the two translations). However, both reductions yield (in the worst case) QBFs of *depth* 4, i.e., where the maximal number of alternations of quantifiers in any path of the formula tree is 3. In fact, the formula trees for the encodings yield the following common quantifier dependencies [2], which can be illustrated as follows (where $V_1, \ldots, V_5$ denote pairwise disjunct sets of atoms comprising all atoms occurring in each translation):



Most of the available QBF-solvers require the input QBF to be in *prenex conjunctive normal form* (CNF), i.e., where the quantifiers are separately placed in a prefix and the remaining propositional formula is in conjunctive normal form. Thus, given an arbitrary QBF, in order to apply a solver requiring input in such form, one has to (i) "linearize" the quantifiers and (ii) transform the propositional part into CNF (which is usually done by introducing new atoms to avoid an exponential blow-up of the formula size).

Step (i) 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. Obviously, with growing formula size, the number of possible arrangements increases [2]. Inspecting the quantifier dependencies of our encodings, as illustrated by the above diagram, we can group $\exists V_5$ either together with $\exists V_2$ or with $\exists V_4$. This yields the following two basic ways for prenexing our encodings:

$$\uparrow\!: \ \forall V_1 \exists V_2 \exists V_5 \forall V_3 \exists V_4 \quad \text{and} \quad \downarrow\!: \ \forall V_1 \exists V_2 \forall V_3 \exists V_4 \exists V_5.$$

Both variants do not increase the depth of the encoding. Together with the two available encodings $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$, we thus get four different alternatives to represent an inclusion problem in terms of a prenex QBF; we will denote them in what follows by $\mathsf{S}_{\uparrow}[\cdot]$, $\mathsf{S}_{\downarrow}[\cdot]$, $\mathsf{T}_{\uparrow}[\cdot]$, and $\mathsf{T}_{\downarrow}[\cdot]$, respectively. Our system allows to select between these different options and our experiments below give some evidence which one of them has the best performance (with respect to the employed QBF-solver and the employed benchmarks).

3 System Description

The tool `eqcheck` implements the 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, P and Q, and (ii) two sets of atoms, A and B, where A specifies the alphabet of the context and B the set of atoms for projection on the correspondence relation. The reduction ($\mathsf{S}[\cdot]$ or $\mathsf{T}[\cdot]$) and the type of correspondence problem ($\subseteq_B$ or $=_B$) are specified via command-line arguments: `-S`, `-T` to select the kind of reduction; and `-i`, `-e` to check for inclusion or equivalence between the two programs.

In general, the syntax to specify the programs in `eqcheck` corresponds to the basic `DLV` syntax.[1] Propositional `DLV` programs can be passed to `eqcheck` and programs processible for `eqcheck` can be handled by `DLV`. As usual, blanks are skipped during the parsing procedure, and rules are separated by dots. Disjunctions in rule heads are expressed with a lower case "`v`". The symbol "$\leftarrow$" in rules 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. Also grounded predicates constitute a valid input for `eqcheck` and can be expressed as an atom followed by a list of atoms within parentheses.

Considering the example from the introduction, the two programs would be expressed in our syntax as follows:

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

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

We suppose that file `P.dl` contains the code for program P and, accordingly, file `Q.dl` contains the code for program Q. If we want to check whether P is equivalent to Q with respect to the projection to the output predicate $sel(\cdot)$, and restricting the context to programs over $\{a, b\}$, then we need to specify

[1] See `http://www.dlvsystem.com/` for details about `DLV`.

– the context set, stored in a file, say `A`, containing “`(a, b)`”, and
– the projection set, also stored in a file, say `B`, containing “`(sel(a), sel(b))`”.

The invocation syntax for `eqcheck` then is as follows:

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

By default, the encoding $\mathsf{T}[\cdot]$ is chosen. Note that the order of the arguments is important: first, the programs P and Q appear, then the context set A, and lastly the projection set B. An alternative call of `eqcheck` for the above example would be

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

specifying A and B directly from the command line. The complete syntax of `eqcheck` can be seen by invoking it 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 can be piped, e.g., directly to the BDD-based non-normal form QBF-solver `boole`,[2] by means of the command

```
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`). To employ further QBF-solvers, the output has to be processed according to the input syntax of the considered solver.

If the set A (resp., B) is omitted in invocation, then the set of all variables that occur in program P or Q is assumed for 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 between P and Q is done by

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

while ordinary equivalence (with projection over B) is done by

```
eqcheck -e P.dl Q.dl 0 B | boole.
```

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 about `eqcheck` and the benchmark generator discussed below, see

```
http://www.kr.tuwien.ac.at/research/eq/.
```

4 Experimental Results

Our experiments were conducted to determine the behaviour of different QBF-solvers in combination with the encodings $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$ for inclusion checking, or, if the employed QBF-solver requires the input in prenex form, with $\mathsf{S}_{\uparrow}[\cdot]$, $\mathsf{S}_{\downarrow}[\cdot]$, $\mathsf{T}_{\uparrow}[\cdot]$, and $\mathsf{T}_{\downarrow}[\cdot]$. To this end, we implemented a generator, `eqtest`, providing a suite of inclusion problems, which emanate from the proof of the Π_4^P-hardness of inclusion checking [4], in order to have a class of benchmark problems capturing the intrinsic complexity of this task.

The strategy to generate a test case is as follows:

1. generate a QBF Φ by random having three quantifier alternations (the evaluation problem of such QBFs lies at the fourth level of the polynomial hierarchy);
2. reduce Φ to an inclusion problems $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ such that Π holds iff Φ is valid, corresponding to the reduction used in the above mentioned complexity proof of inclusion checking;
3. apply `eqcheck` to generate a corresponding QBF Ψ from Π; and finally
4. evaluate Ψ.

[2] Available at `http://www.cs.cmu.edu/~modelcheck/bdd.html`.

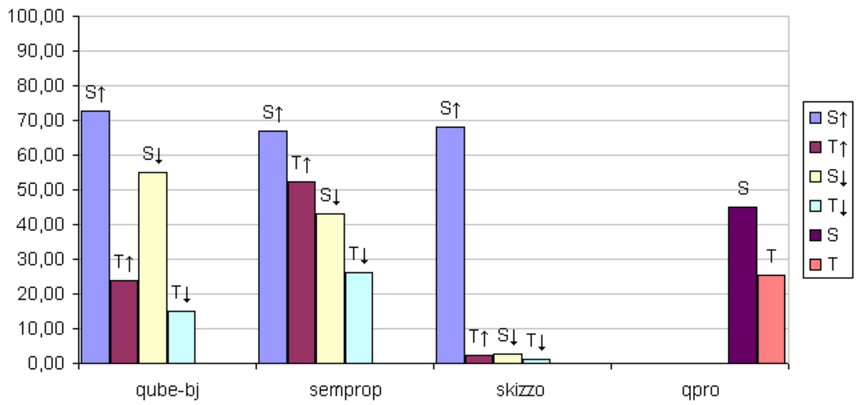


Fig. 1. Results for true problem instances subdivided by solvers and encodings.

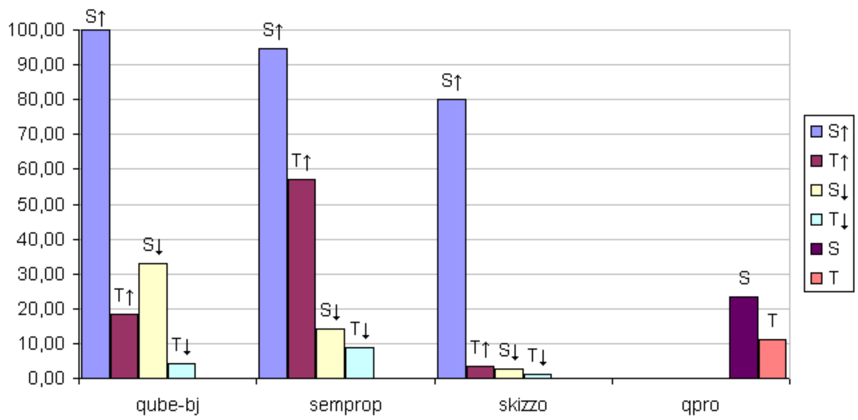


Fig. 2. Results for false problem instances subdivided by solvers and encodings.

Incidentally, this procedure yields also a simple method for validating the correctness of the overall implementation, by simply checking whether Ψ is equivalent to Φ.

We have set up a test series comprising 1000 instances of inclusion problems generated that way (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 and B of atoms are chosen to contain 16 atoms. After employing `eqcheck`, the resulting QBFs possess, in case of translation $\mathsf{S}[\cdot]$, 200 atoms and, in case of translation $\mathsf{T}[\cdot]$, 152 atoms. The additional prenexing step (together with the translation of the propositional part into CNF) yields, in case of $\mathsf{S}[\cdot]$, QBFs with 6575 clauses over 2851 atoms and, in case of $\mathsf{T}[\cdot]$, QBFs with 6216 clauses over 2555 atoms.

We compared four QBF-solvers, viz. `qube-bj` [7], `semprop` [9], `skizzo` [1], and `qpro`. The former three require prenex CNF formulas as input (thus, we tested them using encodings $\mathsf{S}_{\uparrow}[\cdot]$, $\mathsf{S}_{\downarrow}[\cdot]$, $\mathsf{T}_{\uparrow}[\cdot]$, and $\mathsf{T}_{\downarrow}[\cdot]$), and `qpro` is a new solver, currently under development at our department, which admits arbitrary QBFs as input (we tested it with the non-prenex encodings $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$).

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 encodings of form T were evaluated faster than the QBFs stemming from encodings of form S. The performance of the prenex-form solvers `qube-bj`, `semprop`, and `skizzo` is highly dependent on the shifting strategy. For our test set, ↓ dominates ↑. 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*, volume 3632 of *LNCS*, pages 369–376, 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 N. Leone. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.
6. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
7. E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. *Artificial Intelligence*, 145:99–120, 2003.
8. D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF Arena: the SAT'03 Evaluation of QBF Solvers. In *Proc. SAT-03. Selected Revised Papers*, volume 2919 of *LNCS*, pages 468–485, 2004.
9. 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.
10. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
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.