cc$\top$: A Tool for Checking Advanced Correspondence Problems in Answer-Set Programming[*]

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

Martina Seidl
Institut für Softwaretechnik 188/3,
Technische Universität Wien,
Favoritenstraße 9-11,
A-1040 Vienna, Austria
seidl@big.tuwien.ac.at

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

Abstract

In recent work, a general framework for specifying correspondences between logic programs under the answer-set semantics has been defined. The framework allows to define different notions of equivalence, including well-known notions like strong equivalence as well as refined ones based on the projection of answer sets, where not all parts of an answer set are of relevance. In this paper, we describe a system, called cc$\top$, to verify program correspondences in this general framework, relying on linear-time constructible reductions to quantified propositional logic using extant solvers for the latter language as back-end inference engines. We provide a preliminary performance evaluation which sheds light on some crucial design issues.

1. Introduction

Nonmonotonic logic programs under the answer-set semantics [12] represent 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, and inheritance reasoning (see, e.g., Gelfond and Leone [11] 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.* [10], 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 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 are determined in terms of *alphabets*. This kind of program correspondence includes, as special instances, the well-known notions of *strong equivalence* [17], *uniform equivalence* [9], relativised variants thereof [23], as well as the practicably important case of program comparison under *projected* answer sets. In the last setting, not a whole answer set of a program is of interest, but only its intersection on a subset of all letters. This includes, in particular, removal of auxiliary letters.

For illustration, consider the following two programs

[*]This work was partially supported by the Austrian Science Fund (FWF) under grant P18019. The second author was also supported by the Austrian Federal Ministry of Transport, Innovation, and Technology (BMVIT) and the Austrian Research Promotion Agency (FFG) under grant FIT-IT-810806.

which both express the selection of exactly one of the atoms a, b. An atom can only be selected if it can be derived together with the context:

$$\begin{aligned} P = \{\; & sel(b) \leftarrow b, not\ out(b);\ sel(a) \leftarrow a, not\ out(a); \\ & out(a) \vee out(b) \leftarrow a, b\}; \\ Q = \{\; & sel(a) \vee sel(b) \leftarrow a;\ sel(a) \vee sel(b) \leftarrow b; \\ & fail \leftarrow sel(a), not\ a, not\ fail; \\ & fail \leftarrow sel(b), not\ b, not\ fail\}. \end{aligned}$$

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 $A = \{a, b\}$, or, more generally, any set A of atoms not containing the atoms $sel(a)$, $sel(b)$, $out(a)$, $out(b)$, 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 together with some initial experimental results. The overall approach of the system, which we call cc$\top$ ("correspondence-checking tool"), is to reduce correspondences problems to satisfiability in *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 atoms.

The motivation to use such an approach is twofold. First, complexity results [10] 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 realised in a straightforward manner 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 recent work [21], such translations from correspondence checking to QBFs have been developed; moreover, they are constructible in *linear time*. Second, various practicably efficient solvers for quantified propositional logic are currently available (see, e.g., Le Berre *et al.* [15]). Hence, such tools are used as back-end inference engines in our system to verify the correspondence problems under consideration. In fact, reduction methods to QBFs have already been successfully applied in diverse areas like nonmonotonic reasoning [6, 5], paraconsistent reasoning [3, 1], and planning [20].

Previous systems implementing different forms of equivalence, being special cases of correspondence notions in the framework of Eiter *et al.* [10], are SELP [4] and DLPEQ [18]. In SELP, the problem of checking strong equivalence is reduced to propositional logic, making use of SAT solvers as back-end inference engines. Our system generalises SELP in the sense that cc$\top$ handles correspondence problems which coincide with testing strong equivalence by the same reduction as used in SELP. The system DLPEQ, on the other hand, is capable of comparing disjunctive logic programs under ordinary equivalence by a reduction to logic programs, employing extant answer-set solvers as underlying inference engines. While both SELP and DLPEQ are dedicated tools designed to handle specific forms of program correspondence, cc$\top$, in contrast, allows for checking a wide range of parameterisable equivalence tests, useful for debugging purposes and modular programming.

The paper is organised as follows. In the next section, we recapitulate the basic facts about logic programs under the answer-set semantics and quantified propositional logic. Section 3 deals with the details about our tool: first, we review the underlying encodings, then we discuss some special cases as well as some normalisation steps required for most QBF solvers, and finally we provide some information regarding its usage. Section 4, then, reports experimental results and Section 5 wraps up the paper with some concluding remarks.

2. Preliminaries

Throughout the paper, we use the following notation: For an interpretation I, i.e., a set of atoms, and a set $\mathcal{S}$ of interpretations, we write $\mathcal{S}|_I = \{Y \cap I \mid Y \in \mathcal{S}\}$. For a singleton set $\mathcal{S} = \{Y\}$, we write $Y|_I$ instead of $\mathcal{S}|_I$, whenever convenient.

2.1. Logic Programs

We are concerned with *propositional disjunctive logic programs* (DLPs) which are finite sets of rules of form

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

$n \geq m \geq l \geq 0$, where all a_i are propositional atoms from some fixed universe $\mathcal{U}$ and not denotes *default negation*. If all atoms occurring in a program P are from a given set $A \subseteq \mathcal{U}$ of atoms, we say that P is a program *over* A. The set of all programs over A is denoted by $\mathcal{P}_A$. A rule r of form (1) is said to be *true* under an interpretation I, symbolically $I \models r$, iff $\{a_1, \ldots, a_l\} \cap I \neq \emptyset$ whenever $\{a_{l+1}, \ldots, a_m\} \subseteq I$ and $\{a_{m+1}, \ldots, a_n\} \cap I = \emptyset$. If $I \models r$ holds, then I is also said to be a *model* of r. As well, I is a model of a program P iff $I \models r$, for all $r \in P$.

Following Gelfond and Lifschitz [12], an interpretation I is an *answer set* of a program P iff it is a minimal model of the *reduct* P^I, resulting from P by

- deleting all rules containing default negated atoms $not\ a$ such that $a \in I$; and

- deleting all default negated atoms in the remaining rules.

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

In order to semantically compare programs, different notions of equivalence have been introduced in the context of the answer-set semantics. Besides *ordinary equivalence* between programs, which checks whether two programs have the same answer sets, the more restrictive notions of *strong equivalence* [17] and *uniform equivalence* [9] have been introduced: Two programs, P and Q, are strongly equivalent iff $\mathcal{AS}(P \cup R) = \mathcal{AS}(Q \cup R)$, for any program R, and they are uniformly equivalent iff $\mathcal{AS}(P \cup R) = \mathcal{AS}(Q \cup R)$, for any set R of *facts*, i.e., rules of form $a \leftarrow$, for some atom a. Also, relativised equivalence notions, taking the alphabet of the extension set R into account, have been defined [23].

In abstracting from these notions, Eiter *et al.* [10] introduced a general framework for specifying differing notions of program correspondence. In this framework, one parameterises, on the one hand, the *context*, i.e., the class of programs used to be added to the programs under consideration, and, on the other hand, the relation that has to hold between the collection of answer sets of the extended programs. In what follows, we focus on two important instantiations of the general framework, viz. *inclusion problems*, given by quadruples of form $(P, Q, \mathcal{P}_A, \subseteq_B)$, and *equivalence problems*, given by quadruples of form $(P, Q, \mathcal{P}_A, =_B)$. Here, A and B are sets of atoms, with A fixing the alphabet of the context $\mathcal{P}_A$, and $\subseteq_B$ and $=_B$ are projections (into B) of the standard subset and set-equality relation, respectively, defined as follows: for every set $\mathcal{S}, \mathcal{S}'$, $\mathcal{S} \subseteq_B \mathcal{S}'$ iff $\mathcal{S}|_B \subseteq \mathcal{S}'|_B$, and $\mathcal{S} =_B \mathcal{S}'$ iff $\mathcal{S}|_B = \mathcal{S}'|_B$. We say that $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ (resp., $\Pi = (P, Q, \mathcal{P}_A, =_B)$) *holds* iff, for all $R \in \mathcal{P}_A$, $\mathcal{AS}(P \cup R) \subseteq_B \mathcal{AS}(Q \cup R)$ (resp., $\mathcal{AS}(P \cup R) =_B \mathcal{AS}(Q \cup R)$). Note that $(P, Q, \mathcal{P}_A, =_B)$ holds iff $(P, Q, \mathcal{P}_A, \subseteq_B)$ and $(Q, P, \mathcal{P}_A, \subseteq_B)$ jointly hold.

The following proposition briefly summarises the complexity landscape within this framework [10, 19, 23].

Proposition 1 *Given programs P and Q, sets A and B of atoms, and $\rho \in \{\subseteq_B, =_B\}$, deciding whether a correspondence problem $(P, Q, \mathcal{P}_A, \rho)$ holds is:*

1. Π_4^P*-complete, in the general case;*
2. Π_3^P*-complete, providing $A = \emptyset$;*
3. Π_2^P*-complete, providing $B = \mathcal{U}$; and*
4. coNP*-complete, providing $A = \mathcal{U}$.*

While Case 1 deals with arbitrary correspondence problems of the considered forms, for the other cases, we have the following observations: Case 2 amounts to *ordinary equivalence with projection*, i.e., the answer sets of two programs relative to a specified set B of atoms are compared; Case 3 amounts to *strong equivalence relative to A* and includes, as a special case (viz., for $A = \emptyset$), *ordinary equivalence*; finally, Case 4 includes *strong equivalence* (for $B = \mathcal{U}$) as well as strong equivalence with projection.

The Π_4^P-hardness result shows that, in general, checking the correspondence of two programs cannot (presumably) be efficiently encoded in terms of ASP, which has its basic reasoning tasks contained in Σ_2^P or Π_2^P. However, correspondence checking can be efficiently encoded in terms of *quantified propositional logic*.

2.2. Quantified Propositional Logic

Quantified propositional logic is an extension of classical propositional logic in which formulas are permitted to contain quantifications over propositional variables. In particular, this language contains, for any atom p, unary operators of form $\forall p$ and $\exists p$, called *universal* and *existential quantifiers*, respectively, where $\exists p$ is defined as $\neg \forall p \neg$. Formulas of this language are also called *quantified Boolean formulas* (QBFs), and we denote them by Greek upper-case letters throughout this paper.

For a QBF of form $\mathsf{Q}p\,\Psi$, where $\mathsf{Q} \in \{\exists, \forall\}$, we call Ψ the *scope* of $\mathsf{Q}p$. An occurrence of an atom p is *free* in a QBF Φ if it does not occur in the scope of a quantifier $\mathsf{Q}p$ in Φ. In what follows, we tacitly assume that every subformula $\mathsf{Q}p\,\Phi$ of a QBF contains a free occurrence of p in Φ, and for two different subformulas $\mathsf{Q}p\,\Phi$, $\mathsf{Q}q\,\Psi$ of a QBF, we require $p \neq q$. Moreover, given a finite set P of atoms, $\mathsf{Q}P\,\Psi$ stands for any QBF $\mathsf{Q}p_1\mathsf{Q}p_2 \ldots \mathsf{Q}p_n\Psi$ such that the variables $p_1, \ldots, p_n$ are pairwise distinct and $P = \{p_1, \ldots, p_n\}$. Finally, for an atom p and a set I of atoms, $\Phi[p/I]$ denotes the QBF resulting from Φ by replacing each free occurrence of p in Φ by $\top$ if $p \in I$ and by $\bot$ otherwise.

For an interpretation I and a QBF Φ, the relation $I \models \Phi$ is inductively defined as in classical propositional logic, whereby universal quantifiers are evaluated as follows:

$$I \models \forall p\ \Phi \text{ iff } I \models \Phi[p/\{p\}] \text{ and } I \models \Phi[p/\emptyset].$$

The terms *true*, *false*, *satisfiable*, and *valid* are defined as in classical propositional logic. Note that a *closed* QBF, i.e., a QBF without free variable occurrences, is either true under any interpretation or false under any interpretation.

A QBF Φ is said to be in *prenex normal form* (PNF) iff it is closed and of the form

$$\mathsf{Q}_n P_n \ldots \mathsf{Q}_1 P_1\ \phi, \tag{2}$$

where $n \geq 0$, ϕ is a propositional formula, $\mathsf{Q}_i \in \{\exists, \forall\}$ such that $\mathsf{Q}_i \neq \mathsf{Q}_{i+1}$ for $1 \leq i \leq n-1$, $(P_1, \ldots, P_n)$

is a partition of the propositional variables occurring in ϕ, and $P_i \neq \emptyset$, for each $1 \leq i \leq n$. We say that Φ is in *prenex conjunctive normal form* (PCNF) iff Φ is of the form (2) and ϕ is in conjunctive normal form. Furthermore, a QBF of form (2) is also referred to as an *(n, Q_n)-QBF*. Any closed QBF Φ is easily transformed into an equivalent QBF in prenex normal form such that each quantifier occurrence from the original QBF corresponds to a quantifier occurrence in the prenex normal form. Let us call such a QBF the *prenex normal form of* Φ. In general, there are different ways to obtain an equivalent prenex QBF (cf. Egly *et al.* [7] for more details on this issue). The following property is essential:

Proposition 2 *For every $k \geq 0$, deciding the truth of a given $(k, \exists)$-QBF (resp., $(k, \forall)$-QBF) is Σ_k^P-complete (resp., Π_k^P-complete).*

Hence, any decision problem $\mathcal{D}$ in Σ_k^P (resp., Π_k^P) can be mapped in polynomial time to a $(k, \exists)$-QBF (resp., $(k, \forall)$-QBF) Φ such that $\mathcal{D}$ holds iff Φ is valid. In particular, any correspondence problem $(P, Q, \mathcal{P}_A, \rho)$, for $\rho \in \{\subseteq_B, =_B\}$, can be reduced in polynomial time to a $(4, \forall)$-QBF. Our implemented tool, described next, relies on two such mappings, which are actually constructible in *linear time*.

3. Computing Correspondence Problems

We now describe the system cc$\top$, which allows to verify the correspondence of two programs. It relies on efficient reductions from correspondence problems to QBFs [21], reviewed in the first subsection. Then, we briefly discuss special forms of inclusion or equivalence problems. Afterwards, we give details concerning the transformation of the resultant QBFs into PCNF, which is necessary because most extant QBF solvers rely on input of this form. Finally, we give some details concerning the general syntax and invocation of cc$\top$.

3.1. Basic Encodings

Following Tompits and Woltran [21], we present two reductions from inclusion problems to QBFs, $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$, where $\mathsf{T}[\cdot]$ is an explicit optimisation of $\mathsf{S}[\cdot]$ yielding QBFs that always reflect (with respect to their structure) the complexity of the encoded problem (cf. Proposition 1). Equivalence problems can be decided by the composition of two inclusion problems. An encoding for equivalence problems is thus obtained via a conjunction of two instantiations of $\mathsf{S}[\cdot]$ (or $\mathsf{T}[\cdot]$). In view of this, for every equivalence problem $\Pi = (P, Q, \mathcal{P}_A, =_B)$, we shall denote by Π' and Π'' the associated inclusion problems $(P, Q, \mathcal{P}_A, \subseteq_B)$ and $(Q, P, \mathcal{P}_A, \subseteq_B)$, respectively.

For our encodings, we use the following building blocks. We use sets of globally new atoms in order to refer to different assignments of the atoms from the compared programs within a single formula. Formally, given an indexed set V of atoms, we assume (pairwise) disjoint copies $V_i = \{v_i \mid v \in V\}$, for every $i \geq 1$. Furthermore, let

1. $(V_i \leq V_j) := \bigwedge_{v \in V}(v_i \rightarrow v_j)$;
2. $(V_i < V_j) := (V_i \leq V_j) \wedge \neg(V_j \leq V_i)$; and
3. $(V_i = V_j) := (V_i \leq V_j) \wedge (V_j \leq V_i)$.

Observe that the latter is equivalent to $\bigwedge_{v \in V}(v_i \leftrightarrow v_j)$.

These three "operators" allow to compare different subsets of atoms from a common set under subset inclusion, proper-subset inclusion, and equality, respectively. In accordance to the renaming of atoms, we use subscripts as a general renaming schema for formulas and rules. That is, for each $i \geq 1$, α_i is the result of replacing each occurrence of an atom p in α by p_i, where α is any formula or rule. Furthermore, for a rule r of form (1), we define $H(r) = a_1 \vee \cdots \vee a_l$, $B^+(r) = a_{l+1} \wedge \cdots \wedge a_m$, and $B^-(r) = \neg a_{m+1} \wedge \cdots \wedge \neg a_n$; and for a program P, we define

$$P_{i,j} = \bigwedge_{r \in P} \big((B^+(r_i) \wedge B^-(r_j)) \rightarrow H(r_i)\big).$$

Formally, we have the following relation: Let P be a program over atoms V, I an interpretation, and $X, Y \subseteq V$ such that, for some i, j, $I|_{V_i} = X_i$ and $I|_{V_j} = Y_j$. Then, $X \models P^Y$ iff $I \models P_{i,j}$.

We proceed with the first encoding.

Definition 1 *Let P and Q be programs over V, $A, B \subseteq V$, and $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ an inclusion problem. Then,*

$$\mathsf{S}[\Pi] := \neg\exists V_1 \Big(P_{1,1} \wedge \mathsf{S}^1(P, A) \wedge \forall V_3 \big(\mathsf{S}^2(Q, A, B) \rightarrow \mathsf{S}^3(P, Q, A)\big)\Big), \textit{ where}$$

$$\mathsf{S}^1(P, A) := \forall V_2 \big(((A_2 = A_1) \wedge (V_2 < V_1)) \rightarrow \neg P_{2,1}\big),$$
$$\mathsf{S}^2(Q, A, B) := \big((A \cup B)_3 = (A \cup B)_1\big) \wedge Q_{3,3}, \textit{ and}$$
$$\mathsf{S}^3(P, Q, A) := \exists V_4 \big((V_4 < V_3) \wedge Q_{4,3} \wedge \big((A_4 < A_1) \rightarrow \forall V_5 (((A_5 = A_4) \wedge (V_5 \leq V_1)) \rightarrow \neg P_{5,1})\big)\big).$$

In the scope, Φ, of $\exists V_1$ the conditions for deciding whether a so-called *spoiler* [10] for the inclusion problem Π exists are encoded. Such a spoiler for Π exists iff Π does *not* hold. Hence, the encoding Φ is unsatisfiable iff Π holds. Thus, since $\mathsf{S}[\Pi] = \neg\exists V_1 \Phi$, we get:

Proposition 3 ([21]) *For any inclusion problem Π, Π holds iff $\mathsf{S}[\Pi]$ is valid.*

In what follows, we review a more compact encoding which, in particular, reduces the number of universal quantifications. The idea is to save on the fixed assignments, as, e.g., in $\mathsf{S}^2(Q, A, B)$, where we have $(A \cup B)_3 = (A \cup B)_1$. This calls for a more subtle renaming schema for programs, however. Let $\mathcal{V}$ be a set of indexed atoms, and let r be a rule. Then, $r^{\mathcal{V}}_{i,k}$ results from r by replacing each atom x in r by x_i, providing $x_i \in \mathcal{V}$, and by x_k otherwise. For a program P, we define

$$P^{\mathcal{V}}_{i,j,k} := \bigwedge_{r \in P} \big((B^+(r^{\mathcal{V}}_{i,k}) \wedge B^-(r^{\mathcal{V}}_{j,k})) \rightarrow H(r^{\mathcal{V}}_{i,k})\big).$$

Moreover, for every $i \geq 1$, every set V of atoms, and every set C, $V^C_i := (V \setminus C)_i$.

Definition 2 *Let P and Q be programs over V and $A, B \subseteq V$. Furthermore, let $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ be an inclusion problem and $\mathcal{V} = V_1 \cup V^A_2 \cup V^{A \cup B}_3 \cup V_4 \cup V^A_5$. Then,*

$$\begin{aligned}\mathsf{T}[\Pi] := \neg \exists V_1 \Big(& P_{1,1} \wedge \mathsf{T}^1(P, A, \mathcal{V}) \wedge \\ & \forall V^{A\cup B}_3 \big(Q^{\mathcal{V}}_{3,3,1} \rightarrow \mathsf{T}^3(P, Q, A, \mathcal{V})\big)\Big), \textit{ where}\end{aligned}$$

$$\begin{aligned}\mathsf{T}^1(P, A, \mathcal{V}) &:= \forall V^A_2 \big((V^A_2 < V^A_1) \rightarrow \neg P^{\mathcal{V}}_{2,1,1}\big) \textit{ and} \\ \mathsf{T}^3(P, Q, A, \mathcal{V}) &:= \exists V_4 \big((V_4 < ((A \cup B)_1 \cup V^{A\cup B}_3)) \wedge \\ &\quad Q^{\mathcal{V}}_{4,3,1} \wedge \big((A_4 < A_1) \rightarrow \\ &\quad \forall V^A_5 ((V^A_5 \leq V^A_1) \rightarrow \neg P^{\mathcal{V}}_{5,1,4})\big)\big).\end{aligned}$$

Analogous to Proposition 3, the following holds:

Proposition 4 ([21]) *For any inclusion problem Π, Π holds iff $\mathsf{T}[\Pi]$ is valid.*

Corollary 1 *Let Π be an equivalence problem. The following statements are equivalent:* (*i*) *Π holds;* (*ii*) *$\mathsf{S}[\Pi'] \wedge \mathsf{S}[\Pi'']$ is valid; and* (*iii*) *$\mathsf{T}[\Pi'] \wedge \mathsf{T}[\Pi'']$ is valid.*

3.2. Special Cases

We now analyse how our encodings behave in certain instances of the equivalence framework which are located at lower levels of the polynomial hierarchy (cf. Proposition 1).

For the case of *strong equivalence* [17], i.e., problems of form $\Pi = (P, Q, \mathcal{P}_A, =_A)$ with $A = \mathcal{U}$, the encodings $\mathsf{T}[\Pi']$ and $\mathsf{T}[\Pi'']$ can be drastically simplified, since $V^A_2 = V^A_3 = V^A_5 = \emptyset$. In particular, $\mathsf{T}[\Pi']$ is equivalent to

$$\neg \exists V_1 \Big(P_{1,1} \wedge \big(Q_{1,1} \rightarrow \exists V_4 \big((V_4 < V_1) \wedge Q_{4,1} \wedge \neg P_{4,1}\big)\big)\Big).$$

Now, the encoding for strong equivalence amounts to a single unsatisfiability test, witnessing the coNP-completeness complexity for this problem [19].

It is straightforward to check that for other special cases similar simplifications can be achieved [21]. For strong equivalence relative to a set A of atoms [23], the QBF simplifies since $V^{A\cup B}_3 = \emptyset$. The case of ordinary equivalence, i.e., problems of form $\Pi = (P, Q, \mathcal{P}_A, =)$ with $A = \emptyset$, is, indeed, a special case of relativised strong equivalence. As an additional optimisation we can drop the subformula

$$(A_4 < A_1) \rightarrow \forall V^A_5 \big((V^A_5 \leq V^A_1) \rightarrow \neg P^{\mathcal{V}}_{5,1,4}\big)$$

from part T^3 of $\mathsf{T}[\Pi']$ because $A = \emptyset$. We point out that those simplifications are incorporated within our system.

For the encoding $\mathsf{T}[\cdot]$, the structure of the resulting QBF always reflects the complexity of the correspondence problem according to Proposition 1. This does not hold for formulas stemming from $\mathsf{S}[\cdot]$, however. In any case, our tool implements both encodings in order to provide interesting benchmarks for QBF solvers with respect to their capability to find implicit optimisations for equivalent QBFs.

3.3. Transformations into Normal Forms

Most available QBF solvers require the input QBF to be in a certain normal form, viz. in prenex conjunctive normal form (PCNF). Hence, to employ these solvers for our tool, the translations described above have to be transformed by a further two-phased normalisation step: (1) translation of the QBF into prenex normal form (PNF); and (2) translation of the propositional part of the formula in PNF into CNF.

The step of prenexing is not deterministic. Indeed, there are various so-called *prenexing strategies* [7]. The selection of such a strategy (also depending on the solver used) crucially influences the running times (see also our results below). For illustration, consider the quantifier dependencies for the encoding $\mathsf{S}[\cdot]$:

$$\begin{tikzcd}[column sep=small] & \forall V_1 \arrow[dl, no head] \arrow[dr, no head] & \\ \exists V_2 & & \exists V_3 \arrow[d, no head] \\ & & \forall V_4 \arrow[d, no head] \\ & & \exists V_5 \end{tikzcd}$$

Note that we consider the quantifier dependencies cleansed with respect to their polarities by applying the usual equivalence preserving transformations as known from first-order logic. Here, the left branch results from the subformula S^1 and the right one results from the subformula $\forall V_3(\mathsf{S}^2(Q, A, B) \rightarrow \mathsf{S}^3(P, Q, A))$.

Inspecting these quantifier dependencies, we can group $\exists V_2$ either together with $\exists V_3$ or with $\exists V_5$. This yields the following two basic ways for prenexing our encodings:

$\uparrow$: $\forall V_1 \exists (V_2 \cup V_3) \forall V_4 \exists V_5$; and

$\downarrow$: $\forall V_1 \exists V_3 \forall V_4 \exists (V_5 \cup V_2)$.

Together with the two encodings $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$, we thus get four different alternatives to represent an inclusion problem

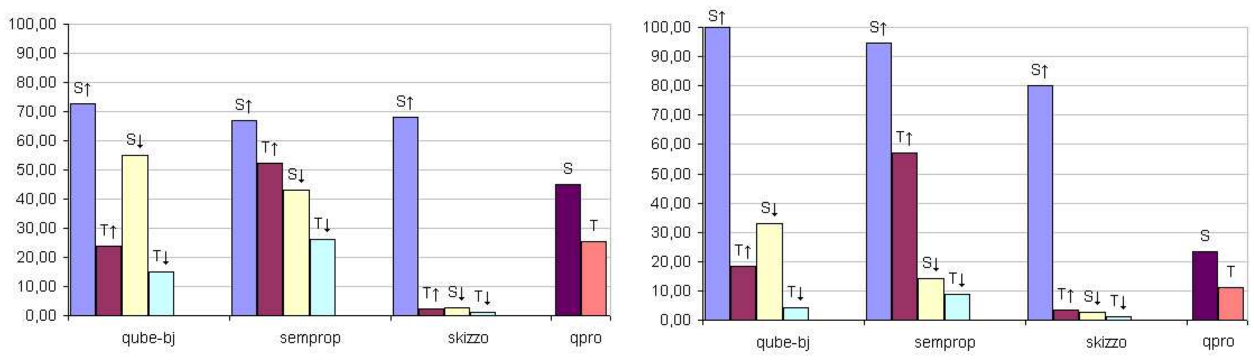


Figure 1. Π_4^P-hard random problems: Running times (in seconds) for true (left chart) and false (right chart) instances.

in terms of a prenex QBF; we will denote them by $\mathsf{S}_{\uparrow}[\cdot]$, $\mathsf{S}_{\downarrow}[\cdot]$, $\mathsf{T}_{\uparrow}[\cdot]$, and $\mathsf{T}_{\downarrow}[\cdot]$, respectively.

Concerning the transformation of the propositional part of a prenex QBF into CNF, we apply a method following Tseitin [22] in which new atoms, so-called *labels*, are introduced abbreviating subformula occurrences and which has the property that the resultant CNFs are always polynomial in the size of the input formula.

3.4. The Implemented Tool

The system cc$\top$ 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, together with the potential simplifications discussed above. It takes as input two programs, P and Q, and 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.

The syntax to specify programs in cc$\top$ corresponds to the basic `DLV` syntax.[1] cc$\top$ is entirely developed 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. More information about cc$\top$ and how to use it, as well as information about the benchmarks below, is available at the following URL:

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

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

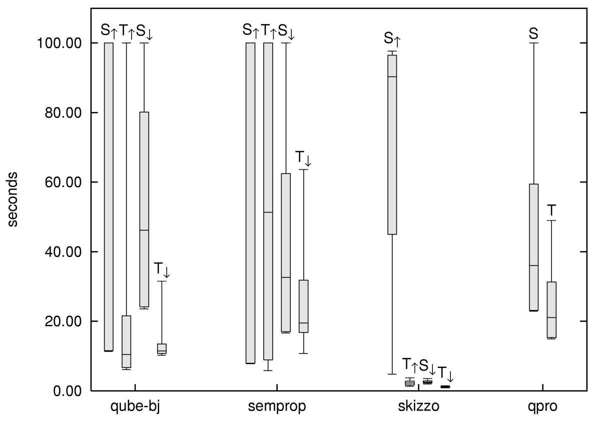
4. Experimental Results

Our experiments were conducted to determine the behaviour of different QBF solvers in combination with the encodings $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$, 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 $\mathsf{T}_{\downarrow}[\cdot]$. We implemented a generator of inclusion problems which emanate from the proof of the Π_4^P-hardness of inclusion checking [10]. The strategy to obtain such instances is to

1. generate a $(4, \forall)$-QBF Φ in PCNF by random;
2. reduce Φ to a problem $\Pi = (P, Q, \mathcal{P}_A, \subseteq_B)$ such that Π holds iff Φ is valid; and
3. apply cc$\top$ to derive the corresponding encoding Ψ for Π.

We use here a parameterisation for the generation of random QBFs such that the benchmark set yields a nearly 50% distribution between the true and false instances. We set up a test series comprising 1000 instances of inclusion problems generated that way (465 of them evaluating to true). Program P has 620 rules, and program Q has 280 rules, using a total of 40 atoms; the sets A and B contain 16 atoms. After employing cc$\top$, 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 state-of-the-art QBF solvers in our analysis: `qube-bj` [13], `semprop` [16], `skizzo` [2], and `qpro` [8]. The former three require QBFs in PCNF as input (thus, we tested them using encodings $\mathsf{S}_{\uparrow}[\cdot]$, $\mathsf{S}_{\downarrow}[\cdot]$, $\mathsf{T}_{\uparrow}[\cdot]$, and $\mathsf{T}_{\downarrow}[\cdot]$), while `qpro` admits arbitrary QBFs as input (we tested it with the non-prenex encodings $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$).



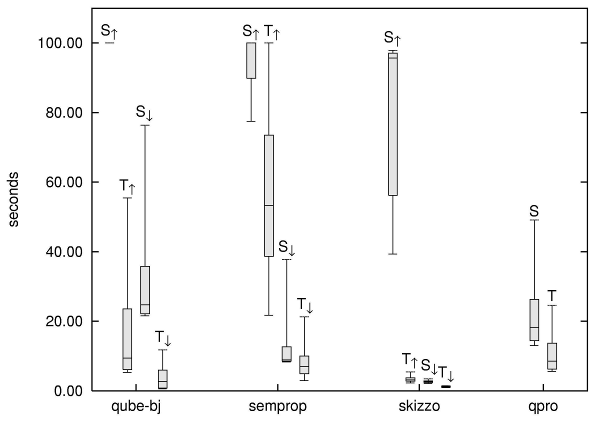


Figure 2. Whisker-box plots corresponding to Figure 1 for true (left chart) and false (right chart) instances.

Our results are depicted in Figure 1. The y-axis shows the (arithmetically) average running time in seconds (time-out was 100 seconds) for each solver with respect to the chosen translation and prenexing strategy. The performance of the PCNF-solvers turned out to be highly dependent on the prenexing strategy, and $\downarrow$ dominated $\uparrow$.

A more thorough analysis of the data with respect to their distribution is given in Figure 2. By means of whisker-box plots, we depict, for each measuring point, median (horizontal line inside the box), 25%- and 75%-quantile (lower and upper border of the boxes, respectively), and the 5%- and 95%-quantile (lower and upper horizontal bar at the end of the vertical lines, the so-called *whiskers*, respectively). Due to the chosen time-out of 100 seconds, the whisker-box plots are slightly distorted near the 100 seconds border.

For the special case of ordinary equivalence, we compared our approach against the system DLPEQ [18], which is based on a reduction to disjunctive logic programs, using `gnt` [14] as underlying answer-set solver. The benchmarks rely on randomly generated $(2, \exists)$-QBFs. Each QBF is reduced to a program such that the latter possesses an answer set iff the original QBF is valid. The idea is to compare the program with itself having a randomly selected rule dropped, thus simulating a "sloppy" programmer (for more details, cf. Oikarinen and Janhunen [18]). The results are shown in Figure 3. The size of the original QBFs (with respect to the number of variables) is given on the x-axis, and running times (in seconds) are given on the y-axis; time-out was set to 120 seconds. For cc$\top$, we compared the same back-end solvers as above, using encoding $\mathsf{T}[\cdot]$. Recall that for ordinary equivalence cc$\top$ provides $(2, \forall)$-QBFs, thus we can resign on the distinction between prenexing strategies.

Not surprisingly, the special-purpose DLPEQ system turns out to be faster than cc$\top$. Interestingly, however, is

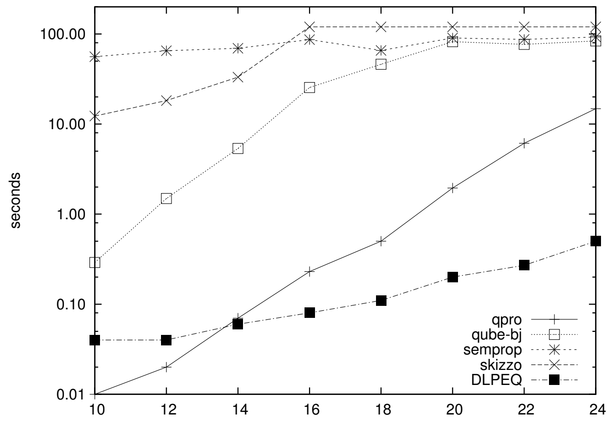


Figure 3. Comparing ccT against DLPEQ for the special case of ordinary equivalence.

the observation that, among the tested QBF solvers, `qpro` is the most competitive one, while the PCNF-QBF solvers perform bad even for small instances.

5. Conclusion

In this paper, we discussed an implementation for advanced program comparisons in answer-set programming via encodings into quantified propositional logic. This approach was motivated by the high computational complexity we have to face for correspondence checking, making a direct realisation via ASP hard to accomplish. Since practicably efficient solvers for quantified propositional logic are available, they can be employed as back-end inference engines to verify correspondence problems using the proposed encodings. Moreover, since such problems are one of the few natural ones lying above the second level of the polynomial hierarchy, yet still in PSPACE, we believe that our encodings also provide valuable benchmarks for evaluating QBF solvers, for which there is currently a lack of structured problems with more than one quantifier alternation (cf. Le Berre *et al*. [15]).

References

[1] O. Arieli and M. Denecker. Reducing Preferential Paraconsistent Reasoning to Classical Entailment. *Journal of Logic and Computation*, 13(4):557–580, 2003.

[2] M. Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *Proceedings of the 20th International Conference on Automated Deduction* (*CADE-20*), volume 3632 of *Lecture Notes in Computer Science*, pages 369–376. Springer, 2005.

[3] P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Representing Paraconsistent Reasoning via Quantified Propositional Logic. In *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*, pages 84–118. Springer, 2005.

[4] Y. Chen, F. Lin, and L. Li. SELP - A System for Studying Strong Equivalence Between Logic Programs. In *Proceedings of the 8th International Conference on Logic Programming and Non-Monotonic Reasoning* (*LPNMR 2005*), volume 3552 of *Lecture Notes in Artificial Intelligence*, pages 442–446. Springer, 2005.

[5] J. Delgrande, T. Schaub, H. Tompits, and S. Woltran. On Computing Solutions to Belief Change Scenarios. *Journal of Logic and Computation*, 14(6):801–826, 2004.

[6] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings of the 17th National Conference on Artificial Intelligence* (*AAAI 2000*), pages 417–422. AAAI Press/MIT Press, 2000.

[7] U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In *Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing* (*SAT 2003*), volume 2919 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2004.

[8] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In *Proceedings of the 17th European Conference on Artificial Intelligence* (*ECAI 2006*), 2006.

[9] T. Eiter and M. Fink. Uniform Equivalence of Logic Programs under the Stable Model Semantics. In *Proceedings of the 19th International Conference on Logic Programming* (*ICLP 2003*), volume 2916 in *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.

[10] T. Eiter, H. Tompits, and S. Woltran. On Solution Correspondences in Answer Set Programming. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (*IJCAI 2005*), pages 97–102, 2005.

[11] M. Gelfond and N. Leone. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[12] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[13] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic satisfiability. *Artificial Intelligence*, 145:99–120, 2003.

[14] T. Janhunen, I. Niemelä, D. Seipel, and P. Simons. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM Transactions on Computational Logic*, 7(1):1–37, 2006.

[15] D. Le Berre, M. Narizzano, L. Simon, and L. A. Tacchella. The Second QBF Solvers Comparative Evaluation. In *Proceedings of the 7th International Conference on the Theory and Applications of Satisfiability Testing* (*SAT 2004*), volume 3542 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2005.

[16] R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Proceedings of the 11th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods* (*TABLEAUX 2002*), volume 2381 of *Lecture Notes in Artificial Intelligence*, pages 160–175. Springer, 2002.

[17] V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[18] E. Oikarinen and T. Janhunen. Verifying the Equivalence of Logic Programs in the Disjunctive Case. In *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning* (*LPNMR 2004*), volume 2923 of *Lecture Notes in Artificial Intelligence*, pages 180–193. Springer, 2004.

[19] D. Pearce, H. Tompits, and S. Woltran. Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In *Proceedings of the 10th Portuguese Conference on Artificial Intelligence* (*EPIA 2001*), volume 2258 of *Lecture Notes in Artificial Intelligence*, pages 306–320. Springer, 2001.

[20] J. Rintanen. Constructing Conditional Plans by a Theorem Prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[21] H. Tompits and S. Woltran. Towards Implementations for Advanced Equivalence Checking in Answer-Set Programming. In *Proceedings of the 21st International Conference on Logic Programming* (*ICLP 2005*), volume 3668 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2005.

[22] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pages 234–259. Seminars in Mathematics, V.A. Steklov Mathematical Institute, vol. 8, Leningrad, 1968. English translation: Consultants Bureau, New York, 1970, pp. 115–125.

[23] S. Woltran. Characterizations for Relativized Notions of Equivalence in Answer Set Programming. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence* (*JELIA 2004*), volume 3229 of *Lecture Notes in Artificial Intelligence*, pages 161–173. Springer, 2004.