cc⊤: A Correspondence-Checking Tool for Logic Programs under the Answer-Set Semantics

Johannes Oetsch[1], Martina Seidl[2], Hans Tompits[1], and Stefan Woltran[1]

[1] Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,tompits,stefan}@kr.tuwien.ac.at
[2] Institut für Softwaretechnik 188/3, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
seidl@big.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 captures different notions of equivalence, including well-known ones like ordinary, strong, and uniform 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 an implementation to verify program correspondences in this general framework. The system, called cc⊤, relies on linear-time constructible reductions to *quantified propositional logic* and uses extant solvers for the latter language as back-end inference engines.

1 General Information

To support engineering tasks in *answer-set programming* (ASP) [4], an important issue is to determine the equivalence of different problem encodings, given by two logic programs. Various notions of equivalence between programs have been studied in the literature [7, 2, 11] including the recently proposed framework by Eiter *et al.* [3], 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*. This kind of program correspondence includes the well-known notions of *ordinary equivalence*, *strong equivalence* [7], *uniform equivalence* [2], *relativised* variants of the latter two [11], as well as the practicably important case of program comparison under *projected* answer sets as special instances. In the latter setting, not a whole answer set of a program is of interest, but only its intersection on a subset of all letters, corresponding to a removal of auxiliary letters in computation.

In this paper, we briefly describe the main features of the system cc⊤ (short for "correspondence-checking tool"), which implements correspondence problems in the framework of Eiter *et al.* [3]. Compared to similar tools which are restricted to the notions of strong and ordinary equivalence [1, 9], cc⊤ supports the user with more fine-grained equivalence notions, allowing practical comparisons useful for debugging and modular programming. Further information about cc⊤ is also available on the Web at

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

2 System Specifics

Theoretical Background. We are concerned here with *disjunctive logic programs* with default negation over a universe $\mathcal{U}$ of propositional atoms under the *answer-set semantics* [5]. Given a program P, we denote by $\mathcal{AS}(P)$ the collection of its answer sets; moreover, P_A denotes the class of all programs given over a set $A \subseteq \mathcal{U}$ of atoms. Two programs, P and Q, are *ordinarily equivalent* iff $\mathcal{AS}(P) = \mathcal{AS}(Q)$. P and Q are *strongly equivalent* [7] iff, for any program R, $\mathcal{AS}(P \cup R) = \mathcal{AS}(Q \cup R)$. In abstracting from these equivalence notions, Eiter *et al.* [3] introduced the notion of a *correspondence problem* which allows to specify, on the one hand, a *context*, i.e., a 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 collections of answer sets of the extended programs. Following Eiter *et al.* [3], we focus here on correspondence problems where the context is parametrised in terms of alphabets and the comparison relation is a projection of the standard subset or set-equality relation. In formal terms, a correspondence problem, Π, (over $\mathcal{U}$) is a quadruple of form $(P, Q, \mathcal{P}_A, \rho_B)$, where $P, Q \in \mathcal{P}_\mathcal{U}$, $A, B \subseteq \mathcal{U}$ are sets of atoms, and ρ_B is either $\subseteq_B$ or $=_B$, which are defined as follows: for any sets S, S', $S \subseteq_B S'$ iff $S|_B \subseteq S'|_B$, and $S =_B S'$ iff $S|_B = S'|_B$, where $S|_B = \{I \cap B \mid I \in S\}$. We say that Π *holds* iff, for all $R \in \mathcal{P}_A$, $(\mathcal{AS}(P \cup R), \mathcal{AS}(Q \cup R)) \in \rho_B$. We call Π an *equivalence problem* if ρ_B is given by $=_B$, and an *inclusion problem* if ρ_B is given by $\subseteq_B$, for some $B \subseteq \mathcal{U}$. 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.

Example 1. Consider the following two programs 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):

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

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

Both programs use "local" atoms, $out(\cdot)$ and $fail$, respectively, which are expected not to appear in the context. We thus may consider $\Pi = (P, Q, \mathcal{P}_A, =_B)$ as a suitable equivalence problem, specifying $A = \{a, b\}$ (or, more generally, taking A as any set of atoms not containing $sel(a)$, $sel(b)$, $out(a)$, $out(b)$, and $fail$) and $B = \{sel(a), sel(b)\}$. It is a straightforward matter to check that Π, defined in this way, holds. □

Implementation Methodology. The overall approach of cc$\top$ is (i) to reduce correspondence problems, as introduced above, 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, and (ii) to use extant QBF solvers as back-end inference engines for evaluating the resulting QBFs. The theoretical basis of this approach has been developed in previous work [10], where reductions constructible in *linear time and space* are provided. The motivation for adopting such a

reduction approach is due to the fact that correspondence checking is hard [3], lying on the fourth level of the polynomial hierarchy (thus, QBFs are a suitable target formalism), and since several practicably efficient QBF solvers are available (see, e.g., [6] for an overview about different QBF solvers).

Concerning the translation step, cc$\top$ implements the necessary reductions [10] (together with some simplifications, see [8] for details) from a given inclusion or equivalence problem Π to a corresponding QBF Φ such that Φ is valid iff Π holds. The reductions are designed along so-called *spoilers* [3]: The existence of a spoiler for a given inclusion problem Π indicates that Π does not hold; equivalence tests are encoded by two inclusion tests. In general, the complexity of correspondence checking is Π_4^P-complete, leading to QBFs matching this intrinsic complexity, i.e., they possess up to three quantifier alternations. However, if the specified problem falls into an easier class, cc$\top$ provides an encoding in terms of QBFs which are less involving.

For the evaluation of the resultant QBFs, the user has to employ an off-the-shelf QBF solver. Several such tools are nowadays available [6], but most of them require the input to be in a specific normal form. In such a case, the generated QBFs have to be processed according to the input syntax of the considered solver. Details about the normal-form translation employed by cc$\top$ can be found elsewhere [8].

Applying the System. The system 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 specifies the set of atoms used for the projection in the chosen correspondence relation. The user can select (via command-line options) between two kinds of reductions (see [10] for details), a more naive one or an optimised one, which is also the default option. As well, it can be selected whether the programs are compared with respect to an inclusion or an equivalence problem. The syntax of the programs is the basic `DLV` syntax.[1]

Let us consider the two programs P and Q from Example 1, and suppose they are stored in files `P.dl` and `Q.dl`, respectively. If we want to use cc$\top$ for checking 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

- the context set, stored in file `A`, containing the string "`(a, b)`", and
- the projection set stored in file `B`, containing the string "`(sel(a), sel(b))`".

By default, cc$\top$writes the resulting QBF to the standard-output device. The QBF can then be processed further by QBF solvers. The output can also be piped, e.g., directly to the BDD-based QBF solver `boole`.[2] Choosing the latter way, invoking cc$\top$ on our example thus looks as follows:

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

In this case, the output (from `boole`) is `0` or `1` as answer for the input correspondence problem. In our example, the correspondence holds and the output is therefore `1`.

We developed cc$\top$ 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.

[1] See `http://www.dlvsystem.com/` for details about `DLV`.
[2] See `http://www.cs.cmu.edu/~modelcheck/bdd.html`.

3 Discussion

In this paper, we presented an implementation for advanced program comparisons in answer-set programming via encodings into quantified propositional logic. In other work [8], we reported about initial experimental evaluations of our tool, also containing a comparison between cc$\top$ and the system DLPEQ [9], which computes ordinary equivalence by means of ASP solvers. We furthermore note that, for the special case of checking strong equivalence, our system uses reductions to SAT, i.e., to the satisfiability problem of (ordinary) propositional logic. This is basically done in the same way as in the special-purpose strong-equivalence checker SELP [1]. Thus, our system can be understood as a generalisation of that approach. Compared to these other systems, cc$\top$, however, processes a much broader range of correspondence problems, which are also computationally more involving.

We consider our system as a starting point for a tool box to support modular programming and offline program simplification. Future work includes an extension of the system to other classes of logic programs (like, e.g., nested logic programs) and to further correspondence notions (in particular, ones based on uniform equivalence [2]).

Acknowledgements. 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 by the Austrian Research Promotion Agency (FFG) under grant FIT-IT-810806.

References

1. Y. Chen, F. Lin, and L. Li. SELP - A System for Studying Strong Equivalence between Logic Programs. In *Proc. LPNMR'05*, volume 3662 of *LNCS*, pages 442–446. Springer, 2005.
2. 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. Springer, 2003.
3. T. Eiter, H. Tompits, and S. Woltran. On Solution Correspondences in Answer-Set Programming. In *Proc. IJCAI'05*, pages 97–102, 2005.
4. M. Gelfond and N. Leone. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.
5. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
6. D. Le Berre, M. Narizzano, L. Simon, and A. Tacchella. The Second QBF Solvers Comparative Evaluation. In *Proc. SAT'04*, volume 3542 of *LNCS*, pages 376–392. Springer, 2005.
7. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
8. J. Oetsch, M. Seidl, H. Tompits, and S. Woltran. A Tool for Advanced Correspondence Checking in Answer-Set Programming. In *Proc. NMR'06*, pages 20–29, 2006.
9. E. Oikarinen and T. Janhunen. Verifying the Equivalence of Logic Programs in the Disjunctive Case. In *Proc. LPNMR'04*, volume 2923 of *LNCS*, pages 180–193. Springer, 2004.
10. 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. Springer, 2005.
11. S. Woltran. Characterizations for Relativized Notions of Equivalence in Answer-Set Programming. In *Proc. JELIA'04*, volume 3229 of *LNCS*, pages 161–173. Springer, 2004.