cc⊤ on Stage: Generalised Uniform Equivalence Testing for Verifying Student Assignment Solutions*

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

[1] Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,tompits}@kr.tuwien.ac.at
woltran@dbai.tuwien.ac.at
[2] Institut für Softwaretechnik, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
seidl@big.tuwien.ac.at

Abstract. The tool cc⊤ is an implementation for testing various parameterised notions of program correspondence between logic programs under the answer-set semantics, based on reductions to quantified propositional logic. One such notion is *relativised uniform equivalence with projection*, which extends standard uniform equivalence via two additional parameters: one for specifying the input alphabet and one for specifying the output alphabet. In particular, the latter parameter is used for *projecting* answer sets to the set of designated output atoms, i.e., ignoring auxiliary atoms during answer-set comparison. In this paper, we discuss an application of cc⊤ for verifying the correctness of students' solutions drawn from a laboratory course on logic programming, employing relativised uniform equivalence with projection as the underlying program correspondence notion. We complement our investigation by discussing a performance evaluation of cc⊤, showing that discriminating among different back-end solvers for quantified propositional logic is a crucial issue towards optimal performance.

1 Introduction

This paper deals with a system for testing various refined notions of program correspondence for nonmonotonic logic programs under the answer-set semantics, called cc⊤ (standing for "correspondence-checking tool") [1]. It belongs to a current line of research in answer-set programming (ASP) about questions of program equivalence relevant for different software engineering tasks like optimisation, modular programming, and verification. This research was for the most part initiated by the seminal work of Lifschitz, Pearce, and Valverde [2] about *strong equivalence*, which is defined to hold between two programs P and Q iff $P \cup R$ and $Q \cup R$ are ordinarily equivalent, i.e., have the same answer sets, for every program R (here, R is called *context*). Albeit strong equivalence circumvents the failure of ordinary equivalence to yield a replacement property similar to the one of classical logic, it is however too restrictive for certain applications. This led to the investigation of more liberal notions, chiefly among

* This work was partially supported by the Austrian Science Fund (FWF) under projects P18019 and P21698.

them *uniform equivalence* [3], which is defined similar to strong equivalence except that context programs are restricted to be sets of *facts*. In any case, both strong and uniform equivalence do not take standard programming techniques like the use of local (auxiliary) variables into account, which may occur in some subprograms but which are ignored in the final solutions. In other words, these notions do not admit the *projection* of answer sets to a set of dedicated output atoms. To accommodate issues like the above, Eiter et al. [4] introduced a general framework for specifying parameterised notions of program correspondence, allowing both answer-set projection as well as the specification which kind of context class should be used for program comparison. Thus, these notions generalise not only strong and uniform equivalence but also *relativised* versions thereof [5] (where relativisation refers to the possibility of specifying the alphabet of the context class).

The system cc$\top$ was developed as a checker for specific correspondence problems belonging to the framework of Eiter et al. [4], based on reductions to the satisfiability problem of quantified propositional logic.[3] Such a reduction approach is motivated by two aspects: (i) the complexity of the considered problems—lying on the third and fourth level of the polynomial hierarchy, respectively—is captured by certain classes of quantified propositional formulas, and (ii) the availability of advanced solvers for quantified propositional logic.

Here, we are interested in specific correspondence problems computable by cc$\top$, viz. *propositional query equivalence problems* (PQEPs) [6], which generalise uniform equivalence amounting to *relativised uniform equivalence with projection*.[4] In particular, we discuss how PQEPs can be used to verify the correctness of solutions provided by students as part of their assignments for a laboratory course on knowledge-based systems at our university, relative to a reference solution. The assignments are taken from the domain of model-based diagnosis and use the diagnosis front-end of the well-known ASP solver DLV [7] as underlying reasoning engine. The main difficulty for verifying the students' solutions is that PQEPs deal with propositional programs only whilst the solution programs are non-ground. A naive grounding would not be feasible, so we resorted to a special technique restricting the domain to admissible inputs as well as employing the intelligent grounder of DLV. It turned out that verifying the solutions in this way yielded less false positives than with a test script currently in use, which is based on a collection of sample test cases.

As cc$\top$ admits the use of different QBF solvers as back-end engines, we also report about an experimental evaluation of the tool using a set of benchmark problems showing the runtime behaviour of the system depending on a chosen solver. The experiments were based on a set of parameterisable benchmarks stemming from the hardness proof of the complexity analysis of the corresponding equivalence problems [6]. These benchmarks have the particular advantage that they can be used to easily verify the correctness not only of cc$\top$ but also of the employed QBF solvers. This proved to be very

[3] Recall that quantified propositional logic is an extension of ordinary propositional logic allowing quantifications over atomic formulas. Following custom, we refer to formulas of quantified propositional logic as *quantified Boolean formulas* (QBFs).

[4] The name PQEP stems from taking a database point of view in which programs are considered as queries over databases.

helpful during the development of the system. The experiments show that discriminating among different back-end QBF solvers is crucial towards optimal performance.

The paper is organised as follows. In Section 2, we recapitulate the relevant aspects from ASP and correspondence checking, as well as from quantified propositional logic. Afterwards, in Section 3, we review the theoretical basis of cc$\top$, including some optimisations employed in the system. This is followed by Section 4 containing a discussion of the experimental results. Section 5 discusses the application of cc$\top$ for verifying students' solutions. The paper concludes with a brief summary and outlook in Section 6.

2 Background

Answer-set semantics. We are concerned with *disjunctive logic programs* (DLPs) which are finite sets of safe rules of form

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

where $n \geq m \geq l \geq 0$, all a_i are atoms from some fixed vocabulary $\mathcal{U}$, and "not" denotes *default negation*. Recall that safety means that all variables occurring in the head or negative body also occur in the positive body. A rule or program containing no variables is *ground*. The *grounding* of a program P relative to a set C of constants is defined as usual and denoted by $grd(P, C)$. Programs containing only atoms of arity 0 are called *propositional*. Following custom, we will identify ground programs with propositional ones. The set of all atoms occurring in a program P is denoted by $At(P)$. By a *fact*, we understand a rule of form $a \leftarrow$, usually just written a. Note that facts must be ground. Given a finite set A of ground atoms, the power set, 2^A, of A thus represents the set of all programs containing facts from A only.

Following Gelfond and Lifschitz [8], an interpretation I (i.e., a set of ground atoms) is an *answer set* of a ground program P iff it is a minimal model of the *reduct* P^I, resulting from P by (i) deleting all rules containing a default-negated atom $\mathrm{not}\ a$ such that $a \in I$, and (ii) deleting all default-negated atoms in the remaining rules. The answer sets of a non-ground program are given by the answer sets of the grounding over its Herbrand universe. The set of all answer sets of a program P is denoted by $\mathcal{AS}(P)$.

We continue with recapitulating the relevant program correspondence notions. For this, we consider propositional programs only in what follows as cc$\top$ deals with just these kinds of programs. To begin with, for collections $\mathcal{S}, \mathcal{S}'$ of sets of ground atoms, a set B of ground atoms, and $\odot \in \{\subseteq, =\}$, we define $\mathcal{S} \odot_B \mathcal{S}'$ as $\{Y \cap B \mid Y \in \mathcal{S}\} \odot \{Y \cap B \mid Y \in \mathcal{S}'\}$. Following Oetsch et al. [6], a *propositional query inclusion problem*, or *PQIP*, is a tuple of form $(P, Q, 2^A, \subseteq_B)$ and a *propositional query equivalence problem*, or *PQEP*, is a tuple of form $(P, Q, 2^A, =_B)$, where P and Q are two programs and A and B are sets of atoms, intuitively referring to sets of input and output atoms, respectively. We say that $(P, Q, 2^A, \odot_B)$ *holds*, for $\odot \in \{\subseteq, =\}$, iff, for each set of facts $F \in 2^A$, $AS(P \cup F) \odot_B AS(Q \cup F)$. For a PQEP $\Pi = (P, Q, 2^A, =_B)$, the PQIPs $\Pi^{\rightarrow} = (P, Q, 2^A, \subseteq_B)$ and $\Pi^{\leftarrow} = (Q, P, 2^A, \subseteq_B)$ are *associated* with Π. Clearly, Π holds iff both $\Pi^{\rightarrow}$ and $\Pi^{\leftarrow}$ hold.

PQEPs express *ordinary equivalence*, *uniform equivalence* [3], and *strong equivalence* [2] as follows: If P and Q are formed over alphabet $\mathcal{U}$, then P and Q are (i) ordinarily equivalent iff the PQEP $(P, Q, \{\emptyset\}, =_{\mathcal{U}})$ holds, (ii) uniformly equivalent iff

$(P, Q, 2^{\mathcal{U}}, =_{\mathcal{U}})$ holds, and (iii) strongly equivalent iff $(P, Q, \mathcal{P}_{\mathcal{U}}, =_{\mathcal{U}})$ holds, where $\mathcal{P}_{\mathcal{U}}$ is the set of all programs over $\mathcal{U}$.

Concerning the complexity of PQIPs and PQEPs, as shown previously [6], given programs $P, Q \in \mathcal{P}_{\mathcal{U}}$, sets $A, B \subseteq \mathcal{U}$ of atoms, and $\odot \in \{\subseteq, =\}$, deciding whether $(P, Q, 2^A, \odot_B)$ holds is Π_3^P-complete. Moreover, the problem is Π_2^P-complete in case $B = \mathcal{U}$. Both hardness results hold even for arbitrary but fixed A.

Quantified propositional logic. The complexity results above show that PQIPs and PQEPs can be efficiently reduced to *quantified propositional logic*, an extension of classical propositional logic in which formulas are permitted to contain quantifications over propositional variables. Such formulas are also called *quantified Boolean formulas* (QBFs); we denote them by upper-case Greek letters. Similar to predicate logic, $\exists$ and $\forall$ are used as symbols for existential and universal quantification, respectively.

For an interpretation I and a QBF Φ, the relation $I \models \Phi$ is defined analogously as in classical propositional logic, with the additional conditions that $I \models \exists p \Psi$ iff $I \models \Psi[p/\top]$ or $I \models \Psi[p/\bot]$, and $I \models \forall p \Psi$ iff $I \models \Psi[p/\top]$ and $I \models \Psi[p/\bot]$, for $\Phi = \mathsf{Q} p \Psi$ with $\mathsf{Q} \in \{\exists, \forall\}$, where $\Psi[p/\phi]$ denotes the QBF resulting from Ψ by replacing each free occurrence of p in Ψ by ϕ.[5] Satisfiability and validity of a QBF are defined analogously as for formulas in classical propositional logic. Note that for closed QBFs it holds that the notions of satisfiability and validity coincide.

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 $P = \{p_1, \ldots, p_n\}$. A QBF Φ is said to be in *prenex normal form* (PNF) iff it is closed and of the form $\mathsf{Q}_n P_n \ldots \mathsf{Q}_1 P_1 \, \phi$, where $n \geq 0$, ϕ is a propositional formula, and $\mathsf{Q}_i \in \{\exists, \forall\}$ such that $\mathsf{Q}_i \neq \mathsf{Q}_{i+1}$ for $1 \leq i \leq n-1$. Moreover, if ϕ is in conjunctive normal form, then Φ is in *prenex conjunctive normal form* (PCNF), and if ϕ is in disjunctive normal form, then Φ is in *prenex disjunctive normal form* (PDNF). A QBF $\Phi = \mathsf{Q}_n P_n \ldots \mathsf{Q}_1 P_1 \, \phi$ 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 Φ corresponds to a quantifier occurrence in the prenex normal form.

Well-known complexity results for the evaluation problem of QBFs imply that PQIPs and PQEPs can be efficiently reduced to $(3, \forall)$-QBFs. These reductions are the central theoretical basis for cc$\top$ and are discussed next.

3 Underlying Translations of cc$\top$

In this section, we recapitulate the basic encodings for mapping PQIPs and PQEPs into QBFs [6] and introduce a slightly simplified encoding.

We start with some notation and ancillary definitions. Given a set V of atoms, we assume (pairwise) disjoint copies $V^i = \{v^i \mid v \in V\}$, for every $i \geq 1$. Furthermore, we define $(V^i \leq V^j)$ as $\bigwedge_{v \in V}(v^i \rightarrow v^j)$, $(V^i < V^j)$ as $(V^i \leq V^j) \wedge \neg(V^j \leq V^i)$, and $(V^i = V^j)$ as $(V^i \leq V^j) \wedge (V^j \leq V^i)$. Loosely speaking, these operators allow to compare different subsets of atoms from a common set V under subset inclusion, proper-subset inclusion, and equality, respectively.

[5] The notion of a free variable occurrence is defined similarly as in predicate logic.

We use superscripts as a general renaming scheme for formulas and rules. That is, for each $i \geq 1$, α^i expresses the result of replacing each occurrence of an atom v in α by v^i, where α is any formula or rule. For a rule r of form (1), we define $H(r) = a_1 \vee \dots \vee a_l$, $B^+(r) = a_{l+1} \wedge \dots \wedge a_m$, and $B^-(r) = \neg a_{m+1} \wedge \dots \wedge \neg a_n$. We identify empty disjunctions with $\bot$ and empty conjunctions with $\top$.

In order to express properties of logic programs and respective program reducts in the language of quantified propositional logic, we introduce the following concept: Given a propositional program P, define $P^{\langle i,j \rangle} = \bigwedge_{r \in P} \big((B^+(r^i) \wedge B^-(r^j)) \rightarrow H(r^i)\big)$. Then, for any propositional program P with $At(P) = V$, any interpretation I, and any $X, Y \subseteq V$ such that, for some $i, j \geq 0$, $I \cap V^i = X^i$ and $I \cap V^j = Y^j$, it holds that $X \models P^Y$ iff $I \models P^{\langle i,j \rangle}$ [9].

We are now in a position to state the encoding due to Oetsch et al. [6].

Proposition 1. *Let $\Pi = (P, Q, 2^A, \subseteq_B)$ be a PQIP, $At(P \cup Q) = V$, $A, B \subseteq V$, and*

$$\mathsf{S}[\Pi] = \forall V^1 \forall A^2 \neg\big(\Phi_\Pi \wedge \forall V^4\big((B^4 = B^1) \rightarrow \Psi_\Pi\big)\big), \text{ where}$$
$$\Phi_\Pi = P^{\langle 1,1 \rangle} \wedge (A^2 \leq A^1) \wedge \forall V^3\Big(\big((A^2 \leq A^3) \wedge (V^3 < V^1)\big) \rightarrow \neg P^{\langle 3,1 \rangle}\Big) \text{ and}$$
$$\Psi_\Pi = \Big(\big(Q^{\langle 4,4 \rangle} \wedge (A^2 \leq A^4)\big) \rightarrow \exists V^5\big(\big((A^2 \leq A^5) \wedge (V^5 < V^4)\big) \wedge Q^{\langle 5,4 \rangle}\big)\Big).$$

Then, Π holds iff $\mathsf{S}[\Pi]$ is valid. Moreover, a PQEP $\Omega = (P, Q, 2^A, =_B)$ holds iff $\mathsf{S}[\Omega^{\rightarrow}] \wedge \mathsf{S}[\Omega^{\leftarrow}]$ is valid.

Besides the above encoding $\mathsf{S}[\cdot]$, cc$\top$ implements a slightly adapted version which we introduce next. The key observation for the subsequent adaption is that we use a *fixed assignment* for atoms in view of the subformula $B^4 = B^1$ of $\mathsf{S}[\cdot]$. Hence, for the quantifier block $\forall V^4$, it is sufficient to take only atoms from $V^4 \setminus B^4$ into account and replace all occurrences of atoms $v^4 \in B^4$ by v^1 within the remaining part of the formula. We thus obtain:

Theorem 1. *Let $\Pi = (P, Q, 2^A, \subseteq_B)$ be a PQIP, $At(P \cup Q) = V$, and $A, B \subseteq V$. Then, Π holds iff $\mathsf{T}[\Pi] = \forall V^1 \forall A^2 \neg\big(\Phi_\Pi \wedge \forall (V^4 \setminus B^4)\, \Psi_\Pi[B^4/B^1]\big)$ is valid, where Φ_Π and Ψ_Π are the QBFs from Proposition 1 and $\Psi_\Pi[B^4/B^1]$ is the result of replacing all occurrences of atoms $v^4 \in B^4$ in Ψ_Π by v^1.*

Obviously, all encodings introduced so far, are (i) always linear in the size of P, Q, A, and B, and (ii) possess at most two quantifier alternations in any branch of the formula tree. The latter shows that any such encoding is easily translated into a $(3, \forall)$-QBF. Thus, the complexity of evaluating these QBFs is not harder than the complexity of the encoded decision problems, which shows the adequacy of the encodings in the sense of Besnard et al. [10]. The benefit of the refined encodings is, however, that the number of universally quantified variables is reduced—in fact, in some specific cases, one quantifier block even vanishes. This guarantees adequacy also for some special cases of query problems with lower complexity. Note that by a proper parameterisation of a PQIP (resp., PQEP) also some important special cases of correspondence checking can be realised, e.g., uniform equivalence and ordinary equivalence. It can easily be verified that all special cases without projection have in common that the resulting encodings based on $\mathsf{T}[\cdot]$ yield QBFs with at most one quantifier alternation in each branch of the formula tree, witnessing their Π_2^P-membership.

4 Dress Rehearsal: Some Preliminary Performance Evaluation

In this section, we present a preliminary experimental evaluation of our implementation, in order to assess the behaviour of cc$\top$ under different QBF solvers, different encodings, and different problem settings in terms of runtime performance.

In the spirit of previous experiments with cc$\top$ [1], we use a reduction from QBFs to PQIPs as given by the Π_3^P-hardness proof for deciding PQIPs [6]. This provides us with a class of random benchmark problems for cc$\top$ which is easily parameterisable and which reflects, in some sense, the inherent hardness of the problem. More precisely, the method is as follows: (i) generate a random $(3, \forall)$-QBF Φ in PDNF; (ii) reduce Φ to a PQIP $\Pi_\Phi = (P, Q, 2^A, \subseteq_B)$ such that Φ is valid iff Π_Φ holds [6]; and (iii) apply cc$\top$ to derive the corresponding encoding $\mathsf{S}[\Pi_\Phi]$ or $\mathsf{T}[\Pi_\Phi]$. A particular advantage of this method is that it allows in a straightforward way to verify the correctness of the overall system: just check whether Φ and one of $\mathsf{S}[\Pi_\Phi]$ or $\mathsf{T}[\Pi_\Phi]$ have the same truth value. Indeed, with the help of this feature, we were able to find errors in some QBF solvers.

Our benchmark set consists of 1000 instances. The randomly generated QBFs of Step (i) contain 24 different atoms each. From those 24 atoms, each quantifier block binds 8 of them. Each term in the PDNF contains 4 atoms which are selected randomly among the 24 atoms and are negated with probability 0.5. The whole formula consists of 38 terms. From the 1000 instances, 506 evaluate to true and 494 evaluate to false. Thus, the ratio between true and false instances is close to 1 which indicates that the instances are neither under-constrained nor over-constrained. From each Φ, we construct a PQIP $\Pi_\Phi = (P, Q, 2^A, \subseteq_B)$ such that Φ is true iff Π_Φ holds. Note that P, Q, and B are determined by the reduction but the context A can be chosen arbitrarily.

For our experiments, we use three different settings, viz. the empty context $A = \emptyset$, the full context $A = \mathcal{U}$, and an in-between setting $\emptyset \subseteq A \subseteq \mathcal{U}$. For the last setting, each atom occurring in one of the two programs P and Q is in A with probability 0.5. We consider both encodings from PQIPs to QBFs, $\mathsf{S}[\cdot]$ and $\mathsf{T}[\cdot]$, together with the three settings for the context. We compare the QBF solvers `semprop` [11] (release 24/02/02), `qube-bj` [12] (v1.2), `quantor` [13] (release 25/01/04), and `qpro` [14], all of them showed to be competitive in previous QBF evaluations. The solvers `qpro`, `qube-bj`, and `semprop` are based on the standard DPLL decision procedure extended by special learning techniques whereas `quantor` implements a combination of resolution and variable expansion. All solvers except `qpro` require the input to be in prenex conjunctive normal form. Thus, for those solvers, an intermediate prenexing step is necessary. All experiments were carried out on a 3.0 GHz Dual Intel Xeon workstation, with 4 GB of RAM and Linux version 2.6.8.

Figure 1 summarises the results of the comparison. The different QBF solvers, encodings ($\mathsf{S}[\cdot]$, $\mathsf{T}[\cdot]$), and settings for the context (empty, half-full, full, respectively) are given on the abscissa, and the median runtimes in seconds are depicted on the ordinate.

Observe that the alternative encoding $\mathsf{T}[\cdot]$ does not achieve faster runtimes for all solvers, although it uses less variables. For `qpro` and `qube-bj`, QBFs from $\mathsf{T}[\cdot]$ are solved—as one would expect—faster. This is not the case for `semprop` and `quantor`, where `semprop` solves QBFs from $\mathsf{S}[\cdot]$ slightly faster and `quantor` solves them much faster. The median runtime for `quantor` with full context and encoding $\mathsf{T}[\cdot]$ is even greater than 100 seconds. Concerning the influence of the context parameterisation on

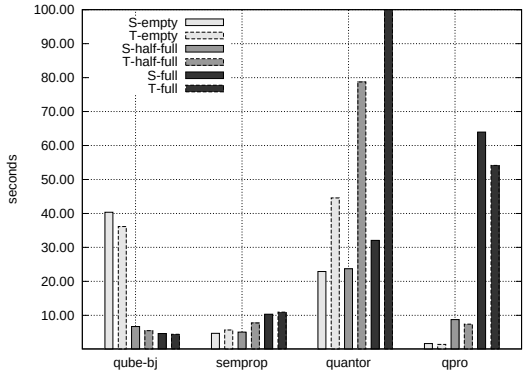

Fig. 1. Median runtimes for different solvers, encodings, and problem settings.

the runtimes, the non-normal-form solver `qpro` achieves best results for the empty context but rather poor results for the full context. For `qube-bj` the contrary is true, however, i.e., it achieves best results for the full context but poor results for the empty context—a quite surprising observation. Finally, the most robust solver in this aspect is `semprop`. Recall that each of the derived PQIPs $(P, Q, 2^A, \subseteq_B)$ either holds for any A, or does not hold for any A. The assignments of atoms from X^1 in our encodings which "guess" context-program candidates are thus irrelevant for the truth value of the QBFs. As `qpro` does not implement any heuristics concerning the selection of atoms, it is not surprising that runtimes scale exponentially with respect to the context. Interestingly, the runtimes for `qube-bj` even worsens without those "decoy" variables.

The results in Fig. 2 provide some deeper insights concerning the runtime behaviour of the non-normal-form solver `qpro` and the normal-form solvers `semprop`, `qube-bj`, and `quantor`, respectively. For those graphs, the abscissa gives the runtime in seconds (scaled logarithmically) and the ordinate gives the number of solved problem instances. This means that for each runtime in the data we depict how many instances were solved with runtime less than or equal to that time. The different curves correspond to the different combinations of the chosen encoding and context parameterisation. For better legibility, different symbols are attached to the curves.

5 cc$\top$ on Stage: A Verification Application

We next discuss an application of cc$\top$ for verifying the correctness of certain programs. In particular, these programs represent the solutions of students as part of their assignments for a laboratory course on knowledge-based systems at our university. We compare these solutions relative to a reference program based on verifying certain PQEPs. As the involved programs are non-ground, we need special techniques to take this into account. Hence, our results demonstrate also how our reduction approach to QBFs can be applied to non-ground programs as well.

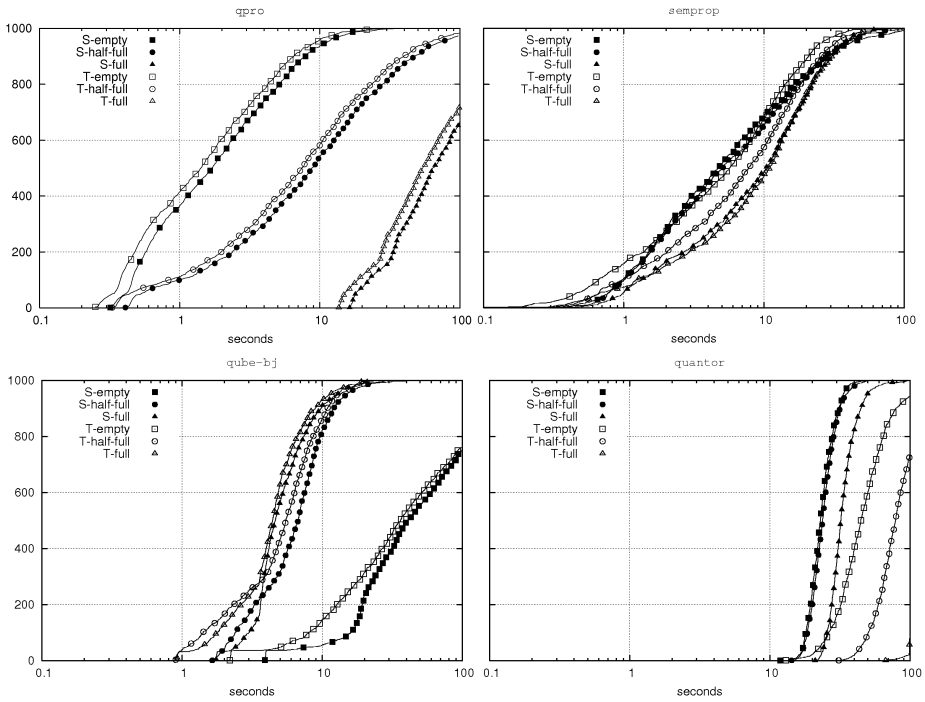

Fig. 2. Runtime distribution for `qpro`, `semprop`, `qube-bj`, and `quantor`.

One of the objectives of the course is to model a simple air-conditioning system by means of logic programs and, based on this model, to solve Reiter-style diagnosis tasks [15] with the dedicated diagnosis front-end of `DLV` [7]. The programs we consider here should represent the correct behaviour of the components of the air-condition system and are taken from three installments of the course between 2006 and 2008. The problem description slightly changed from year to year, yet Fig. 3 prototypically illustrates the specification of such an air-conditioning system. This system consists of four components, viz. a heater (`h`), a cooler (`c`), a switch (`s`), and a valve (`v`). They are connected by air lines (grey bars) and data lines (ordinary lines). The students' task is to model each component of the system as well as the connections between the components and some additional constraints required for diagnosing by respective programs. The problem description provides detailed specifications of the system and its components and defines the predicates to be used for the input and output of the single components and the whole system. The system's input airstream (`air_in`) is modeled by a temperature value, ranging from 0 to 60, and a value specifying whether or not air is streaming (`on` or `off`). The same holds for the output airstream (`air_out`). The input values of `threeway_in` is one of `cool`, `heat`, and `off`. Performance is regulated by the input value of `scale_in` which can be 0, 1, 2, or 3. The following specification determines the normal behaviour of the heater component:

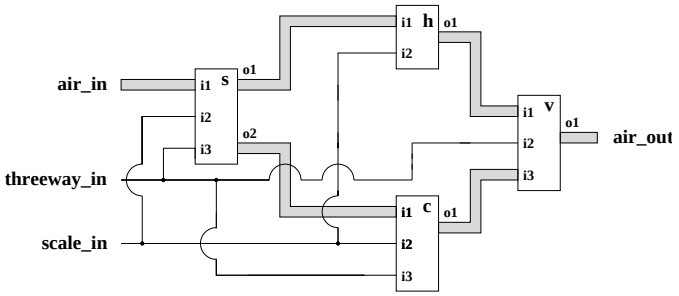

Fig. 3. Overall architecture of one particular air-conditioning system considered for a course.

> The heater warms-up the incoming airstream by three times the value set on the data line but only to a maximum value of 45. If the incoming airstream is already warmer than 45, then it is propagated unaltered to the heater's output. Air is streaming at the component's output iff it is streaming at the component's input. If it is not streaming at the output, the temperature is set to the value defined by a dedicated predicate `ambient_t`$(\cdot)$.

The following program represents this specification in `DLV` syntax. Note that `ab`$(\cdot)$ denotes the special predicate employed by `DLV` referring to a defective component.

```
% relates output airstream (on, off) with input airstream
s(H,o1,X) :- heater(H), s(H,i1,X), not ab(H).
% warming-up of the airstream according to the specification
t(H,o1,X) :-  heater(H), t(H,i1,Y), d(H,i2,Z), s(H,o1,on),
   A = Z*3, X = Y + A, X <= 45, not ab(H).
t(H,o1,45) :- heater(H), t(H,i1,Y), d(H,i2,Z), s(H,o1,on),
   A = Z*3, X = Y + A, X > 45, t(H,i1,T), T <= 45, not ab(H).
t(H,o1,T) :-  heater(H), t(H,i1,Y), d(H,i2,Z), s(H,o1,on),
   A = Z*3, X = Y + A, X > 45, t(H,i1,T), 45 < T, not ab(H).
% temperature of output airstream when air is not streaming
t(H,o1,X) :- heater(H), s(H,o1,off), ambient_t(X), not ab(H).
```

We will not go into further details but turn instead our attention to the considered verification tasks determining the correctness of the students' solutions.

Verification with cc$\top$. Let us denote by σ a specification that has to be represented by a logic program. Furthermore, let $Stud(\sigma)$ be a student's attempt to represent σ while $Ref(\sigma)$ is the reference solution. The possible input for the program specified by σ is assumed to be defined over a fixed set $I(\sigma)$ of ground predicates, and, similarly, the output is fixed by a set $O(\sigma)$ of ground predicates. Usually, specifications make further (implicit) assumptions concerning the input, e.g., some predicates need to be defined always or are restricted to be functional in some argument. We call a set $A(\sigma) \subseteq I(\sigma)$ satisfying such assumptions *admissible with respect to* σ.

Definition 1. *A program* $Stud(\sigma)$ *is* correct *with respect to a specification* σ *iff, for any admissible set* $A(\sigma) \subseteq I(\sigma)$, $AS(Stud(\sigma) \cup A(\sigma)) =_{O(\sigma)} AS(Ref(\sigma) \cup A(\sigma))$.

Note that both $Stud(\sigma)$ and $Ref(\sigma)$ are non-ground programs. For instance, for the heater specification from above, σ_H, the set $I(\sigma_H)$ contains the atoms `heater`(C),

$\texttt{ambient\_t}(T)$, $\texttt{d}(C, \texttt{i2}, Z)$, $\texttt{t}(C, \texttt{i1}, T)$, $\texttt{s}(C, \texttt{i1}, \texttt{on})$, and $\texttt{s}(C, \texttt{i1}, \texttt{off})$, while the set $O(\sigma_H)$ can be fixed to atoms $\texttt{t}(C, \texttt{o1}, T)$, $\texttt{s}(C, \texttt{o1}, \texttt{on})$, and $\texttt{s}(C, \texttt{o1}, \texttt{off})$, where T ranges from 0 to 60, $Z \in \{0, 1, 2, 3\}$, and $C \in \{\texttt{s}, \texttt{h}, \texttt{c}, \texttt{v}\}$. Any subset $A(\sigma_H) \subseteq I(\sigma_H)$ is admissible if $A(\sigma_H)$ contains exactly one predicate $\texttt{ambient\_t}(\cdot)$, and predicates $\texttt{t}(\cdot, \cdot, \cdot)$ and $\texttt{s}(\cdot, \cdot, \cdot)$ are functional in their third argument.

In order to apply cc$\top$ for our verification purpose, the overall strategy is to ground programs $\mathit{Stud}(\sigma)$ and $\mathit{Ref}(\sigma)$ and then to reduce the problem of program correctness to a PQEP. The reason why a reduction to standard uniform equivalence or ordinary equivalence (with additional guessing rules) is not feasible, is the necessity of answer-set projection which has two sources: first, programmers usually employ auxiliary atoms which are not considered as output predicates, and second, new atoms are sometimes added by the grounding procedure (we return to this point in a moment). In terms of complexity theory, projection is the reason why deciding PQEPs is exponentially harder than to decide problems of ordinary equivalence or uniform equivalence.

First, we outline how to handle the restriction to consider only admissible inputs. We express admissibility conditions, as exemplified above, by constraints. For instance, the admissibility conditions for $\texttt{ambient\_t}(\cdot)$ for the heater can be encoded by the following program:

```
def_ambient_t :- ambient_t(T).              :- not def_ambient_t.
:- ambient_t(T1), ambient_t(T2), T1 <> T2.
```

In general, we denote by $C(\sigma)$ the program representing admissibility constraints on the input according to specification σ. The following result establishes the connection between program correctness and PQEPs:

Theorem 2. *$\mathit{Stud}(\sigma)$ is correct with respect to σ iff the PQEP $(P, Q, 2^A, =_B)$ holds, where $P = \mathit{grd}(\mathit{Stud}(\sigma) \cup C(\sigma), D)$, $Q = \mathit{grd}(\mathit{Ref}(\sigma) \cup C(\sigma), D)$, $A = I(\sigma)$, $B = O(\sigma)$, and D is a finite set containing all constants in $\mathit{Stud}(\sigma) \cup \mathit{Ref}(\sigma) \cup C(\sigma) \cup I(\sigma)$.*

Verifying students' solutions by following the above theorem and then applying cc$\top$ is in principle possible since our domain is finite, but the resulting programs would be prohibitively large. So, instead of applying a naive grounding by strictly following the definition, we make use of the *intelligent grounding component* of DLV. This means that several optimisations are performed, e.g., input rewriting, deletion of rules whose body is always false, and semi-naive evaluation. The choice of enabled options has significant impact on the runtimes of the subsequently employed QBF solver, however. We also remark that some optimisations, e.g., the input rewriting, introduce new atoms. Thus, not only auxiliary atoms used by a programmer but also such new atoms stemming from the grounding request the use of projection in equivalence tests. Note that by using DLV's intelligent grounder, we can use strong negation as well as integer arithmetics and comparison predicates in the programs. The grounder translates these constructs such that they do not occur in the ground programs.

However, the optimisations of the intelligent grounder may be too excessive. For example, the grounding of the program for the heater above would result in the empty program since there are no facts, and therefore the bodies of the rules are always false. Our concrete method to ground programs is as follows: Let P be a program and σ its underlying specification. First, augment P by rules $a \leftarrow a'$ and $a' \vee a''$ for any $a \in I(\sigma)$,

where a' and a'' are globally new atoms. Then, ground the augmented version of P. Finally, delete all rules containing primed or double-primed atoms from the resulting program. This method guarantees that the semantics of the ground program, possibly joined with atoms from $I(\sigma)$, is correctly preserved under the conservative assumption that the grounder only preserves ordinary equivalence.

However, the resulting programs are still too large. We thus sacrifice completeness of the verification problems by restricting the sets $I(\sigma)$ to contain only certain relevant predicates. For the heater specification σ_H from above for example, we restrict $I(\sigma_H)$ in such a way that not all temperature values from 0 to 60 are considered but only an interval around 45 since it is very likely that if a student program is not correct, then it will diverge from the specification on input from this interval.

Results of the verification. As already mentioned, we considered student data from three semesters. All experiments were carried out on a 3.0 GHz Quad Core Intel Xeon workstation, with 33 GB of RAM and SuSE Linux version 10.3. We used the QBF solver `qpro` with encoding $\mathsf{T}[\cdot]$, as it turned out that all other solvers mentioned in the previous section showed a runtime behaviour several orders of magnitude worse than `qpro`'s. Concerning the setting for the grounder, we achieved best performance when the option for input rewriting was disabled. The reason is that this optimisation introduces new atoms which seems to be disadvantageous for `qpro`.

We also compared the outcomes of the equivalence tests with results from a test approach currently used in the course. In the latter, test cases (admissible subsets of input predicates) are individually specified, and then it is tested whether a student's program and our reference program yield the same answer sets when joined with the test cases. Such sets of test cases usually comprise 10 to 20 instances. As it will turn out, many errors were undetected by our current approach, thus it is rather prone to false-positives with respect to the verification task.

Table 1 summarises the results of our experiments. We provide the year of the semester a course took place, the name of the component we considered, the number of instances of that component, the number of instances classified as correct by the current approach, the number of instances classified as correct by our reduction approach, the average runtime in seconds, as well as the median runtime for solving the QBFs. Components `c`, `h`, `s`, `v` denote the cooler, heater, switch, and valve as before, while 'all' refers to the overall program consisting of all components, the encoding of the connections between them, and additional constraints required for diagnosing. The ground programs for the component tests contain up to 985 rules. The number of variables in the resulting QBFs ranges from 229 to 623. For the overall tests, programs contain up to 4818 rules, and the QBFs contain 949 to 3143 variables. Note that whenever a program is classified as not correct by the current approach, then it is classified as not correct by the cc$\top$ approach as well. Hence, the difference between the numbers of programs classified as correct by the two approaches is the number of false positives for the current approach. Table 1 shows that runtimes for the solved QBFs keep in reasonable bounds. Coming as no surprise, the cc$\top$ approach reveals significantly more incorrect solutions than the current approach. The reason that the number of correct overall programs is not smaller than the minimum number of its correct components is mainly due to different restrictions on what admissible input means. The two significantly small numbers of correct

Table 1. Outcomes of the program verification.

semester	component	number of	classified as correct		runtimes	
		instances	current approach	cc$\top$ approach	average	median
ws2006	c	50	44	38	0.9	1.0
	h	50	39	32	1.0	1.1
	s	50	29	22	0.4	0.1
	v	50	40	34	5.1	5.6
	all	50	42	32	70.2	103.0
ws2007	c	78	67	56	0.8	0.8
	h	78	69	59	0.6	0.6
	s	78	52	0	1.4	1.5
	v	78	48	8	4.5	2.9
	all	78	60	39	491.4	894.0
ws2008	c	100	54	40	1.3	2.3
	h	100	70	13	0.2	0.2
	s	100	59	28	1.4	3.0
	v	100	53	25	0.6	1.1
	all	100	52	19	132.3	72.5

solutions for the switch and the valve component in the 'ws2007' test set is because of subtle differences between the reference and the student solutions in case some input values are missing. If this is considered to be too strict, one can simply exclude such cases by changing the admissibility constraints accordingly.

6 Conclusion

In this paper, we discussed how correspondence problems which allow to restrict the alphabet of the context class and which facilitate the removal of auxiliary atoms in the comparison—two important concepts for program equivalence in practice—can be used in a concrete scenario. Moreover, though cc$\top$ processes propositional programs only, it can still be employed for program comparisons of non-ground programs. We recapitulated some details of the tool, based on an efficient reduction to QBFs, discussed one particular optimisation, and analysed experiments with different QBF solvers on a random benchmark set which reveals interesting differences of the solvers depending on the particular problem parameterisation and the choice of the encoding. More relevantly, we considered an application concerning the verification of programs.

There remain many issues for future work. For model-based diagnosis, native concepts of equivalence, directly defined in terms of a diagnosis problem, would be useful. In case programs are not equivalent, a counterexample that gives information why the programs are not equivalent would be of great value. cc$\top$ can be used to generate QBFs such that assignments for the open variables correspond to such counterexamples. However, few solvers can compute such assignments—to extend `qpro` in this way is future work. Also, often non-ground programs are formulated over a language with an infinite domain. An important topic is to single out at least sufficient conditions when we

can restrict this domain to a finite subdomain such that program equivalence over this subdomain implies equivalence over the unrestricted domain.

Concerning related work, we mention the system `DLPEQ` [16] for deciding ordinary equivalence, which is based on a reduction to logic programs, and the system `SELP` [17] for checking strong equivalence, which is based on a reduction to classical logic. Strong equivalence between non-ground programs can be decided by a dedicated system that is based on a reduction to a decidable fragment of first-order logic [18].

References

1. Oetsch, J., Seidl, M., Tompits, H., Woltran, S.: ccT: A Tool for Checking Advanced Correspondence Problems in Answer-Set Programming. In: Proc. CIC 2006, IEEE (2006) 3–10
2. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Transactions on Computational Logic **2**(4) (2001) 526–541
3. Eiter, T., Fink, M.: Uniform Equivalence of Logic Programs under the Stable Model Semantics. In: Proc. ICLP 2003. Vol. 2916 of LNCS, Springer (2003) 224–238
4. Eiter, T., Tompits, H., Woltran, S.: On Solution Correspondences in Answer Set Programming. In: Proc. IJCAI 2005. (2005) 97–102
5. Woltran, S.: Characterizations for Relativized Notions of Equivalence in Answer Set Programming. In: Proc. JELIA 2004. Vol. 3229 of LNCS, Springer (2004) 161–173
6. Oetsch, J., Tompits, H., Woltran, S.: Facts do not Cease to Exist Because They Are Ignored: Relativised Uniform Equivalence with Answer-Set Projection. In: Proc. AAAI 2007, AAAI Press (2007) 458–464
7. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: The Diagnosis Frontend of the DLV System. AI Communications **12**(1-2) (1999) 99–111
8. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
9. Tompits, H., Woltran, S.: Towards Implementations for Advanced Equivalence Checking in Answer-Set Programming. In: Proc. ICLP 2005. Vol. 3668 of LNCS, Springer (2005) 189–203
10. Besnard, P., Schaub, T., Tompits, H., Woltran, S.: Representing Paraconsistent Reasoning via Quantified Propositional Logic. In: Inconsistency Tolerance. Vol. 3300 of LNCS, Springer (2005) 84–118
11. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: Proc. TABLEAUX 2002. Vol. 2381 of LNCS, Springer (2002) 160–175
12. Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for Quantified Boolean Logic Satisfiability. Artificial Intelligence **145** (2003) 99–120
13. Biere, A.: Resolve and Expand. In: Proc. SAT 2004. Vol. 3542 of LNCS, Springer (2005) 59–70
14. Egly, U., Seidl, M., Woltran, S.: A Solver for QBFs in Negation Normal Form. Constraints **14**(1) (2009) 38–79
15. Reiter, R.: A Theory of Diagnosis from First Principles. Artificial Intelligence **32**(1) (1987) 57–95
16. Oikarinen, E., Janhunen, T.: Verifying the Equivalence of Logic Programs in the Disjunctive Case. In: Proc. LPNMR 2004. Vol. 2923 of LNCS, Springer (2004) 180–193
17. Chen, Y., Lin, F., Li, L.: SELP - A System for Studying Strong Equivalence Between Logic Programs. In: Proc. LPNMR 2005. Vol. 3662 of LNCS, Springer (2005) 442–446
18. Eiter, T., Faber, W., Traxler, P.: Testing Strong Equivalence of Datalog Programs - Implementation and Examples. In: Proc. LPNMR 2005. Vol. 3662 of LNCS, Springer (2005) 437–441