Automated Verification of Equivalence Properties in Advanced Logic Programs

Jan Heuer[1]

Abstract: During the development process of an Answer Set Programming (ASP) encoding, it would be desirable to have a tool that can automatically verify whether one subprogram can safely be replaced by another one. Formally this corresponds to the problem of verifying the *strong equivalence* of two programs. The translation tool ANTHEM was developed to be used in conjunction with an *automated theorem prover* to verify strong equivalence. The current version of ANTHEM only works on *positive programs* with a restricted input language. This paper[2] extends the translation in ANTHEM to support a larger subset of the input language, namely *negation*, *simple choices*, and *pools*. The support of negation requires an additional translation step that maps formulas from the logic of here-and-there to classical logic. Furthermore, some automated theorem provers are compared for their ability to verify the strong equivalence of logic programs.

Keywords: Answer Set Programming; Automated Theorem Proving; Formal Verification; Logic Programming

1 Introduction

In recent years, many industrial applications have been built using Answer Set Programming [EGL16, Fa18]. Because of this, the need for formal verification tools for Answer Set Programs has increased. One possible application of formal verification is to decide whether a modified version of a program can replace the original without changing the behaviour on the application domain. This task is made even more complex as, in other programming paradigms, it is customary to split an application into smaller program parts. In order to decide if the optimised version of the program can replace the original one, one has to verify whether the new program works together correctly with the *other program parts* on *any input*. This idea is captured by the concept of *strong equivalence* between two logic programs [LPV01], which is the question whether a program can replace another one in any context without modifying the semantics.

In order to verify this question, the translation tool ANTHEM was developed [LLS19]. Using ANTHEM in conjunction with an *automated theorem prover* enables the automated verification

[1] University of Potsdam jan.heuer@uni-potsdam.de
Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – 457292495; funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 457292495.

[2] This paper is a shortened version of the authors bachelor thesis [He20] which can be found at:
`https://github.com/janheuer/anthem/files/11896360/thesis.pdf`

of strong equivalence of two logic programs. The translation of ANTHEM is based on the translation τ used to define the semantics of the input language of GRINGO [Ge15]. However, τ transforms logic programs into infinitary propositional formulas, which are not suitable for automated theorem proving. Therefore, ANTHEM instead uses a new translation τ^*, which transform programs into finite first-order formulas. But the output produced by ANTHEM has the semantics of the *logic of here-and-there* whereas automated theorem provers work with the semantics of *classical logic*. In the special case that the input programs are *positive programs*[3] the two semantics coincide, which makes the usage of ANTHEM possible.

To illustrate this restriction let us look at the following example of two logic programs defining a transitive relation p. The first program uses a simple choice rule[4] to define the relation p and then uses a basic rule to make the relation transitive. The second program again uses a choice rule to define the relation. However, to ensure that this relation is transitive a constraint is used this time.

Program 1

```
{ p(X,Y) } :- q(X), q(Y).
p(X,Y) :- p(X,Z), p(Z,Y),
          q(X), q(Y), q(Z).
```

Program 2

```
{ p(X,Y) } :- q(X), q(Y).
:- p(X,Z), p(Z,Y), not p(X,Y),
   q(X), q(Y), q(Z).
```

It is not possible to verify the strong equivalence of these two programs using the current version of ANTHEM. A warning message is produced, stating that the inputs are *non-positive programs*, from which we can conclude that the semantics of the output is the logic of here-and-there.

Furthermore, the input language supported by ANTHEM is restricted. For example it does not support the usage of pools as in the fact `c(r;g;b).` to succinctly express multiple values to succinctly the multiple facts `c(r). c(g). c(b).`

The goal of this work[2] is to overcome these current limitations of ANTHEM to enable the verification of the strong equivalence of programs containing the features mentioned above. The main obstacle to achieving this is to map the semantics of the logic of here-and-there to the semantics of classical logic. To do so the transformation σ^* is presented in Sect. 3, which is based on a transformation defined in [PTW01]. Furthermore, this work extends the translation τ^* implemented by ANTHEM to handle programs containing pools in Sect. 4. Both the extended translation τ^* and the transformation σ^* are implemented in a new version of ANTHEM[5] described in Sect. 5. As a secondary goal, we compare the ability of several automated theorem provers to verify strong equivalence problems generated by ANTHEM in Sect. 6.

[3] That is programs consisting of rules with a single atom in the head and no negation in the body.

[4] For details on choice rules see [Ge15, Ge19]. In this work we only consider simple choice, i.e. rules of the form `{p(X)} :- q(X).` which correspond to the formula $q(X) \to p(X) \vee \neg p(X)$.

[5] https://github.com/janheuer/anthem/releases/tag/v0.3

2 Background

The logic of here-and-there is an intermediate logic between classical and intuitionistic logic. It was first connected to answer set programming by [Pe97], who introduced a new logical characterisation of answer sets as a form of minimal models in the logic of here-and-there. It uses a standard propositional language built from a set of *atoms* $\mathcal{P}$ as well as the usual logical symbols $\bot$, $\neg$, $\wedge$, $\vee$, and $\rightarrow$. The *logical complexity* of a formula ϕ, written as $lc(\phi)$, is the number of occurrences of the logic symbols $\neg$, $\wedge$, $\vee$, and $\rightarrow$ in ϕ. An *HT-interpretation* is an ordered pair $\langle H, T\rangle$ of sets of atoms such that $H \subseteq T$.

The definition of satisfiability in the logic of here-and-there is in parts analogous to classical logic; however, it significantly differs in the definition of negation and implication.

Definition 1 *The satisfiability relation $\models$ is recursively defined as follows:*

- $\langle H,T\rangle \not\models \bot$,
- *for any atom* $p \in \mathcal{P}$, $\langle H,T\rangle \models p$ *if* $p \in H$,
- $\langle H,T\rangle \models \phi_1 \wedge \phi_2$ *if* $\langle H,T\rangle \models \phi_1$ *and* $\langle H,T\rangle \models \phi_2$,
- $\langle H,T\rangle \models \phi_1 \vee \phi_2$ *if* $\langle H,T\rangle \models \phi_1$ *or* $\langle H,T\rangle \models \phi_2$,
- $\langle H,T\rangle \models \neg\phi$ *if* $\langle T,T\rangle \not\models \phi$,
- $\langle H,T\rangle \models \phi_1 \rightarrow \phi_2$ *if* $\langle H,T\rangle \models \phi_1$ *implies* $\langle H,T\rangle \models \phi_2$ *and*
  $\langle T,T\rangle \models \phi_1$ *implies* $\langle T,T\rangle \models \phi_2$.

Note that $\langle T,T\rangle \models \phi$ is equivalent to $T \models \phi$ (in classical logic). The strong equivalence of two logic programs is defined as follows [LPV01]:

Definition 2 *Two programs,* Π_1 *and* Π_2*, are strongly equivalent if and only if for every program* Π*, the answer sets of* $\Pi_1 \cup \Pi$ *and* $\Pi_2 \cup \Pi$ *are the same.*

The following theorem from [LPV01, Theorem 1] reduces the question of strong equivalence to a satisfiability problem in the logic of here-and-there.

Theorem 1 *Two programs,* Π_1 *and* Π_2 *are strongly equivalent if and only if their representations as propositional formulas are equivalent in the logic of here-and-there.*

3 Expressing the Semantics of the Logic of Here-And-There in Classical Logic

The main idea of expressing satisfiability in the logic of here-and-there in classical logic is to introduce a new set of atoms which represent the values of formulas in the "there" world.

Formally, given a set of atoms $\mathcal{P}$, a new set of atoms $\mathcal{P}'$ is introduced as $\mathcal{P}' = \{p' \mid p \in \mathcal{P}\}$, which is disjoint to $\mathcal{P}$. The formula ϕ' is the result of replacing every atom p in ϕ with p'. Given any set $T \subseteq \mathcal{P}$, T' is defined as $T' = \{p' \mid p \in T\}$. Intuitively the "unprimed" formulas represent the formulas evaluated in the "here" world, while the primed formulas represent the ones evaluated in the "there" world (i.e. the relation $\langle T, T\rangle \models \phi$).

Additionally, the condition $H \subseteq T$, i.e. every atom true "here" is also true "there", has to be encoded. This is done by adding the following set of formulas: $\mathcal{A} = \{p \rightarrow p' \mid p \in \mathcal{P}\}$. Satisfiability in the logic of here-and-there can then be expressed in classical logic by the following transformation (based on [PTW01, Definition 2]):

Definition 3 *Let ϕ be a formula. Then, $\sigma^*(\phi)$ is recursively defined as follows:*

- *if $\phi \in \mathcal{P} \cup \{\top, \bot\}$, then $\sigma^*(\phi) = \phi$,*
- *if $\phi = (\phi_1 \circ \phi_2)$, for $\circ \in \{\wedge, \vee\}$, then $\sigma^*(\phi) = \sigma^*(\phi_1) \circ \sigma^*(\phi_2)$,*
- *if $\phi = \neg\psi$, then $\sigma^*(\phi) = \neg\psi'$,*
- *if $\phi = (\phi_1 \rightarrow \phi_2)$, then $\sigma^*(\phi) = (\sigma^*(\phi_1) \rightarrow \sigma^*(\phi_2)) \wedge \left(\phi_1' \rightarrow \phi_2'\right)$.*

In order to state how the equivalence of two formulas in the logic of here-and-there can be expressed in classical logic, let us first consider two lemmas. The first lemma related interpretations in the logic of here-and-there to interpretations in classical logic.

Lemma 1 *There exists a 1-to-1 correspondence between interpretations $\langle H, T\rangle$ in the logic of here-and-there and classical interpretations I over the alphabet $\mathcal{P} \cup \mathcal{P}'$ such that $I \models \mathcal{A}$.*

This can easily be verified by setting $I = H \cup T'$ given a HT-interpretation $\langle H, t\rangle$, and setting $H = I \cap \mathcal{P}$ and $T = \{p \mid p \in I \cap \mathcal{P}'\}$ given a classical interpretation I, for details see [He20]. The second lemma relates satisfiability in the logic of here-and-there to satisfiability in classical logic.

Lemma 2 *Let ϕ be a formula. An HT-interpretation $\langle H, T\rangle$ satisfies ϕ, if and only if the classical interpretation $I = H \cup T'$ (over the alphabet $\mathcal{P} \cup \mathcal{P}'$) satisfies $\sigma^*(\phi)$.*

This can be verified by comparing Definition 3 and Definition 1, for details see [He20]. Finally, the theorem on expressing the equivalence of two formulas in the logic of here-and-there in classical logic can be stated:

Theorem 2 (HT-Equivalence in Classical Logic) *Let ϕ_1 and ϕ_2 be formulas. The formula $\phi_1 \leftrightarrow \phi_2$ is valid in the logic of here-and-there, if and only if the formula $\sigma^*(\phi_1) \leftrightarrow \sigma^*(\phi_2)$ is satisfied by every classical interpretation I over the alphabet $\mathcal{P} \cup \mathcal{P}'$ such that $I \models \mathcal{A}$.*

Proof. The formula $\phi_1 \leftrightarrow \phi_2$ is satisfiable in the logic of here-and-there iff for every HT-interpretation $\langle H,T\rangle$, $\langle H,T\rangle \models \phi_1$ iff $\langle H,T\rangle \models \phi_2$. Similarly, $\sigma^*(\phi_1) \leftrightarrow \sigma^*(\phi_2)$ is satisfied by every classical interpretation I iff for every classical interpretation I, $I \models \sigma^*(\phi_1)$ iff $I \models \sigma^*(\phi_2)$.

By Lemma 1 there is a 1-to-1 correspondence between interpretations $\langle H,T\rangle$ in the logic of here-and-there and classical interpretations I with $I \models \mathcal{A}$. By Lemma 2 $\langle H,T\rangle$ satisfies ϕ_i iff the corresponding classical interpretation I satisfies $\sigma^*(\phi_i)$, for $i = 1,2$. Therefore, $\langle H,T\rangle \models \phi_1$ iff $\langle H,T\rangle \models \phi_2$ is equivalent to $I \models \sigma^*(\phi_1)$ iff $I \models \sigma^*(\phi_2)$. □

4 Translating Logic Programs into Classical First-Order Formulas

4.1 Input Language

The definition of the input language in this section extends the input language of the previous version of ANTHEM [LLS19] by pooling[6]. Pooling is a feature similar to intervals. Both are shorthand notations allowing one to express a set of values in a single term. However, the difference is that intervals only make it possible to express a set of consecutive integers. With pools, it is possible to express non-consecutive integers as well as non-numerical values. Using pools in atoms in either the head or the body is again similar to using an interval, corresponding to a disjunction or conjunction respectively.

More formally, the following two alternatives are added to the recursive definition of *program terms* [LLS19, Sect. 2][7]:

- if $t_1, \dots, t_k$ are program terms then $(t_1, \dots, t_k)$ is a program term,
- if $t_1, \dots, t_k$ are program terms then $(t_1; \dots; t_k)$ is a program term.

The definition of an *atom* is extended by the following [LLS19, Sect. 2]:

- $p(\mathbf{t_1}; \dots; \mathbf{t_k})$ is an atom, where p is a symbolic constant and each $\mathbf{t_i}$ is a tuple of program terms.

4.2 Transforming Logic Programs into First-Order Formulas

The definition of the translation τ^* in this section corresponds to the definition from [LLS19, Sect. 6] and is extended to cover the new input language. We only present the new parts of the definitions, for the full definitions see [LLS19, Sect. 6]. The target language of this

[6] For more formal details on pools see [Ge15, Ge19].
[7] In [Ge15, Sect. 2.1] program terms are simply called terms.

translation is a standard first-order language with quantifiers. Notably, this language has variables of two sorts program variables and integer variables.

τ^* is defined using two translations τ^B and τ^H for the body and head of a rule respectively. Before defining these translations the formula $val_t(Z)$, expressing that Z is one of the values of the program term t, needs to be defined. This is necessary, as in the input language a term can express a set of values (e.g. by using an interval or a pool), whereas the target language does not include sets.

The following extends the definition of $val_t(Z)$ ([LLS19, Sect. 6]) by tuples and pools.

Definition 4 *For every program term t the formula $val_t(Z)$, where Z is a program variable that does not occur in t, is recursively defined as follows:*

- *if t is $(t_1, \ldots, t_k)$, then $val_t(Z)$ is*

$$\exists I_1 \ldots I_k (Z = (I_1, \ldots, I_k) \land val_{t_1}(I_1) \land \cdots \land val_{t_k}(I_k)),$$

- *if t is $(t_1; \ldots; t_k)$, then $val_t(Z)$ is*

$$\exists I_1 \ldots I_k (val_{t_1}(I_1) \land \cdots \land val_{t_k}(I_k) \land (Z = I_1 \lor \cdots \lor Z = I_k)),$$

where $I_1, \ldots, I_k$ are fresh program variables.

Next are the definition of the translations τ^B and τ^H which are applied to the expressions in the body and head of a rule respectively. They extend the definitions in [LLS19, Sect. 5] to handle atoms containing pools.

Definition 5

- $\tau^B(p(t_1; \ldots; t_k))$ *is* $\tau^B(p(t_1)) \lor \cdots \lor \tau^B(p(t_k))$,
- $\tau^B(\textit{not}\, p(t_1; \ldots; t_k))$ *is* $\tau^B(\textit{not}\, p(t_1)) \lor \cdots \lor \tau^B(\textit{not}\, p(t_k))$,
- $\tau^B(\textit{not not}\, p(t_1; \ldots; t_k))$ *is* $\tau^B(\textit{not not}\, p(t_1)) \lor \cdots \lor \tau^B(\textit{not not}\, p(t_k))$.

Definition 6

- $\tau^H(p(t_1; \ldots; t_k))$ *is* $\tau^H(p(t_1)) \land \cdots \land \tau^H(p(t_k))$,
- $\tau^H(\{p(t_1; \ldots; t_k)\})$ *is* $\tau^H(\{p(t_1)\}) \land \cdots \land \tau^H(\{p(t_k)\})$.

Finally, τ^* can be defined using these components.

Definition 7

$$\tau^*(H \leftarrow B_1 \wedge \cdots \wedge B_n)$$

is defined as the universal closure of the formula

$$\tau^B(B_1) \wedge \cdots \wedge \tau^B(B_n) \rightarrow \tau^H(H).$$

For any program Π, $\tau^*\Pi$ is the set of formulas τ^*R for every rule R in Π.

4.3 Expressing Strong Equivalence in Classical Logic

After applying τ^*, the resulting formulas will still have the semantics of the logic of here-and-there. Therefore, it is still not possible to verify the strong equivalence of two programs using theorem provers for classical first order logic. An exception is the case of two positive logic programs, as in that case, the semantics of the logic of here-and-there coincides with classical logic. This is the basis for the previous version of ANTHEM [LLS19].

However, with a first-order generalisation of the transformation σ^* given by Definition 3, it is possible to reduce the problem of verifying strong equivalence to classical first-order logic, by expressing the semantics of the formulas from the logic of here-and-there in classical logic. The first-order generalisation is given by the following definition extending Definition 3:

Definition 8 *Let ϕ be a formula. Then, $\sigma^*(\phi)$ is recursively defined as follows:*

- *if $\phi \in \mathcal{P} \cup \{\top, \bot\}$, then $\sigma^*(\phi) = \phi$,*
- *if $\phi = (\phi_1 \circ \phi_2)$, for $\circ \in \{\wedge, \vee\}$, then $\sigma^*(\phi) = \sigma^*(\phi_1) \circ \sigma^*(\phi_2)$,*
- *if $\phi = \neg\psi$, then $\sigma^*(\phi) = \neg\psi'$,*
- *if $\phi = (\phi_1 \rightarrow \phi_2)$, then $\sigma^*(\phi) = (\sigma^*(\phi_1) \rightarrow \sigma^*(\phi_2)) \wedge \left(\phi'_1 \rightarrow \phi'_2\right)$,*
- *if $\phi = QX\psi$, for $Q \in \{\forall, \exists\}$ then $\sigma^*(\phi) = QX\sigma^*(\psi)$.*

Using this generalised σ^*, the following theorem can be established[8] (for a proof see [He20]):

Theorem 3 *A program Π_1 is strongly equivalent to a program Π_2 if and only if $\sigma^*(\tau^*\Pi_1)$ is equivalent to $\sigma^*(\tau^*\Pi_2)$ in every classical interpretation I (over the alphabet $\mathcal{P} \cup \mathcal{P}'$) such that $I \models \mathcal{A}$.*

[8] We assume bot classical as well as HT-interpretations to interpret equality and arithmetic in the standard way.

5 Implementation

The extension of τ^* as well as the additional transformation σ^* are implemented in ANTHEM 0.3[9]. To obtain the Abstract Syntax Tree (AST) of the given input program ANTHEM makes use of the CLINGO API. The AST is then transformed into a first-order AST representing the first-order formulas using τ^*. During the transformation, a flag keeps track of the semantics of the formulas. If a rule contains either negation or a choice this flag is set to the semantics of the logic of here-and-there.

Depending on how the semantics-flag is set, after the transformation defined by τ^* terminates, σ^* is applied. Before transforming the formulas, the respective prime atoms and the prime axioms are created. The implementation of the actual transformation defined by σ^* (Definition 8) is done in two steps. First, all formulas are duplicated. Second, in the first copy all negated atoms are replaced by their "primed" variant, and in the second copy all atoms are replaced by their "primed" variant. This is a slight simplification of σ^* as defined in Definition 8. But it can easily be seen that the implemented σ^* is equivalent to Definition 8 (in the context of the considered input language).

Finally, ANTHEM outputs the obtained first-order formulas. If ANTHEM is used with one input program, ANTHEM prints all the formulas obtained from the rules in the program. If ANTHEM is called with two input programs a formula is constructed that expresses the strong equivalence of the two programs. Applied to programs containing negation or simple choices ANTHEM prints the info message

```
info: mapped to output semantics: classical logic
```

indicating that σ^* was applied to transform the formulas to be in the semantics of classical logic.

In order to verify the strong equivalence of two programs using automated theorem provers, ANTHEM provides the output format TPTP [Su09]. Specifically, ANTHEM uses the typed first-order form (TFF [Su12]) with integer arithmetic. The output of ANTHEM can then be used as the input to an automated theorem prover to verify the strong equivalence of the two input programs. Some options for automated theorem provers supporting the TFF language with integer arithmetic are compared in the next section.

6 Experimental Evaluation

6.1 Automated Theorem Provers

A large number of automated theorem provers support the TPTP language. However, the number of provers supporting the TPTP dialect TFF with integer arithmetic is significantly

[9] https://github.com/janheuer/anthem/releases/tag/v0.3

smaller. From the available options, some have also not been updated in a long time and trying to install them results in errors.

The following provers are tested in the remainder of this section: CVC4 version: 1.8[10], PRINCESS version: 2020-03-12[11], VAMPIRE version: 4.5.1[12], and ZIPPERPOSITION version: 1.5[13].

6.2 Example Logic Programs

Before comparing the theorem provers let us look at some example logic programs containing negation and simple choices (i.e. with just a single atom), which ANTHEM can translate to classical first-order logic by making use of σ^* (Sect. 4.3).

The first example replaces a program stating that `p` is true when `q` or the negation of `q` holds by a fact.

Program 1.A

```
p :- q.
p :- not q.
```

Program 1.B

```
p.
```

The two programs are not strongly equivalent, as by adding the rule `q :- p` Program 1.A does not have an answer set while Program 1.B has the answer set $\{p, q\}$. Therefore, all theorem provers are unable to verify the strong equivalence.

The next example shows that the previous two programs can be made strongly equivalent by adding the constraint `:- q`.

Program 2.A

```
p :- q.
p :- not q.
:- q.
```

Program 2.B

```
p.
:- q.
```

All theorem provers verify that the two programs are strongly equivalent in under a second.

Next we will consider two slightly longer programs. The final rule of the first program `s :- not r` can be safely removed from the program as at least one of `p` and `q` always holds (because of the first two rules) and so the body of either rule four or five is satisfied.

[10] `https://github.com/CVC4/CVC4/releases/tag/1.8`, with options `--lang tptp --stats --tlimit =300000`

[11] `http://www.philipp.ruemmer.org/princess-sources.shtml`, with options`-inputFormat=tptp -portfolio=casc -timeout=300000`

[12] `https://github.com/vprover/vampire/releases/tag/4.5.1`, as VAMPIRE provides parallelisation it is used in two configurations: `--mode casc --time_limit 300` (denoted by VAMPIRE), and using the parallelisation with the additional option `--cores 4` (denoted by VAMPIREP)

[13] `https://github.com/sneeuwballen/zipperposition/releases/tag/1.5`, with options `-timeout 300`

Program 3.A
```
p :- not q.
q :- not p.
r :- p, q.
s :- p.
s :- q.
s :- not r.
```

Program 3.B
```
p :- not q.
q :- not p.
r :- p, q.
s :- p.
s :- q.
```

The strong equivalence of the programs is verified by all theorem provers.

The next example attempts to rewrite a logic program, stating that exactly one of `p` and `q` has to be true, without using choice rules.

Program 4.A
```
{ p }.
{ q }.
:- p, q.
:- not p, not q.
```

Program 4.B
```
p :- not q.
q :- not p.
```

However, the two programs are not strongly equivalent as adding both `p` and `q` as facts results in a valid answer set $\{p, q\}$ for Program 4.B while Program 4.A does not have an answer set with these facts. Consequently, none of the theorem provers manages to verify the strong equivalence.

By adding the constraint `:- p, q` to the second program, the two programs can be made strongly equivalent as shown in the following example.

Program 5.A
```
{ p }.
{ q }.
:- p, q.
:- not p, not q.
```

Program 5.B
```
p :- not q.
q :- not p.
:- p, q.
```

The strong equivalence of the two programs can successfully be verified by all theorem provers.

Next is are the example programs from Sect. 1 defining a transitive relation p using a choice rule in combination with a basic rule and constraint respectively.

Program 6.A
```
{ p(X,Y) } :- q(X), q(Y).
p(X,Y) :- p(X,Z), p(Z,Y),
          q(X), q(Y), q(Z).
```

Program 6.B
```
{ p(X,Y) } :- q(X), q(Y).
:- p(X,Z), p(Z,Y), not p(X,Y),
   q(X), q(Y), q(Z).
```

ZIPPERPOSITION is the only prover that does not manage to verify this example in under 300 s.

Finally, Program 7.B attempts to simplify Program 7.A by removing the constraint `:- p(X), not q(X)` and instead adding the condition `q(X)` to the body of the choice rule.

Program 7.A

```
{ p(X) }.
:- p(X), not q(X).
```

Program 7.B

```
{ p(X) } :- q(X).
```

This is, however, not a strongly equivalent transformation, as simply adding the fact `p(1)` results in Program 7.A being unsatisfiable while Program 7.B has an answer set consisting of just `p(1)`. Thus, none of the theorem provers verifies the strong equivalence of the two programs.

6.3 Comparison of Automated Theorem Provers

The theorem provers were compared on a set of 21 example strong equivalence problems. All provers were used with a timeout of 300 s. For full details on the used examples and detailed results see [He20].

VAMPIRE managed to verify the most examples, just running into a timeout for one example. CVC4 only failed to do so on four of the examples with positive programs. PRINCESS failed on seven examples, and ZIPPERPOSITION failed on nine examples.

	CVC4	PRINCESS	VAMPIRE	VAMPIREp	ZIPPERPOSITION
#Solved problems	17	14	20	20	12
Average time	0.020 s	3.811 s	7.368 s	1.347 s	0.180 s

Tab. 1: Results of the theorem prover comparison

In most examples, CVC4 has the shortest run time. Using the parallelisation of VAMPIRE brings significant improvements in most examples, often bringing the time onto a similar level as CVC4 or even making VAMPIRE faster. Both PRINCESS and ZIPPERPOSITION are most of the time quite a bit slower than CVC4 and VAMPIRE (with PRINCESS being the slower one), but they still manage to verify most examples in one second or less.

In conclusion, CVC4 and VAMPIRE seem to be the best fit for the kind of problems generated by ANTHEM. While both PRINCESS and ZIPPERPOSITION do not perform much worse, they are not a better choice (than CVC4 or VAMPIRE) in any of the examples and run into timeouts more often compared to CVC4 and VAMPIRE. Therefore, using CVC4 and VAMPIRE in conjunction with ANTHEM seems to be the best option.

7 Contributions and Future Work

First, the new transformation σ^* was introduced. A similar transformation was first introduced by [PTW01]. The transformation considered here is a simplified version of the one in [PTW01]. In [PTW01] the same simplification is only applied to expressions (i.e. formulas only consisting of atoms and the connectives $\wedge$, $\vee$, and $\neg$). In this work, σ^* is used to express the equivalence of two formulas in the logic of here-and-there in classical logic, which is formalised in Theorem 2.

Second, the translation τ^* was extended to cover an input language containing pools. Theorem 3 formalises how the extended τ^* can be combined with σ^* to express the strong equivalence of two logic programs with an extended input language in classical logic.

Third, a new version of ANTHEM implements the extended translation τ^* as well as the transformation σ^* enabling the verification of strong equivalence of non-positive programs by using an automated theorem prover for classical logic. A similar system was developed by [CLL05], which utilises a transformation similar to σ^*. However, the system only supported ground programs in a much more limited input language compared to ANTHEM. While disjunctions are supported, choices, comparisons, arithmetic expressions, intervals, and pools are not supported. The requirement of ground programs means that programs have to be instantiated with a particular input, thus losing the ability to verify the equivalence under all possible inputs. Another difference is how the verification is done. The system of [CLL05] uses a SAT solver and can generate counter-examples for programs that are not strongly equivalent. Other similar system were introduced in [ETW05] for the propositional case and in [Oe06] for the non-propositional case. These two systems offer the advantage of considering more notions of equivalence besides strong equivalence. Both tools, however, have a more restricted input language than ANTHEM.

Fourth, options for automated theorem provers which are compatible with ANTHEM are investigated. Four automated theorem provers (CVC4, PRINCESS, VAMPIRE, and ZIPPERPOSITION) are tested on several examples. CVC4 and VAMPIRE emerged as the best options to use with ANTHEM. PRINCESS and ZIPPERPOSITION both only manage to verify fewer examples compared to CVC4 and VAMPIRE.

Future work on ANTHEM is mainly concerned with extending the input language. Desirable language features to support include an extended version of choices and aggregates, as both are essential features in most applications of answer set programming.

Future work will also have to include a further investigation into the theorem provers used in conjunction with ANTHEM. As the strong equivalence problems naturally lie in a non-classical logic, exploring theorem provers for either the logic of here-and-there or intuitionistic logic could be beneficial [Ot08, Ot21]. However, it seems most likely that it will be necessary to assist the theorem provers in the verification process. A method to do so has already been implemented for the version of ANTHEM used to verify the correctness of logic programs [Fa20]. A different approach could be an integration with interactive theorem proving systems that support a combination of interactive and automated theorem proving.

Acknowledgments

I would like to thank Patrick Lühne and Torsten Schaub for the supervision of my bachelor thesis. Furthermore, I thank Vladimir Lifschitz for corrections on some formal aspects and Christoph Wernhard as well as the anonymous referees for their comments.

Bibliography

[CLL05] Chen, Yin; Lin, Fangzhen; Li, Lei: SELP – A System for Studying Strong Equivalence Between Logic Programs. In: Logic Programming and Nonmonotonic Reasoning, volume 3662, pp. 442–446. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[EGL16] Erdem, Esra; Gelfond, Michael; Leone, Nicola: Applications of Answer Set Programming. AI Magazine, 37(3):53, October 2016.

[ETW05] Eiter, Thomas; Tompits, Hans; Woltran, Stefan: On Solution Correspondences in Answer-Set Programming. p. 97–102, 2005.

[Fa18] Falkner, Andreas; Friedrich, Gerhard; Schekotihin, Konstantin; Taupe, Richard; Teppan, Erich C.: Industrial Applications of Answer Set Programming. KI - Künstliche Intelligenz, 32(2-3):165–176, August 2018.

[Fa20] Fandinno, Jorge; Lifschitz, Vladimir; Lühne, Patrick; Schaub, Torsten: Verifying Tight Logic Programs with Anthem and Vampire. Theory and Practice of Logic Programming, 20(5):735–750, September 2020.

[Ge15] Gebser, Martin; Harrison, Amelia; Kaminski, Roland; Lifschitz, Vladimir; Schaub, Torsten: Abstract Gringo. Theory and Practice of Logic Programming, 15(4-5):449–463, July 2015.

[Ge19] Gebser, Martin; Kaminski, Roland; Kaufmann, Benjamin; Lindauer, Marius; Ostrowski, Max; Romero, Javier; Schaub, Torsten; Thiele, Sven; Wanko, Philipp: Potassco User Guide. Second edition edition, 2019.

[He20] Heuer, Jan: Automated Verification of Equivalence Properties in Advanced Logic Programs. Bachelor Thesis, University of Potsdam, October 2020.

[LLS19] Lifschitz, Vladimir; Lühne, Patrick; Schaub, Torsten: Verifying Strong Equivalence of Programs in the Input Language of Gringo. In: Logic Programming and Nonmonotonic Reasoning, volume 11481, pp. 270–283. Springer International Publishing, Cham, 2019.

[LPV01] Lifschitz, Valdimir; Pearce, David; Valverde, Agustín: Strongly Equivalent Logic Programs. ACM Transactions on Computational Logic, 2(4):526–541, October 2001.

[Oe06] Oetsch, Johannes; Seidl, Martina; Tompits, Hans; Woltran, Stefan: Cc⊤: A Correspondence-Checking Tool for Logic Programs Under the Answer-Set Semantics. In: Logics in Artificial Intelligence, volume 4160, pp. 502–505. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Ot08] Otten, Jens: leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic (System Descriptions). In: Automated Reasoning. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 283–291, 2008.

[Ot21] Otten, Jens: The nanoCoP 2.0 Connection Provers for Classical, Intuitionistic and Modal Logics. In: Automated Reasoning with Analytic Tableaux and Related Methods. Springer International Publishing, Cham, pp. 236–249, 2021.

[Pe97] Pearce, David: A New Logical Characterisation of Stable Models and Answer Sets. In: Non-Monotonic Extensions of Logic Programming, volume 1216, pp. 57–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[PTW01] Pearce, David; Tompits, Hans; Woltran, Stefan: Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In: Progress in Artificial Intelligence, volume 2258, pp. 306–320. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[Su09] Sutcliffe, Geoff: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning, 43(4):337–362, December 2009.

[Su12] Sutcliffe, Geoff; Schulz, Stephan; Claessen, Koen; Baumgartner, Peter: The TPTP Typed First-Order Form with Arithmetic. In: Logic for Programming, Artificial Intelligence, and Reasoning, volume 7180, pp. 406–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.