Finding Counterexamples for External Equivalence: Preliminary Report

Jorge Fandinno[1], Jan Heuer[2], Tomi Janhunen[3], Javier Romero[2] and Torsten Schaub[2,4]

[1]University of Nebraska Omaha, USA
[2]University of Potsdam, Germany
[3]Tampere University, Finland
[4]Potassco Solutions GmbH, Germany

Abstract

Refactoring is a common practice to improve code quality and maintainability while preserving external behavior. In Answer Set Programming, the *anthem* system supports this process by verifying the external equivalence of two encodings through a translation into first-order formulas, which are subsequently processed by automated theorem provers. However, while theorem proving is often successful for established equivalences, it frequently fails to terminate when programs are non-equivalent, leaving developers without actionable feedback during iterative refactoring. In this paper, we address this shortcoming by developing an approach for the automated generation of counterexamples for external equivalence. We extend an existing transformation for propositional programs to a first-order setting that covers a substantial subset of the *clingo* language, including non-tight encodings and aggregates. Furthermore, we generalize the transformation to handle private predicates through a novel guess-and-check formulation. Our approach is implemented in the *anthem-cx* system, providing developers with concrete witnesses of non-equivalence to facilitate the rapid identification of logical flaws in ASP encodings.

CEUR-WS.org/Vol-4311/ASPOCP_regular_05.pdf

1. Introduction

Refactoring is the process of restructuring existing code to improve its internal properties without altering its external behavior. In the context of Answer Set Programming (ASP [1]), the *anthem* system [2] supports this practice by verifying *external equivalence* [3], which ensures that two encodings exhibit identical behavior relative to a specified interface. To this end, *anthem* translates non-ground (locally-tight) ASP programs into first-order formulas, leveraging off-the-shelf theorem provers to establish their external equivalence.

In practice, this approach faces a limitation stemming from the semi-decidability of first-order theorem proving. While valid equivalence properties can often be proven in reasonable time, automated theorem provers may diverge when two programs are not equivalent, effectively running "forever" and leaving the equivalence question open. This lack of termination is particularly detrimental during iterative refactoring, which relies on rapid, fail-fast feedback. Furthermore, an inconclusive proof attempt provides no actionable insight into the underlying error. By shifting the focus to the generation of counterexamples, we address both of these practical issues. Although finding counterexamples remains generally undecidable, it naturally complements the theorem proving approach. In practice, counterexample search is highly effective precisely when equivalence does not hold. Not only does this provide a means to swiftly identify non-equivalence, but it also yields concrete witnesses, through specific input instances and their diverging output models, that allow developers to pinpoint logical flaws.

In this paper, we develop and implement an approach for finding counterexamples to external equivalence problems, specifically targeting the verification tasks addressed by the *anthem* system. While designed to complement *anthem*'s theorem-proving capabilities, our approach significantly

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026) co-located with the Federated Logic Conference (FLoC 2026).

jfandinno@unomaha.edu (J. Fandinno); jan.heuer@uni-potsdam.de (J. Heuer); tomi.janhunen@tuni.fi (T. Janhunen); javier.romero.davila@uni-potsdam.de (J. Romero); torsten.schaub@uni-potsdam.de (T. Schaub)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

expands its practical scope by accommodating a large subset of *clingo*'s input language [4], including non-tight encodings and aggregates. Our approach builds upon a transformation introduced in [5] for verifying weak equivalence of propositional programs. We extend this technique to verify the external equivalence of non-ground *clingo* encodings. In doing so, we generalize the account of private predicates, which provide local auxiliary definitions not subject to the equivalence check. Our counterexample generator is implemented in the *anthem-cx* tool and freely available.[1]

The rest of this paper is structured as follows. We introduce necessary background in Section 2. We define our transformation for generating counterexamples in Section 3 and discuss its correctness in a simplified setting that does not include private predicates. We extend the correctness discussion to account for private predicates in Section 4. While the transformation remains correct for restricted use of private predicates, we also discuss a novel adaptation accounting for arbitrary use of private predicates, as well as practical conditions to decide when this more general approach is necessary. In Section 5, we outline the design of the *anthem-cx* system, discuss its use as a companion tool to *anthem* and evaluate the combination of tools on a benchmark set. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. Background

We consider a reduced fragment of the language of Abstract Gringo [6] and ASP-Core-2 [7]: non-disjunctive programs with non-recursive aggregates without conditional literals, arithmetic terms, comparisons, function symbols, or constructs such as pools and intervals.

Syntax. An *atom* has the form $p(\boldsymbol{t})$ where p is a predicate and $\boldsymbol{t}$ is a tuple of terms, i.e., variables and constants. A *symbolic literal* is an atom possibly preceded by one or two occurrences of not.

An *aggregate atom* has the form $f\{\boldsymbol{t}_1 : L_1; \ldots; \boldsymbol{t}_n : L_n\} \prec u$, where

- f is one of $\#count, \#sum, \#min, \#max$,
- each $\boldsymbol{t}_i$ is a tuple of terms,
- each L_i is a (possibly empty) conjunction of symbolic literals,
- $\prec$ is one of the relations '$=$', '$\neq$', '$<$', '$>$', '$\leq$', '$\geq$',
- u is a term.

An *aggregate literal* is an aggregate atom possibly preceded by one or two occurrences of not. A *literal* is a symbolic literal or an aggregate literal. It is *negative* if it starts with one or two occurrences of not, and *positive* otherwise.

A *rule* has the form $h \leftarrow B$, where the head h is either $\bot$ or a symbolic atom, and the body B is a (possibly empty) conjunction of literals. We consider a choice rule $\{h\} \leftarrow B$, where h is an atom and B as above, as an abbreviation for the rule $h \leftarrow B \wedge not\ not\ h$. A *program* is a finite set of rules.

As in [7], we restrict the aggregates to be non-recursive. Further language constructs, such as head aggregates and choices with multiple elements and conditions, can be regarded as syntactic abbreviations, since they can be reduced to this fragment.

Semantics. The semantics for non-ground programs in our language is given in [6] by a transformation into infinitary propositional theories [8].

External Equivalence. A primary mode of *anthem* checks *external equivalence* [3], which is closely related to relativized uniform equivalence with projection [9, 10].[2] In this setting, programs are

[1] Available at: `https://github.com/potassco/anthem-cx`.

[2] External equivalence allows for more fine grained control of considered inputs. As in relativized uniform equivalence with projection we specify the input predicates but we can additionally restrict the set of possible inputs. E.g., we can specify that the first argument of the predicate $p/2$ is an integer between 1 and 50, or that an input graph (via $node/1$ and $edge/2$) can not contain self loops. Another difference is that external equivalence forbids the use of input predicates in rule heads.

associated with a user guide $(\mathcal{I}, \mathcal{O}, \mathcal{D})$ consisting of disjoint finite sets $\mathcal{I}$ and $\mathcal{O}$ of predicate symbols, called *input* and *output predicates*, respectively, and a collection $\mathcal{D}$ of *finite*[3] sets of ground atoms over $\mathcal{I}$, representing the admissible sets of *input atoms* for the programs.[4] Inputs not belonging to $\mathcal{D}$ fall outside the scope of the user guide. An *output atom* is a ground atom whose predicate belongs to $\mathcal{O}$.

A program adheres to a user guide if no input predicate appears in its rule heads. The predicate symbols in a program that are neither input nor output predicates are considered *private*. When comparing programs relative to a user guide, we assume their sets of private predicate symbols are disjoint. Given a program P, a user guide $(\mathcal{I}, \mathcal{O}, \mathcal{D})$, and an input $I \in \mathcal{D}$, the *external behavior* of P under input I is the set of all stable models of $P \cup I$ restricted to output atoms. Two programs P and Q are *externally equivalent* with respect to $(\mathcal{I}, \mathcal{O}, \mathcal{D})$ if they exhibit identical external behavior for every input in $\mathcal{D}$.

A *counterexample* to the external equivalence of P and Q with respect to $(\mathcal{I}, \mathcal{O}, \mathcal{D})$ is a pair of sets (I, O) where $I \in \mathcal{D}$ and O is an external behavior of either P or Q but not both. We say that a counterexample is for the *forward direction* if O is an external behavior of P, and for the *backward direction* otherwise. We define the *size* of a counterexample as the number of constants occurring in I.

We illustrate the definition of external equivalence with the orphan problem introduced in [3]. The set of input predicates is $\mathcal{I} = \{\mathit{father}, \mathit{mother}, \mathit{living}\}$, where *father* and *mother* are binary predicates and *living* is a unary predicate. The set of output predicates is $\mathcal{O} = \{\mathit{orphan}\}$, where *orphan* is a unary predicate. The domain $\mathcal{D}$ of the user guide is the collection of all sets of input atoms.

The first program, P_O, uses a private predicate *parentLiving* to define *orphan*

$$\begin{aligned} \mathit{parentLiving}(X) &\leftarrow \mathit{father}(Y, X) \wedge \mathit{living}(Y), \\ \mathit{parentLiving}(X) &\leftarrow \mathit{mother}(Y, X) \wedge \mathit{living}(Y), \\ \mathit{orphan}(X) &\leftarrow \mathit{living}(X) \wedge \mathit{not}\ \mathit{parentLiving}(X). \end{aligned}$$

While the second program, Q_O, defines *orphan* directly

$$\begin{aligned} \mathit{orphan}(X) \leftarrow\ & \mathit{living}(X) \wedge \mathit{father(Y,X)} \wedge \mathit{mother(Z,X)} \wedge \\ & \mathit{not}\ \mathit{living}(Y) \wedge \mathit{not}\ \mathit{living}(Z). \end{aligned}$$

The two programs are not externally equivalent, as we have the counterexample (I, O) with $I = \{\mathit{living}(0)\}$ and $O = \{\mathit{orphan}(0)\}$ of size 1 in the forward direction. Note that there is no counterexample in the backwards direction. If we instead take the domain $\mathcal{D}'$ as the collection of all inputs such that every living person has exactly one father and mother, then the programs are externally equivalent with respect to $(\mathcal{I}, \mathcal{O}, \mathcal{D}')$.

3. Basic Transformation

Our approach builds on an early method for testing the equivalence of logic programs [5]. We first outline the idea and then formulate it within a first-order setting. Here, we assume that the programs do not contain private predicates. We consider private predicates in Section 4.

Given two programs P and Q with a user guide $(\mathcal{I}, \mathcal{O}, \mathcal{D})$, we construct a program that is satisfiable if P exhibits an external behavior not shared by Q. Intuitively, this program performs the following four tasks:

1. Guess an input $I \in \mathcal{D}$.
2. Compute a stable model M of $P \cup I$.
3. Compute the minimal model M' of the reduct of $Q \cup I$ with respect to M.
4. Check whether the projections of M and M' onto $\mathcal{O}$ differ.

[3] The omission of this condition is an oversight in [3]. The restriction is due to the semantics of Abstract Gringo [6] which is defined for finite sets of rules only.

[4] This is a simplified version of user guides from [3], which additionally involve placeholders and their valuations.

If the preceding check reveals a discrepancy, we have found a counterexample that reflects a difference in the external behaviors of P and Q.

In our setting, the program $E_d(P, Q)$ is parametrized by the input size d. It consists of one module for each task: a generator $G(d)$ enumerating inputs, the program P, a modified copy $Q^\star$ of Q addressing Task 3, and a discrepancy checker C.

Before defining $G(d)$, note that we cannot represent $\mathcal{D}$ explicitly as the set of admissible inputs may be infinite. Instead, following [3], we specify $\mathcal{D}$ by a set $\mathcal{A}$ of assumptions. A finite set of input atoms belongs to $\mathcal{D}$ iff it satisfies all assumptions in $\mathcal{A}$. In this work, assumptions are constraints over input predicates.[5] For example, the domain $\mathcal{D}$ of the orphan example is represented by $\mathcal{A} = \emptyset$, while the domain $\mathcal{D}'$ is represented by

$$\begin{aligned}\mathcal{A}' = \{&\bot \leftarrow \mathit{living}(X) \wedge \mathit{not}\ \#\mathit{count}\{Y : \mathit{mother}(Y, X)\} \neq 1,\\ &\bot \leftarrow \mathit{living}(X) \wedge \mathit{not}\ \#\mathit{count}\{Y : \mathit{father}(Y, X)\} \neq 1\}.\end{aligned}$$

Given a natural number d, the component $G(d)$ contains the facts $dom(0), \ldots, dom(d-1)$ as well as the fact $dom(c)$ for every constant c occurring in P or Q. For every n-ary input predicate $p \in \mathcal{I}$ and $n \geq 0$, $G(d)$ contains a choice rule

$$\{p(X_1, \ldots, X_n)\} \leftarrow dom(X_1) \wedge \cdots \wedge dom(X_n).$$

In addition, $G(d)$ contains the assumptions $\mathcal{A}$.

To accomplish Tasks 2 and 3 within the same encoding, we introduce a new starred copy $q^\star$ for each output predicate $q \in \mathcal{O}$.[6] The component $Q^\star$ is then obtained by replacing in Q every positive literal $q(\boldsymbol{t})$ such that $q \in \mathcal{O}$ by $q^\star(\boldsymbol{t})$, and every occurrence of $\bot$ by the fresh atom bot. Observe that all rule heads in $Q^\star$ are free of $\bot$ and predicates from $\mathcal{I} \cup \mathcal{O}$.

Lastly, the component C uses a fresh atom diff to indicate a discrepancy between models. Specifically, C contains the rules

$$\begin{aligned}\bot &\leftarrow \mathit{not}\ \mathit{diff},\\ \mathit{diff} &\leftarrow \mathit{bot},\\ \mathit{diff} &\leftarrow p(X_1, \ldots, X_n) \wedge \mathit{not}\ p^\star(X_1, \ldots, X_n), \text{ and}\\ \mathit{diff} &\leftarrow \mathit{not}\ p(X_1, \ldots, X_n) \wedge p^\star(X_1, \ldots, X_n)\end{aligned}$$

for every n-ary output predicate $p \in \mathcal{O}$ with $n \geq 0$.

Altogether, for a natural number d, we obtain

$$E_d(P, Q) = G(d) \cup P \cup Q^\star \cup C.$$

The part $G(d) \cup P$ generates stable models M of $P \cup I$, where I is an input from $\mathcal{D}$. In $Q^\star$, negative literals are left unchanged. As a result, they are evaluated with respect to M, simulating the reduct of $Q \cup I$ with respect to M. The atoms derived from $Q^\star$, together with I, form a minimal model M' of this reduct. Finally, C checks whether the output atoms of M and M' differ (via diff), or some constraint from $Q^\star$ is violated (via additionally bot).

We return to the orphan example with the original domain $\mathcal{D}$. Even though the program P_O contains a private predicate, we can still use our method as for $E_d(P_O, Q_O)$ only Q_O needs to be free of private predicates. The $\cdot^\star$ transformation only changes the rule head of Q_O:

$$\begin{aligned}\mathit{orphan}(X)^\star \leftarrow\ &\mathit{living}(X) \wedge \mathit{father(Y,X)} \wedge \mathit{mother(Z,X)} \wedge\\ &\mathit{not}\ \mathit{living}(Y) \wedge \mathit{not}\ \mathit{living}(Z).\end{aligned}$$

The program $E_1(P_O, Q_O)$ has the stable model $\{\mathit{living}(0), \mathit{orphan}(0), \mathit{diff}\}$ representing the previously mentioned counterexample.

[5] In [3], assumptions are first-order formulas over input predicates.

[6] We do not need copies of input predicates as they cannot occur in rule heads.

3.1. Correctness

We discuss the correctness of $E_d(P, Q)$ by considering its parts separately.

The generator program $G(d)$ contains the constants occurring in P or Q as well as the set of constants $\{0, \ldots, d-1\}$ as domain elements (via dom). Since $G(d)$ contains the assumptions $\mathcal{A}$, every input generated by $G(d)$ belongs to $\mathcal{D}$.

However, $G(d)$ does not generate all possible inputs. The generated inputs have size at most $d + n$ where n is the number of constants occurring in P and Q. But not all inputs of this size are generated because we consider only a fixed set of domain elements. In particular, we do not obtain inputs containing negative integers or symbolic constants not occurring in P or Q, even after increasing the parameter d.

In the following, let $\mathcal{U} = (\mathcal{I}, \mathcal{O}, \mathcal{D})$ be the user guide of the external equivalence problem. For a set $\mathcal{P}$ of predicates, let $\mathcal{P}^\star$ be the set $\{p^\star \mid p \in \mathcal{P}\}$. Similarly, for a set A of atoms, let $A^\star$ be the set $\{p^\star(\boldsymbol{t}) \mid p(\boldsymbol{t}) \in A\}$. Let $\mathcal{U}^\star$ be the user guide with input predicates $\mathcal{I} \cup \mathcal{O}$, output predicates $\mathcal{O}^\star \cup \{bot\}$, and domain consisting of all sets of atoms over $\mathcal{I} \cup \mathcal{O}$.

We state two properties related to $Q^\star$: correctness and uniqueness. Correctness states that $Q^\star$ captures the external behavior of Q.

Lemma 1. *Let I and O be an input and an output of $\mathcal{U}$, respectively. The set O is an external behavior of Q with respect to $\mathcal{U}$ under input I iff the set $O^\star$ is an external behavior of $Q^\star$ with respect to $\mathcal{U}^\star$ under input $I \cup O$.*

This lemma justifies checking whether O is an external behavior of Q under input I by checking whether $O^\star$ is an external behavior of $Q^\star$ under input $I \cup O$.

We say that Q satisfies *uniqueness* with respect to $\mathcal{U}$ if for every input $I \in \mathcal{D}$ and every set of atoms O over $\mathcal{O}$, the program $Q^\star \cup I \cup O$ has a unique stable model.

Lemma 2. *If Q does not contain private predicates then it satisfies uniqueness with respect to $\mathcal{U}$.*

The reason is that $Q^\star$ is a stratified program: aggregates are not recursive, and negative symbolic literals use only predicates from $\mathcal{I} \cup \mathcal{O}$, which do not occur in the rule heads of $Q^\star$.

The program C receives as input the outputs of P and $Q^\star$. The former is some set of atoms O over $\mathcal{O}$, while the latter is a set of atoms U over $\mathcal{O}^\star \cup \{bot\}$. C is satisfied iff the atom $diff$ is derived. This is the case if (i) $bot \in U$ or (ii) there is some $p(\boldsymbol{t})$ with $p \in \mathcal{O}$ such that either $p(\boldsymbol{t}) \in O$ and $p^\star(\boldsymbol{t}) \notin U$ or $p^\star(\boldsymbol{t}) \in U$ and $p(\boldsymbol{t}) \notin O$. It follows that C is satisfied iff $O^\star \neq U$.

Finally, we put the results of the individual program parts together. $G(d)$ generates some input I of size $d + n$ of the user guide $\mathcal{U}$ where n is the number of constants occurring in P and Q. P has some external behavior O under input I. By uniqueness of Q (Lemma 2), $Q^\star$ has a unique external behavior U under input $I \cup O$. By correctness of $Q^\star$ (Lemma 1), $O^\star = U$ iff O is an external behavior of Q. Then we have that C is satisfied iff $O^\star \neq U$ which is the case iff O is not an external behavior of Q.

Theorem 1. *If Q has no private predicates, and $E_d(P, Q)$ is satisfiable for a natural number d, then P and Q are not externally equivalent with respect to $\mathcal{U}$.*

From the stable model M of $E_d(P, Q)$ we obtain the sets I and O defining the counterexample by projecting M to $\mathcal{I}$ and $\mathcal{O}$, respectively.

4. Private Predicates

4.1. Correctness of $E_d(P, Q)$

The presence of private predicates in Q does not affect our discussion from Section 3.1 about the correctness of $G(d)$ or C. We only have to consider whether we can lift the two properties related to $Q^\star$.

It turns out that correctness (Lemma 1) lifts directly to this extended setting, but uniqueness (Lemma 2) does not. In the presence of private predicates, the program $Q^\star$ simulates a partial reduct where the literals over private predicates are left unchanged. This may generate many stable models and break the approach. For example, consider the problem where $P = \{\{x\}, p \leftarrow x\}$, $Q = \{\{y\}, p \leftarrow y\}$, and the user guide has $\mathcal{I} = \emptyset$, $\mathcal{O} = \{p\}$, and $\mathcal{D} = \{\emptyset\}$. Clearly, P and Q are externally equivalent. But the program $Q^\star = \{\{y\}, p^\star \leftarrow y\}$ has two stable models, and for each model of P it can generate the wrong one. Thus, although the program $E_d(P, Q)$ should be unsatisfiable, it has two stable models: $\{x, p, \mathit{diff}\}$ and $\{y, p^\star, \mathit{diff}\}$.

Theorem 1 remains correct whenever P and Q satisfy the uniqueness property.[7]

Theorem 2. *If Q satisfies the uniqueness property with respect to $\mathcal{U}$, and $E_d(P, Q)$ is satisfiable for some natural number d, then P and Q are not externally equivalent with respect to $\mathcal{U}$.*

For example, in the orphan problem the program P_O satisfies uniqueness. Thus, if $E_d(Q_O, P_O)$ has a stable model, we obtain a correct counterexample. But as the orphan problem does not have a counterexample in the backwards direction, $E_d(Q_O, P_O)$ is unsatisfiable for every value d.

4.2. Guess-and-Check Transformation

For programs that do not satisfy the uniqueness criterion, we adapt our transformation using guess-and-check answer set programming.

A *guess-and-check problem* is a pair (G, C) of logic programs G and C. A solution to such a problem is a stable model X of G such that $X \cup C$ is unsatisfiable. The approach was introduced in [12]. We use an implementation for non-disjunctive logic programs[8] based on techniques described in [13].

Our guess program is $E_d^{\mathcal{G}}(P, Q) = G(d) \cup P$. It generates an input I of size $d + n$ (where n is the number of constants occurring in P and Q) and an external behavior O of P. Its correctness follows from the correctness of $G(d)$ discussed in Section 3.1.

Our check program is $E^{\mathcal{C}}(P, Q) = Q^\star \cup C'$, where C' is obtained from C by replacing the constraint $\bot \leftarrow \mathit{not}\ \mathit{diff}$ by $\bot \leftarrow \mathit{diff}$. It searches for an external behavior of Q (under input I) that coincides with O (the external behavior of P). If such an external behavior exists, then, by Lemma 1, $O^\star$ is an external behavior of $Q^\star$ under input $I \cup O$. In this case, diff is not derived in C', the constraint $\bot \leftarrow \mathit{diff}$ is satisfied, and the check program is satisfiable. Hence, the stable model of the guess program is not a solution for the guess-and-check problem. On the other hand, if no such external behavior of Q exists, the check program is unsatisfiable, and the stable model of the guess program is a solution.

Let $E_d^{\mathcal{GC}}(P, Q)$ denote the guess-and-check problem $(E_d^{\mathcal{G}}(P, Q), E^{\mathcal{C}}(P, Q))$.

Theorem 3. *If $E_d^{\mathcal{GC}}(P, Q)$ has a solution for some natural number d, then P and Q are not externally equivalent with respect to $\mathcal{U}$.*

For example, recall the externally equivalent programs from Section 4.1. While the program $E_d(P, Q)$ finds an incorrect counterexample, the guess-and-check program $E_d^{\mathcal{GC}}(P, Q)$ has no solution.

4.3. Verifying Uniqueness

Since guess-and-check problems are solved using disjunctive logic programs, it is useful to identify programs that satisfy the uniqueness condition, to avoid the additional complexity of disjunctive ASP. However, deciding whether a program satisfies uniqueness is already coNP-hard in the propositional case [11, Proposition 4.14]. We therefore consider two sufficient criteria below.

Stratification. Checking whether the program $Q^\star$ is stratified is a sufficient syntactic criterion for uniqueness. This can be checked directly on the signed dependency graph [14] of Q, after removing the edges induced by negative literals over output predicates. For the orphan example, this criterion establishes uniqueness.

[7] In [11] uniqueness is called *enough visible atoms (EVA)* in the context of private predicates.

[8] Available at: `https://github.com/potassco/guess_and_check`.

Local Uniqueness. If $E_d(P, Q)$ finds a potential counterexample and we do not know whether Q satisfies uniqueness, we may consider a local notion of uniqueness to check the validity of the counterexample. We say that Q is *locally unique* for a potential counterexample (I, O) if $Q^\star$ has a unique stable model under input $I \cup O$. If this condition holds, then (I, O) is a valid counterexample, even if Q does not satisfy uniqueness in general.

This suggests the following method. Start searching for potential counterexamples using $E_d(P, Q)$. If one is found, then check whether Q is locally unique for it. If that is the case, return the counterexample; otherwise, switch to the guess-and-check approach. If $E_d(P, Q)$ has no stable models, then the method can stop, provided that $Q^\star$ is call-consistent [14], i.e., its signed dependency graph has no loops containing an odd number of negative edges.[9]

Theorem 4. *For every natural number d, if $E_d(P, Q)$ has no stable models and $Q^\star$ is call-consistent, then $E_d^{\mathcal{GC}}(P, Q)$ has no solution.*

This condition is often satisfied in practice. Note that Q may include constraints, but in $Q^\star$ they are translated into normal rules and do not affect call-consistency. But if $Q^\star$ is not call-consistent, we need to use the guess-and-check approach. For example, consider the problem with $P = \emptyset$, $Q = \{x \leftarrow not\ x\}$, and the user guide $\mathcal{I} = \emptyset$, $\mathcal{O} = \emptyset$, and $\mathcal{D} = \{\emptyset\}$. The program $Q^\star = Q$ is not call-consistent. It is also unsatisfiable under every input. Hence, the program $E_d(P, Q)$ will not find a potential counterexample for any value of d. But the guess-and-check program $E_0^{\mathcal{GC}}(P, Q)$ has a solution, the empty set, which corresponds to the counterexample $(\emptyset, \emptyset)$.

5. The *anthem-cx* system

We have implemented the transformations $E_d(P, Q)$ and $E_d^{\mathcal{GC}}(P, Q)$ in the tool *anthem-cx*.[10] The implementation uses *clingo*'s python API to obtain and transform the abstract syntax trees of P and Q.

For the user guide we follow the syntax of *anthem*. While *anthem* supports the declaration of placeholders in a user guide, our method currently does not support placeholders and thus ignores them.[11] For *anthem* assumptions are declared as first-order formulas in the user guide. Such declarations are also ignored by our tool. Instead, assumptions can be passed as a separate file containing ASP constraints (only using input predicates) to *anthem-cx*.

Before transforming the programs, *anthem-cx* renames any conflicting private predicates. Additionally, *anthem-cx* ensures that the introduced auxiliary predicates (such as $diff$) are new by renaming them in case of conflicts.

After obtaining the counterexample programs our tool automatically invokes *clingo* to solve them. For this, the size parameter d is increased until a counterexample is found. The initial value of d can be adjusted (by default 0) and optionally a maximum value can be provided.

Regarding the uniqueness condition, we have implemented the two checks described in Section 4.3. We provide options to adjust which check(s) are used or alternatively disable or force the use of the guess-and-check method.

Finally, we note that our implementation includes experimental support for an even larger input language. In particular, syntactic shortcuts such as head aggregates are unfolded and features such as conditionals and pools are supported.

5.1. Using *anthem-cx* in conjunction with *anthem*

Our tool complements the functionality of *anthem* for investigating whether an external equivalence holds or not. We illustrate this with the *orphan* example from Section 2. The user guide (with domain $\mathcal{D}$)

[9] Here we consider edges induced by literals starting with two occurrences of not as *positive*. This differs from the treatment of such literals when checking stratification (where we consider such edges as *negative*).

[10] Available at: `https://github.com/potassco/anthem-cx`.

[11] A possible approach to support placeholders is to introduce a new predicate encoding the value of a placeholder. The programs are then adjusted to use this predicate and assumptions are added ensuring that each placeholder has exactly one value.

is represented by the following:

```
input: father/2.    input: mother/2.    input: living/1.
output: orphan/1.
```

Running *anthem* to verify the external equivalence of the programs P_O and Q_O results in a timeout as the programs are not externally equivalent. By running our system using the command

```
anthem-cx orphan.1.lp orphan.2.lp orphan.ug
```

we obtain a counterexample of size 1 in the forward direction as indicated by the output of *anthem-cx*:

```
Found a counterexample of size 1 in the forward direction
  Input for the counterexample: living(0)
  External behavior of left: orphan(0)
```

Under the input `living(0)` the first program, P_O, has the external behavior `orphan(0)` which is not an external behavior of the second program, Q_O.

As mentioned in Section 2, the programs are externally equivalent if we restrict the domain to $\mathcal{D}'$. If we add the assumptions $\mathcal{A}'$ from Section 3 (characterizing $\mathcal{D}'$) to *anthem-cx* by appending the following option

```
--assumptions orphan-assumptions.lp
```

to the previous command, the counterexample search no longer terminates. After adding these assumptions to the user guide as first-order formulas[12] *anthem* successfully verifies the external equivalence.

As a second example, we consider path finding on a graph. The user guide of this problem is defined as follows:

```
input: start/1.    input: goal/1.    input: edge/2.
output: path/2.
```

The first program computes a path by starting from the `start` node and consecutively choosing a successor node until the `goal` node is reached.

```
visit(X) :- start(X).
{ path(X,Y) : edge(X,Y) } = 1 :- visit(X), not goal(X).
visit(Y) :- visit(X), path(X,Y).
:- not visit(X), goal(X).
```

The second program follows a similar approach but computes the path in a backward manner. I.e., starting from the `goal` node, a predecessor node is consecutively chosen until the `start` node is reached.

```
reach(X) :- goal(X).
{ path(Y,X) : edge(Y,X) } = 1 :- reach(X), not start(X).
reach(Y) :- reach(X), path(Y,X).
:- not reach(X), start(X).
```

Running *anthem-cx*, we obtain a counterexample of size 1 in the forward direction with input `edge(0,0)` and `start(0)`, under which the first program has the external behavior `path(0,0)`. The second program does not have this external behavior, as it does not generate paths due to the lack of a `goal` node and is thus unsatisfiable.

Under the additional assumption that any input contains exactly one `start` and one `goal` node, the two programs become externally equivalent. However, we cannot verify this using *anthem* as the programs use aggregates and non-tight rules, which are not supported by *anthem*.

5.2. Experiments

Following the outlined approach, we test the combination of *anthem-cx* and *anthem* on a benchmark set of external equivalence examples. As our example problems, we use the benchmark problems from [2]

[12] The translation into first-order formulas needs to be performed manually here. In cases where the constraints are in the input language of *anthem* it is possible to use the translation mode of *anthem* to automatically obtain the first-order formulas.

Table 1
Evaluating the combination of *anthem-cx* and *anthem*

Problem	*anthem-cx* Runtime	*anthem* Runtime
Cover	»	0.197
Tiling	Timeout	Timeout
Primes (1v2)	–	3.612
Primes (2v3)	Timeout	Timeout
Frame	0.309	«
Orphan	0.269	«
Path	0.285	–
Transitive	0.284	–
Tiling'	0.361	–
Independent	0.320	«
Coloring	0.317	«

complemented by a variation of one of those problems (tiling') as well as some new examples (path, transitive, independent, coloring).

For the problems from [2] we disregard any assumptions defined in the user guide as well as any supplementary proof outlines (a feature of *anthem* to assist the proof process). The reason for this is that assumptions are usually not obvious initially and only added after unsuccessful proof attempts and careful consideration of the problem. Similarly, proof outlines are constructed after failed proof attempts. In this way, our setup reflects a user's initial attempt to verify the equivalence of their programs.

As some of the problems from [2] use placeholders we modify them to replace the placeholders by new input predicates and additional assumptions.[13] One of the problems from [2] is incompatible with *anthem-cx* due to the use of an unsafe rule in one of the programs. For the newly added problems we note that some of them are not supported by *anthem* as the programs are not in the input language of *anthem* or are not (locally) tight.

Table 1 contains for each problem the runtime of *anthem-cx* and *anthem* in seconds. The experiments were run on Ubuntu 24.04.4 LTS, Intel(R) Core(TM) Ultra 7 265U, 32 GB RAM with a 300 second time limit and 8 cores for *anthem* (i.e., its underlying theorem prover *vampire*). Problems not compatible with a tool are indicated by "–" for the runtime. Symbols "»" and "«" indicate that the complementary tool found a proof or counterexample, respectively. In these cases the tool must necessarily timeout as no counterexample or proof, respectively, exists. The problems are partitioned into equivalent and non-equivalent ones.

For 9 out of the 11 problems one of the tools terminates within in a few seconds. The two problems where neither tool terminates are equivalent but require the addition of a proof outline for *anthem* to find a proof. For the seven non-equivalent problems, *anthem-cx* finds a counterexample in under a second. Using just *anthem* it would not be clear whether the non-termination (of *anthem*) is due to the difficulty of the proof task or due to non-equivalence.

We note that *vampire*, the theorem prover used by *anthem*, has a portfolio mode, `casc_sat`, for showing satisfiability of a problem corresponding to a counterexample of the problem posed by *anthem*. Using this mode still results in timeouts for all problems that timeout for *anthem*.

6. Related Work

As explained in Section 3, our work builds upon an earlier approach for verifying (weak) equivalence of propositional logic programs [5]. The method is implemented by a translator *lpeq* [15] for combining ground logic programs so that the actual verification task can be carried out by an ASP solver. This work is part of a series of investigations on different notions of equivalences [5, 11, 16, 17] and their

[13] We note that this modification may impact the runtime (of *anthem*) negatively.

implementation using native ASP solvers as back-end solvers. There are further generalizations for disjunctive logic programs [18] and translators [15] also supporting the verification of classical and strong equivalence.

Concepts related to program interfaces and encapsulation are taken increasingly into account, e.g., in terms of private predicates [11] and by splitting public predicates into input and output predicates [16, 17] for program modules.[14] The input language of all these approaches is limited to propositional (ground) programs with more restricted forms of aggregates. Furthermore, the transformations (and the respective implementations) of all aforementioned works accounting for private predicates are limited to programs fulfilling uniqueness. Any modifications to lift this restriction, such as our method from Section 4.2, are not considered.

For the common language fragment, our transformation behaves almost exactly like the one from [17] but there is a key difference in the treatment of private predicates. The original translation $E(P, Q)$ of [17] introduces a second copy of every private predicate of Q, and a second copy of every rule of Q defining a private predicate such that this second copy only refers to the joint input and the external behaviour of P. Moreover, the translation contains additional rules to derive the diff predicate for differences in these two versions of the private predicates. In contrast, our translation does not duplicate the private predicates of Q (and their rules). Under uniqueness, any differences in the two copies of private predicates must result from differences in the output predicates of P and Q. Therefore, as all output predicates are checked for differences the additional copies of private predicates become unnecessary.

Another closely related approach [19, 20, 21, 10] reduces equivalences of disjunctive propositional programs without aggregates to QBFs. These works are based on so-called *correspondence frames*, providing a general framework for specifying equivalence relations, with weak or uniform equivalence (optionally with relativization and projection) as particular instances of this framework. The transformation to QBFs encodes arbitrary correspondence frames. Due to the expressiveness of QBFs, private predicates (even in disjunctive programs) are fully supported. While a follow-up study [22] discusses potential adaptations to the non-ground case no concrete implementation exists.

The overall approach of testing on small inputs corresponds to the so-called *small-scope hypothesis*: a general principle from software testing stating that most errors can be found by testing a program for all inputs of some small size. For ASP, an empirical evaluation of the small-scope hypothesis was conducted in [23]. The idea is to check the equivalence of a program and its mutated version, obtained by introducing small modifications to the program such as dropping literals, adding negation or renaming variables. The empirical evaluation confirmed that the small scope hypothesis holds in this setting. While such an approach of testing on small inputs may hold empirically it is incomplete even for simple classes of programs. This follows directly from the undecidability of query equivalence for Horn programs as shown in [24]. Query equivalence is a special case of external equivalence when there is only a single output predicate.

7. Discussion

We have developed a transformation to generate counterexamples to external equivalence problems by extending an earlier method [5] for verifying weak equivalence of propositional programs. As the input language we consider non-ground programs with non-recursive aggregates corresponding to a large fragment of Abstract Gringo [6] and ASP-Core-2 [7]. Our approach allows for the unrestricted use of private predicates (i.e., predicates not subject to equivalence), lifting a restriction from earlier ASP-based verification work.

We have implemented our approach in the *anthem-cx* tool. First experiments show that our method is effective for identifying non-equivalence by quickly computing concrete counterexamples. This complements existing work on verifying external equivalence using *anthem* [2].

[14] The notions of equivalence in [11, 16, 17] also take into account the numbers of stable models in one-to-one correspondence. However, due to the limitation to programs fulfilling the uniqueness property, this aspect need not be accounted for.

For simplicity, we have restricted the use of aggregates to be non-recursive. However, our results remain valid for unrestricted monotone aggregates (under Ferraris semantics). For general recursive aggregates (and disjunctive programs) this is no longer the case, although Lemma 1 and Theorem 2 continue to hold.

We plan to extend our transformation to account for the full Abstract Gringo language in future work. Furthermore, we plan to adapt the related method from [15] for verifying strong equivalence of propositional programs to the Abstract Gringo language as well.

Acknowledgments

This work was partly funded by DFG grant SCHA 550/15, Germany, and by the National Science Foundation CAREER award 2338635, USA. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Declaration on Generative AI

During the preparation of this work, the authors used Claude and Gemini in order to: Paraphrase and Reword. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] V. Lifschitz, Answer Set Programming, Springer-Verlag, 2019. doi:10.1007/978-3-030-24658-7.

[2] J. Fandinno, Z. Hansen, Y. Lierler, C. Glinzer, J. Heuer, T. Schaub, T. Stolzmann, V. Lifschitz, ANTHEM 2.0: Automated reasoning for answer set programming, Theory and Practice of Logic Programming 25 (2025) 668–684. doi:10.1017/S1471068425100112.

[3] J. Fandinno, Z. Hansen, Y. Lierler, V. Lifschitz, N. Temple, External behavior of a logic program and verification of refactoring, Theory and Practice of Logic Programming 23 (2023) 933–947.

[4] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[5] T. Janhunen, E. Oikarinen, Testing the equivalence of logic programs under stable model semantics, in: S. Flesca, S. Greco, N. Leone, G. Ianni (Eds.), Proceedings of the Eighth European Conference on Logics in Artificial Intelligence (JELIA'02), volume 2424 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002, pp. 493–504. doi:10.1007/3-540-45757-7_41.

[6] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, T. Schaub, Abstract Gringo, Theory and Practice of Logic Programming 15 (2015) 449–463. doi:10.1017/S1471068415000150.

[7] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory and Practice of Logic Programming 20 (2020) 294–309.

[8] M. Truszczyński, Connecting first-order ASP and the logic FO(ID) through reducts, in: E. Erdem, J. Lee, Y. Lierler, D. Pearce (Eds.), Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz, volume 7265 of *Lecture Notes in Computer Science*, Springer-Verlag, 2012, pp. 543–559.

[9] J. Oetsch, H. Tompits, Program correspondence under the answer-set semantics: The non-ground case, in: M. Garcia de la Banda, E. Pontelli (Eds.), Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08), volume 5366 of *Lecture Notes in Computer Science*, Springer-Verlag, 2008, pp. 591–605.

[10] J. Oetsch, M. Seidl, H. Tompits, S. Woltran, Beyond uniform equivalence between answer-set programs, ACM Transaction on Computational Logic 22 (2021) 2:1–2:46. doi:10.1145/3422361.

[11] T. Janhunen, E. Oikarinen, Automated verification of weak equivalence within the SMODELS system, Theory and Practice of Logic Programming 7 (2007) 697–744. doi:10.1017/S1471068407003031.

[12] T. Eiter, A. Polleres, Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications, Theory and Practice of Logic Programming 6 (2006) 23–60.

[13] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own ASP-based system?!, Theory and Practice of Logic Programming 23 (2023) 299–361. doi:10.1017/S1471068421000508.

[14] P. Dung, On the relations between stable and well-founded semantics of logic programs, Theoretical Computer Science 105 (1992) 7–25. doi:https://doi.org/10.1016/0304-3975(92)90285-N.

[15] T. Janhunen, E. Oikarinen, LPEQ and DLPEQ — translators for automated equivalence testing of logic programs, in: [25], 2004, pp. 336–340.

[16] E. Oikarinen, T. Janhunen, Modular equivalence for normal logic programs, in: G. Brewka, S. Coradeschi, A. Perini, P. Traverso (Eds.), Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06), IOS Press, 2006, pp. 412–416.

[17] E. Oikarinen, T. Janhunen, A translation-based approach to the verification of modular equivalence, Journal of Logic and Computation 19 (2009) 591–613.

[18] E. Oikarinen, T. Janhunen, Verifying the equivalence of logic programs in the disjunctive case, in: [25], 2004, pp. 180–193. doi:10.1007/978-3-540-24609-1_17.

[19] H. Tompits, S. Woltran, Towards implementations for advanced equivalence checking in answer-set programming, in: M. Gabbrielli, G. Gupta (Eds.), Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05), volume 3668 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005, pp. 189–203. doi:10.1007/11562931_16.

[20] J. Oetsch, M. Seidl, H. Tompits, S. Woltran, ccT: A correspondence-checking tool for logic programs under the answer-set semantics, in: M. Fisher, W. van der Hoek, B. Konev, A. Lisitsa (Eds.), Proceedings of the Tenth European Conference on Logics in Artificial Intelligence (JELIA'06), volume 4160 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006, pp. 502–505. doi:10.1007/11853886_47.

[21] J. Oetsch, H. Tompits, S. Woltran, Facts do not cease to exist because they are ignored: Relativised uniform equivalence with answer-set projection, in: Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07), AAAI Press, 2007, pp. 458–464. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-072.php.

[22] J. Oetsch, M. Seidl, H. Tompits, S. Woltran, ccT on stage: Generalised uniform equivalence testing for verifying student assignment solutions, in: E. Erdem, F. Lin, T. Schaub (Eds.), Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09), volume 5753 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 2009, pp. 382–395.

[23] J. Oetsch, M. Prischink, J. Pührer, M. Schwengerer, H. Tompits, On the small-scope hypothesis for testing answer-set programs, in: G. Brewka, T. Eiter, S. McIlraith (Eds.), Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12), AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4550.

[24] O. Shmueli, Decidability and expressiveness of logic queries, in: M. Vardi (Ed.), Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87), ACM Press, 1987, pp. 237–249. doi:10.1145/28659.28685.

[25] V. Lifschitz, I. Niemelä (Eds.), Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04), volume 2923 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 2004.