TPLP: Page 1–17.

doi:10.1017/S1471068425100112

ANTHEM 2.0: Automated Reasoning for Answer Set Programming

JORGE FANDINNO, ZACHARY HANSEN and YULIYA LIERLER
University of Nebraska Omaha, Omaha, NE, USA
(*e-mails:* jfandinno@unomaha.edu, zachhansen@unomaha.edu and ylierler@unomaha.edu)

CHRISTOPH GLINZER, JAN HEUER, TORSTEN SCHAUB and TOBIAS STOLZMANN
University of Potsdam, Potsdam, Germany
(*e-mails:* glinzer@uni-potsdam.de, jan.heuer@uni-potsdam.de, torsten.schaub@uni-potsdam.de and tobias.stolzmann@uni-potsdam.de)

VLADIMIR LIFSCHITZ
University of Texas at Austin, Austin, TX, USA
(*e-mail:* lifschitzv@gmail.com)

submitted 17 July 2025; revised 17 July 2025; accepted 27 July 2025

Abstract

ANTHEM 2.0 is a tool to aid in the verification of logic programs written in an expressive fragment of CLINGO 's input language named MINI-GRINGO, which includes arithmetic operations and simple choice rules but not aggregates. It can translate logic programs into formula representations in the logic of here-and-there and analyze properties of logic programs such as tightness. Most importantly, ANTHEM 2.0 can support program verification by invoking first-order theorem provers to confirm that a program adheres to a first-order specification or to establish strong and external equivalence of programs. This paper serves as an overview of the system's capabilities. We demonstrate how to use ANTHEM 2.0 effectively and interpret its results.

KEYWORDS: knowledge representation and nonmonotonic reasoning, logic programming methodology and applications

1 Introduction

We present ANTHEM 2.0, the latest system developed as part of the "Answer Set Programming + Theorem Proving" (ANTHEM) project. This system consolidates the findings and functionalities of previous prototypes into a stable tool for verifying programs in the paradigm of Answer Set Programming (ASP). The ANTHEM 2.0 system (in the sequel we drop 2.0 when referring to this system) can process ASP programs written in a subset of the input language of the grounder GRINGO (Gebser *et al.* 2015).

This subset, described by Fandinno *et al.* (2020), is referred to as MINI-GRINGO. MINI-GRINGO programs can contain arithmetic operations and simple choice rules, but not aggregates. We can use ANTHEM to verify two important equivalence relations between ASP programs: strong equivalence and external equivalence. ASP programs Π_1 and Π_2 are called *strongly equivalent* if replacing Π_1 by Π_2 within any larger program does not affect its stable models (Lifschitz *et al*. 2001). For example, the one-rule *successor* programs

`q(X+1) :- p(X).` and `q(X) :- p(X-1).` (1)

are strongly equivalent. The systems for verifying strong equivalence designed in the past (Janhunen and Oikarinen 2004; Valverde 2004; Chen *et al.* 2005; Oetsch *et al.* 2009) are limited to ground programs; ANTHEM does not suffer this limitation.

```
composite(I*J) :- I = 2..b, J = 2..b.
prime(I) :- I = a..b, not composite(I).
```

Listing 1. A MINI-GRINGO program, `primes.1.lp`.

External equivalence is equivalence with respect to a "user guide" – a description of how the programs are meant to be used (Fandinno *et al.* 2023). Consider, for instance, the program[1] in Listing 1, which finds all primes within an interval $\{a, \ldots, b\}$ with $a > 1$. Its intended use can be described by the user guide in Listing 2.

```
input: a -> integer.          output: prime/1.
input: b -> integer.          assumption: a > 1.
```

Listing 2. A user guide for the primes problem.

It tells us that when we run the program, we are expected to specify appropriate values for the placeholders `a` and `b`. Furthermore, it says that the output of the program consists of atoms in the stable model that contain `prime/1`; any other atoms of the stable model are auxiliary ("private").

ASP programs Π_1 and Π_2 are (externally) equivalent with respect to a user guide if, for any input that is permitted by the user guide, they produce the same output. For example, the program in Listing 1 is externally equivalent to the more efficient program

```
composite(I*J) :- I = 2..b, J = 2..b/I.
prime(I) :- I = a..b, not composite(I).
```

with respect to the user guide defined above, even though the stable models of the two programs differ with respect to the auxiliary predicate `composite/1`.

In the special case when programs do not accept inputs and have no auxiliary predicates, external equivalence means simply that the programs have the same stable models. We call this *weak equivalence*. If we allow auxiliary predicates and inputs, but we restrict the inputs to not contain placeholders, then external equivalence becomes a special case of *relativized uniform equivalence with projection* (Oetsch and Tompits 2008).

[1] Most of the examples in this paper can also be found in the `res/examples` directory of the ANTHEM repository; https://github.com/potassco/anthem. The repository also contains installation instructions and a more detailed user manual.

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

In addition to checking equivalence of programs, ANTHEM can verify the adherence of an ASP program to a specification written in classical first-order logic. For example, the formula below written in the custom language of ANTHEM captures the formal property encoded in the preceding example:

```
forall X$g (prime(X$g) <-> a <= X$g <= b
           and not exists D$i M$i (1 < D$i < X$g and M$i*D$i = X$g)).
```

This formula contains variables of two sorts: a general variable named `X` and integer variables `D` and `M`. The sort of the variable is indicated by the suffix `$g` or `$i` for the sorts *general* and *integer*, respectively. Analogously to how it checks external equivalence between programs, ANTHEM can verify that the `prime/1` predicate as defined by Listing 1 possesses the property encoded by the specification with respect to the user guide in Listing 2.

Using ANTHEM to verify external equivalence between MINI-GRINGO programs involves

- transforming rules into first-order sentences,
- forming program completions,
- reducing the tasks described above to first-order theorem proving, and
- using the theorem prover VAMPIRE (Kovács and Voronkov 2013) for proof search.

Fandinno *et al.* (2020) describe the original system that pioneered some of the presented ideas. We refer to this system as ANTHEM-1. It was designed with the goal of verifying the adherence of a program to a specification written in first-order logic. A related "translate and verify" system, ANTHEM-SE, was developed by Lifschitz *et al.* (2019) for verifying strong equivalence and extended to programs with negation by Heuer (2020). Finally, the ANTHEM-P2P system (Fandinno *et al.* 2023) was an application built on top of the ANTHEM-1 system for verifying external equivalence of programs.

Now, we present ANTHEM 2.0, which combines and extends the functionalities of the previous systems within a new, standalone library and command-line application. Our system helps users verify strong, external, and weak equivalence of programs, analyze properties of programs such as tightness and regularity, and translate programs into the syntax of many-sorted first-order logic. The theory behind ANTHEM has been given a thorough treatment in previous publications, and we refer to these publications for theoretical concepts instead of redefining them here.

2 Preliminaries: Program and "Target" languages

The ABSTRACT GRINGO (AG) language (Gebser *et al.* 2015) is a theoretical representation of the input language accepted by the widely used ASP solver CLINGO. The fragments of AG studied in the context of ANTHEM are commonly referred to as MINI-GRINGO – this is the subset of AG for which an appropriate translation to the syntax of first-order logic has been widely studied. In this paper, we follow the definition presented by Fandinno *et al.* (2020) when we refer to MINI-GRINGO. A MINI-GRINGO program consists of basic rules, choice rules, and constraints:

$$H \text{ :- } B_1, \ldots, B_n. \qquad \{H\} \text{ :- } B_1, \ldots, B_n. \qquad \text{:- } B_1, \ldots, B_n.$$

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

Here H is an atom and each B_i $(1 \leq i \leq n)$ is either an atom, possibly preceded by one or two negation-as-failure symbols, or a comparison. The fundamental task of ANTHEM consists of verifying the correctness of MINI-GRINGO programs (Section 4). To achieve this goal, the system implements multiple translations between representations of these programs in different languages. One of these languages is the TPTP format, which is a standard format for Automated Theorem Prover (ATP) systems (Sutcliffe 2017). Hence, the different verification tasks that ANTHEM can perform are based on the translation of MINI-GRINGO programs into a language of logical formulas, running an ATP system on a task assembled from the translated program(s), and interpreting the results of the ATP system to provide an answer to the verification task. Transforming an equivalence claim about MINI-GRINGO programs into a series of TPTP problems is a non-trivial process and in general requires several intermediate transformations. Thus, at the heart of ANTHEM is a logical language, which we call here simply the *target language*, that supports all these intermediate transformations. Theories written in this language may be interpreted under the semantics of classical first-order logic or the logic of *here-and-there* (HT; Heyting 1930), but *syntactically* this is a first-order language with variables of three sorts: (1) a sort whose universe contains integers, symbolic constants, and special symbols `#inf` and `#sup`, (2) a subsort corresponding to integers, and (3) a subsort corresponding to symbolic constants.

Variables ranging over these sorts are written as “`Name$sort`”, where `Name` is a capitalized word and `sort` is one of the sorts defined above. Certain abbreviations are permitted:

- a general variable named `V` can be written as `V`, `V$g`, or `V$general`;
- an integer variable named `X` can be written as `X$`, `X$i`, or `X$integer`; and
- a symbol variable named `S` can be written as `S$s` or `S$symbol`.

Like ABSTRACT GRINGO and MINI-GRINGO, the target language is a theoretical language – a formula written in this language can be printed as a series of Unicode characters in different ways. For example, the target language formula

$$\forall X(\exists I(I = X \wedge p(I)) \rightarrow q(X)) \tag{2}$$

(in which X is a general variable and I is an integer variable) expresses that q holds for all X such that X is an integer for which p holds. This formula can be read or displayed in a custom ANTHEM syntax as follows:

```
forall X ( exists I$ (I$ = X and p(I$)) -> q(X)).
```

ANTHEM can also display target language formulas in the TPTP syntax. The preceding formula (2) is formatted in TPTP as

```
![X: general]: ( ?[I: $int]: ( ( f__integer__(I) = X ) &
                               p(f__integer__(I)) ) => q(X) ).
```

In Section 4, we show how verification tasks are handled by extending ANTHEM's custom syntax for target language formulas with meta-level declarative statements controlling proof search – we call this the *control language*.

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

3 Translating logic programs and formulas with ANTHEM

ANTHEM supports (1) translations from ASP programs into logical formulas, and (2) transformations of logical formulas within the target language.

3.1 Translating ASP programs into the target language

Past versions of the ANTHEM system (Fandinno *et al.* 2020; Heuer 2020) relied exclusively on a translation known as τ^* (Lifschitz *et al.* 2019) that transformed MINI-GRINGO programs into their formula representations. To accommodate features such as partial arithmetic functions, τ^* produces complex formulas that are not easily understood. However, a broad fragment of MINI-GRINGO programs does not require such complexity. For a class of rules called *regular*, a more natural translation ν has been developed to produce formulas that are more human-readable while maintaining equivalence to their τ^* counterparts (Lifschitz 2021). ANTHEM employs both the τ^* translation[2] and natural translation ν. For example, the natural translations of successor programs (1) are

```
forall X$i (p(X$i) -> q(X$i + 1)), forall X$i (p(X$i - 1) -> q(X$i)).
```

For rules that do not meet the requirements of regularity, ANTHEM falls back on τ^* to obtain their formula representations. Translating nonregular rules is a non-trivial task; for example, consider the rule

```
p(X,Y) :- X / Y > 0.
```

which is not regular due to the partial function of division. We can apply translation τ^* to this rule using the command

```
anthem translate <program> --with tau-star
```

which produces the output

```
forall V1 V2 X Y (V1 = X and V2 = Y and exists Z Z1 (
  exists I$i J$i Q$i R$i (I$i = J$i * Q$i + R$i and (I$i = X and J$i = Y)
    and (J$i != 0 and R$i >= 0 and R$i < J$i) and Z = Q$i)
  and Z1 = 0 and Z > Z1) -> p(V1, V2)).
```

Interestingly, τ^* and ν can be safely combined by applying ν to every regular rule and τ^* to the rules that are not regular. Such a transformation is denoted μ (Fandinno and Lifschitz 2023b). ANTHEM supports μ, ν, and τ^* as translation options:

```
anthem translate <program> --with <mu | natural | tau-star>
```

3.2 Transformations within the target language

To support the verification of ASP programs, ANTHEM provides two transformations that can be applied to logical formulas within the target language. Each of these transformations provides as output a formula whose validity in classical first-order logic supports some of the verification tasks described in Section 4.

[2] In reality, τ^* is an overloaded term referring to several iterations on the idea of translating between fragments of AG and many-sorted formulas. The definition used here follows a slightly corrected version of the seminal ANTHEM-1 publication that can be found at https://arxiv.org/abs/2008.02025.

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

3.2.1 From here-and-there satisfaction to classical satisfaction

Stable models of MINI-GRINGO programs can be described in terms of the logic of HT by selecting the so-called *equilibrium models* (Pearce 2006). This connection has inspired a line of research into translating programs written in ASP input languages into HT theories whose equilibrium models correspond to the answer sets of the original program. Interestingly, for the translations concerning this paper, two programs are strongly equivalent if and only if their formula representations are equivalent in HT (Lifschitz *et al.* 2001). This gives us a method for verifying strong equivalence.

To reduce the task of reasoning about HT theories to reasoning over classical theories, transformations have been proposed that "embed" the behavior of HT-satisfaction into classical satisfaction of transformed formulas. One of the first representations of this process in the ASP literature can be attributed to Pearce *et al.* (2001), who describes a transformation that converts propositional HT theories into classical theories over an extended signature. A generalization γ of this translation to formulas with variables was studied by Heuer (2020) and by Fandinno and Lifschitz (2023b). The γ transformation was fundamental to the design of ANTHEM-SE, which implemented a procedure for strong equivalence checking. For example, if R is the rule

```
q(X,Y+1) :- p(X,Y).
```

then νR is

$$\forall XN(p(X,N) \rightarrow q(X,N+1)).$$

where N is an integer variable. The result of applying γ to this formula is

$$\forall XN((hp(X,N) \rightarrow hq(X,N+1)) \wedge (tp(X,N) \rightarrow tq(X,N+1))).$$

The classical models of $\gamma(\nu R)$ satisfying the additional axioms

$$\forall XY(hp(X,Y) \rightarrow tp(X,Y)) \qquad \text{and} \qquad \forall XY(hq(X,Y) \rightarrow tq(X,Y))$$

correspond to the HT models of νR. In terms of Kripke models, new predicates $hp/1$ and $hq/1$ represent satisfaction in the "here" world; $tp/1$ and $tq/1$ represent satisfaction in the "there" world. We call such additional axioms *ordering* sentences.

The γ transformation plays a central role in the automated verification of strong equivalence (Section 4.1).

3.2.2 Completion

For the so-called *tight* programs (Fandinno *et al.* 2020, Section 6), Clark's Completion (Clark 1978) characterizes the stable models of a logic program as the models of a classical first-order theory. Since its introduction, the idea of completion has been widely generalized. In the context of ANTHEM, we view completion as a transformation on top of target language theories of a certain form. These so-called *completable* theories can be transformed according to the completion procedure described by Fandinno *et al.* (2024) into first-order theories. The τ^* transformation is designed to produce completable theories when applied to MINI-GRINGO programs. For instance, we can compute the

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

completion of the program `choice.1.lp` consisting of the rule $\{q(X)\} \text{ :- } p(X)$ by passing the output of the τ^* translation to the completion operator:

```
anthem translate choice.1.lp --with tau-star \
        | anthem translate --with completion
```

This produces the output

```
forall V1 (q(V1) <-> exists X (V1 = X and
                        exists Z (Z = X and p(Z)) and not not q(V1))).
forall V1 (p(V1) <-> #false).
```

Completion allows us to use first-order theorem provers to verify the external equivalence of programs meeting certain restrictions, such as tightness (Section 4.2). We can confirm that our program is tight with the command

```
anthem analyze choice.1.lp --property tightness
```

4 Verifying logic programs with ANTHEM

The fundamental task of ANTHEM consists of formally verifying properties of MINI-GRINGO programs. This section describes the three verification tasks that ANTHEM can perform: *strong equivalence*, *external equivalence*, and *specification adherence*.

4.1 Strong equivalence

The strong equivalence of two programs Π_1 and Π_2 can be established by deriving the equivalence $\gamma(\tau^*\Pi_1) \leftrightarrow \gamma(\tau^*\Pi_2)$ (or, equivalently, $\gamma(\mu\Pi_1) \leftrightarrow \gamma(\mu\Pi_2)$) from the associated ordering sentences. ANTHEM accomplishes this by constructing a series of subproblems for an ATP system to solve. To illustrate this process, consider the encoding of the property

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

Listing 3. The program `transitive.1.lp`.

"q is transitive on the domain of p" given in Listing 3 (Harrison *et al*. 2017), and a refactoring of this program (`transitive.2.lp`) that replaces the second rule with the constraint

```
:- q(X,Y), q(Y,Z), not q(X,Z), p(X), p(Y), p(Z).      (3)
```

We can verify their strong equivalence by invoking ANTHEM with an instruction to use μ for obtaining the HT formula representation of these two programs:

```
anthem verify --equivalence strong transitive.{1.lp,2.lp} \
    --formula-representation mu
```

Let us denote Rule 1 from Listing 3 as F (this rule is common to both programs). Let G_1 denote Rule 2 from Listing 3, and G_2 denote the constraint (3). Furthermore, let $\mathcal{A}$

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

denote ordering sentences

$$\forall X(hp(X) \to tp(X)) \quad \text{and} \quad \forall X_1 X_2(hq(X_1, X_2) \to tq(X_1, X_2)).$$

The strong equivalence of these programs can be established by showing that

$$\mathcal{A} \to (\gamma(\mu(\{F, G_1\})) \leftrightarrow \gamma(\mu(\{F, G_2\})))$$

is satisfied, in the sense of classical first-order logic, by all standard[3] interpretations (Fandinno and Lifschitz 2023b,Theorem 2). ANTHEM decomposes this task into four subproblems, each of which consists of verifying that one of the following implications is satisfied by all standard interpretations:

$$(\mathcal{A} \wedge \gamma(\mu F) \wedge \gamma(\mu G_1)) \to \gamma(\mu F), \qquad (\mathcal{A} \wedge \gamma(\mu F) \wedge \gamma(\mu G_1)) \to \gamma(\mu G_2),$$
$$(\mathcal{A} \wedge \gamma(\mu F) \wedge \gamma(\mu G_2)) \to \gamma(\mu F), \qquad (\mathcal{A} \wedge \gamma(\mu F) \wedge \gamma(\mu G_2)) \to \gamma(\mu G_1).$$

ANTHEM checks that these implications (two of which are trivial) are entailed by a set of axioms describing standard interpretations using a classical ATP system, currently VAMPIRE, and answers that the two programs are strongly equivalent if it finds that all four implications are entailed by the axioms.

4.2 External equivalence

Strong equivalence can sometimes be *too* strong of a condition when we are comparing the behavior of two programs. Often, we are only interested in confirming that two programs have the same output when paired with the same input. This type of equivalence is called *external equivalence* (Fandinno *et al.* 2023). External equivalence is defined with respect to a *user guide*, which defines a class of acceptable inputs to the programs, and specifies which predicates encode their output.

A user guide contains statements of three types: input declarations, output declarations, and assumptions about the inputs. An input declaration describes an input predicate or a *placeholder* (a symbolic constant i.e., given a value by the program's input). Note that ANTHEM does not require a concrete valuation of placeholders – for instance, Listing 2 only specifies that `a` is an integer greater than 1. This user guide also specifies, via the declaration of `prime/1` as the only output predicate, that only atoms in the stable model that contain `prime/1` are considered when checking for external equivalence. We can verify that the program in Listing 1 is externally equivalent to the more efficient program in Listing 4 for *any* pair of integers (a, b) such that $a > 1$. If `primes.ug` is the file containing the user guide in Listing 2, ANTHEM can automatically prove the aforementioned claim.

```
sqrtb(M) :- M = 1..b, M*M <= b, (M+1)*(M+1) > b.
composite(I*J) :- sqrtb(M), I = 2..M, J = 2..b.
prime(I) :- I = a..b, not composite(I).
```

Listing 4 A refactored primes program, `primes.3.lp`.

To achieve this, ANTHEM first builds the completion of the τ^* formula representation of each of the programs (Sections 3.1 and 3.2.2) and then produces a series of subtasks

[3] Standard interpretations give a standard treatment to operations like integer addition.

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

to verify using VAMPIRE. These subtasks are described by Fandinno *et al.* (2023) and a detailed example is given in Section 5.3. Note that verifying the preceding example is difficult for the current version of VAMPIRE without some human help (see Section 5). It is also worth mentioning that currently, ANTHEM can only automatically verify external equivalence for programs that are tight and lack private recursion (Fandinno *et al.* 2023, Section 6). Note that strong equivalence verification does not suffer this restriction, and, in certain cases, the tightness limitation can be lifted (see Section 5).

4.3 Specification adherence

Thus far, we have discussed how ANTHEM can be used to establish that certain types of equivalences hold between two ASP programs. In this section, we show how ANTHEM can verify the adherence of an ASP program to a specification written in classical first-order logic. This workflow can be viewed as a special type of external equivalence checking, where instead of encoding our specification of desired behavior as an ASP program, we encode it directly in ANTHEM 's control language. For example, we can validate that the program given in Listing 5 correctly solves the exact cover [4] problem by directly encoding the properties our program should possess as follows.

```
{in_cover(1..n)}.
:- I != J, in_cover(I), in_cover(J), s(X,I), s(X,J).
covered(X) :- in_cover(I), s(X,I).
:- s(X,I), not covered(X).
```

Listing 5. An encoding (`cover.lp`) solving the exact cover problem.

First, a solution to this problem is a collection of set identifiers in the range $[1, n]$:

```
spec: forall Y (in_cover(Y) ->
        exists I$ (Y = I$ and I$ >= 1 and I$ <= n$i)).
```

Second, each element that occurs in the union of all sets must occur in the cover:

```
spec: forall X (exists Y s(X, Y) ->
                    exists Y (s(X, Y) and in_cover(Y))).
```

Finally, sets selected to be part of the cover cannot overlap:

```
spec: forall Y Z (exists X (s(X, Y) and s(X, Z))
                    and in_cover(Y) and in_cover(Z) -> Y = Z).
```

Keep in mind that we need to restrict our input to sets identified by the range $[1, n]$:

```
assumption: forall Y (exists X s(X, Y) ->
                exists I$ (Y = I$ and I$ >= 1 and I$ <= n$i)).
```

We can verify Listing 5 against this specification with the following user guide:

[4] An exact cover of a collection S of sets is a subcollection S' of S such that each element of the union of all sets in S belongs to exactly one set in S'.

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

```
input: n -> integer.            input: s/2.
output: in_cover/1.             assumption: n >= 0.
```

using the command

```
anthem verify --equivalence external cover.{lp,spec,ug}
```

This workflow tells us that `cover.lp` has the properties encoded by the specs and that our specification `cover.spec` completely defines the external behavior of the program in `cover.lp` with respect to this user guide. If we remove one of the specs from `cover.spec`, then the preceding command fails, since the specification is no longer a complete description of the program's behavior. However, the following command succeeds.

```
anthem verify --equivalence external cover.{lp,spec,ug} \
              --direction backward
```

This tells us that the program embodies the remaining properties of the specification.

More broadly, for any verification task, ANTHEM attempts to validate some form of equivalence. Thus, we can run the `forward` and `backward` directions of this equivalence proof separately if desired. In the case of external equivalence, we can frame our task as a proof of the equivalence between a program and a specification; our specification can be written as a logic program or as a collection of assumptions about the input and formulas annotated with the *spec* role.

5 ANTHEM in practice

One of the goals of this paper is to provide practical advice for using ANTHEM and demystify the interpretation of the results produced. Chances are your experience with ANTHEM will fall into one of three categories.

5.1 The good

This is the most straightforward case: ANTHEM returns a message like

```
> Success! Anthem found a proof of the theorem.
```

For instance, invoking the command from Section 4.1 produces an output summarizing the two subproblems passed to VAMPIRE and their respective success statuses. Here, all subproblems in both the forward and backward directions of the equivalence proof ended with a "Status: Theorem" message. This means that the subproblem conjecture was successfully derived from the subproblem axioms. Since every subproblem was verified, the programs `transitive.1.lp` and `transitive.2.lp` are strongly equivalent.

5.2 The bad

There are certain types of problems that ANTHEM is not equipped to address. The most common such problem is the task of validating the external behavior of a non-tight program, which the current incarnation of ANTHEM refuses to attempt. One notable exception is the case of *local tightness* (Fandinno *et al.* 2024). Many non-tight programs are still locally tight, such as programs encoding planning problems in which the law of inertia introduces positive recursion. While tightness is a syntactic property of programs that is easily checked by ANTHEM, an effective procedure for checking local tightness has not been developed. If, however, a user has manually proven the local tightness of their program(s), ANTHEM's tightness check can be bypassed, and verification safely completed, by adding the `--bypass-tightness` flag.

5.3 The ugly

When ANTHEM fails to verify a program, it is *not* a proof that the programs are not (strongly, externally) equivalent. It very well may be that the proof exists, but ANTHEM needs some help finding it. What can be done in these cases?

Option 1: Increase resource allocation. ANTHEM lets you increase the timeout `-t` for the backend ATP for each subproblem. You can also parallelize search by increasing the number of cores used by the ATP with the `-m` flag.

Option 2: Explore missing or malformed assumptions. If ANTHEM is hung up on a particular subproblem, consider the axiom set and conjecture. Is it clear that the conjecture follows from the axioms? Sometimes seemingly self-evident assumptions are missing – for a detailed example, see the "orphan" example by Fandinno *et al.* (2023).

Option 3: Write a proof outline. If VAMPIRE is unable to validate a subproblem in a reasonable amount of time (as is often the case for nontrivial tasks), the next step is to find a lemma that can be derived by VAMPIRE from the axioms without help and that is likely to facilitate achieving the goal when added to the list of axioms. Thus, for external equivalence tasks, ANTHEM lets users supply a *proof outline* consisting of annotated formulas that can play three roles: definitions, lemmas, and inductive lemmas. These are target language formulas augmented with instructions for how they should be used within a verification task. Their general form is

```
role(direction)[name]: formula.
```

where `role` and `formula` are required, and `direction` indicates which direction of the equivalence proof the formula is used within. Such outlines can often be very effective.

Definitions are assumed to define the extent of a new predicate introduced for convenience within a proof outline. They have the form

```
definition: forall X ( p(X) <-> F(X) ).
```

where `X` is a tuple of variables, `p` is a fresh predicate symbol, and `F` is a formula with free variables `X`. A sequence of definitions is valid if any defined predicate p used within each F is defined previously in the sequence. For example, the annotated formula (D)

```
definition[D]: forall I$ N$ (sqrt(I$,N$) <->
               I$ >= 0 and I$*I$ <= N$ < (I$+1)*(I$+1)).
```

defines the integer square root (I) of an integer N. Definitions can be used in a proof similarly to assumptions as their purpose is to make writing lemmas easier.

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

Lemmas. ANTHEM interprets proof outlines in an intuitive way: as a series of intermediate claims to be checked en route to a final claim. Previously established results in this sequence are used as assumptions when checking the next claim. When ANTHEM is provided with a proof outline as part of an external equivalence verification task, it attempts to sequentially verify every (inductive) lemma. If successful, it treats the resulting set of formulas as assumptions during the verification task. For example, a proof outline consisting of D and the following two lemmas (L_1 and L_2) is interpreted as an instruction to derive L_1 from D and L_2 from $\{D, L_1\}$.

```
lemma[L1]: sqrt(I$,N$) and (I$+1)*(I$+1) <= N$+1 -> sqrt(I$+1,N$+1).
inductive-lemma[L2]: N$ >= 0 -> exists I$ sqrt(I$,N$).
```

Inductive lemmas have the general form

```
inductive-lemma: forall X N$ ( N$ >= n -> F(X,N$) ).
```

where `n` is an integer, `X` is a tuple of variables, `N` is an integer variable and `F` is a target language formula. Within a proof outline, an inductive lemma is interpreted as an instruction to prove two conjectures:

$$\forall X F(X, n) \qquad \text{and} \qquad \forall X N(N \geq n \wedge F(X, N) \rightarrow F(X, N+1)).$$

If both the first (the base case) and the second (the inductive step) conjectures are proven, then the original formula

$$\forall X N(N \geq n \rightarrow F(X, N))$$

is treated as an axiom in the remaining proof steps. For example, L_2 is interpreted as an instruction to verify

$$\exists I sqrt(I, 0) \qquad \text{and} \qquad \forall N(N \geq 0 \wedge \exists I sqrt(I, N) \rightarrow \exists J sqrt(J, N+1)).$$

We can verify the external equivalence of Listings 1 and 4 with respect to the user guide in Listing 2 with the proof outline `primes.po`, which extends $\{D, L_1, L_2\}$ with the following lemmas, L_3 and L_4:

```
lemma[L3]: b >= 1 -> (sqrtb(I$) <-> sqrt(I$,b)).
lemma[L4]: I$ >= 0 and J$ >= 0 and I$*J$ <= b and sqrtb(N$)
                                  -> I$ <= N$ or J$ <= N$.
```

Let us denote the assumption `a > 0` from Listing 2 as A. Additionally, let us denote the completion of Rule 1 from Listing 1 as F, the completion of Rule 2 from Listing 1 as F', the completion of rules 1 and 2 from Listing 4 as G, and the completion of Rule 3 from Listing 4 as G'. When the following command is invoked

```
anthem verify --equivalence external primes.{1.lp,3.lp,ug,po}
```

ANTHEM verifies the proof outline as described above, then verifies the following subproblems in which the completed definitions of private predicates are treated as assumptions from which to derive the equivalence of the public predicates' definitions:

Table 1. *Tools supporting proof-based verification of ASP programs*

	Language Features				Equivalence				
Tool	Vars.	Disj.	Dbl. Neg.	Arith.	Strong	External	Weak	Spec	Backend
SELP		√			√				SAT
DLPEQ		√			√		√		ASP
CCt		√			√		√		QBF
ANTHEM-SE	√		√	√	√				ATP
ANTHEM-1	√		√	√				√	ATP
ANTHEM-P2P	√		√	√		√	√		ATP
ANTHEM 2.0	√		√	√	√	√	√	√	ATP

$$(A \land L_1 \land L_2 \land L_3 \land L_4 \land F(\mathbf{p}) \land G(\mathbf{q}) \land F'(\mathbf{p})) \to G'(\mathbf{q}),$$
$$(A \land L_1 \land L_2 \land L_3 \land L_4 \land F(\mathbf{p}) \land G(\mathbf{q}) \land G'(\mathbf{q})) \to F'(\mathbf{p}).$$

Here, $\mathbf{p}$ is a list of fresh predicate symbols, $\{composite/1\}$, to replace the private predicates occurring in `primes.1.lp`. Similarly, $\mathbf{q}$ replaces the private predicates occurring in `primes.3.lp` with predicate symbols that won't conflict with $\mathbf{p}$: $\{sqrtb/1, composite\_p/1\}$.

A challenging example Fandinno *et al.* (2024) provide two alternative solutions (Sections 1 and 5) to the frame problem, using different approaches to encoding the law of inertia. While these programs are not tight, they are locally tight, allowing us to safely bypass the tightness check. If we provide ANTHEM with the following proof outline

```
inductive-lemma(backward): N$ >= 0 -> (in(X,Y,N$) -> person(X) )
```

we can verify the external equivalence of the two frame programs. This example is remarkable due to the complexity of the programs (which use powerful modeling features to encode a realistic problem from the ASP literature), and the fact that we can prove external equivalence for arbitrary time horizons $h \geq 0$ using a single inductive lemma.

6 Experimental analysis

This section compares the capabilities (Table 1) and performance (Table 2) of ANTHEM 2.0 against related systems, particularly its predecessors. As mentioned in the Introduction, tools from the ANTHEM family are oriented towards a different class of logic programs than SELP, DLPEQ, and CCT, which were designed for propositional SMODELS or DLV programs with disjunctive heads. As shown in Table 1, they do not support variables, double negation, or arithmetic. Hence, there cannot be a sensible runtime comparison between these systems and ANTHEM 2.0. Furthermore, they employ a different type of backend inference engine – the common theme of these systems is to convert a verification task into a satisfiability problem for some form of solver. SELP, DLPEQ, and CCT use SAT, ASP, and Quantified Boolean Formula (QBF) solvers, respectively. ANTHEM systems, on the other hand, use an automated theorem prover (ATP) as a backend.

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

Table 2. *Comparing ANTHEM 2.0 against its predecessors*

Problem	Equivalence	PO	Competitor	Competitor Runtime $m=2$	$m=4$	$m=8$	ANTHEM 2.0 Runtime $m=2$	$m=4$	$m=8$
Coloring	External		ANTHEM-1	0.15	0.21	0.20	0.13	**0.12**	**0.12**
Cover (1vs)	External		ANTHEM-1	0.40	0.22	0.15	0.11	**0.10**	**0.10**
Cover (1v2)	External		ANTHEM-P2P	0.28	0.25	0.32	0.13	**0.11**	0.13
Division	External		ANTHEM-1		Timeout			Timeout	
Division	External	√	N/A		-		6.20	3.21	**1.68**
Frame	External		ANTHEM-P2P		Timeout			Timeout	
Frame	External	√	N/A		-		36.69	17.97	**7.88**
Primes (1v2)	External		ANTHEM-P2P	5.57	7.21	2.94	2.56	1.14	**0.54**
Primes (2v3)	External		ANTHEM-P2P		Timeout			Timeout	
Primes (2v3)	External	√	N/A		-		95.57	48.14	**29.47**
Primes (2vs)	External		ANTHEM-1		Timeout		6.44	3.14	**1.77**
Bounds	Strong		ANTHEM-SE	0.25	0.15	**0.07**	1.80	0.99	0.48
Choice	Strong		ANTHEM-SE		Timeout		**0.04**	0.09	0.07
Squares	Strong		ANTHEM-SE	17.37	141.43	69.45	5.71	2.53	**1.20**
Successor	Strong		ANTHEM-SE	0.86	0.45	**0.25**	1.26	0.64	0.40
Transitive	Strong		ANTHEM-SE	0.06	**0.05**	0.06	0.07	0.11	**0.05**

In addition to supporting several useful language features as shown in Table 1, ANTHEM 2.0 also permits users to verify multiple types of equivalence. For example, it supports specification adherence verification in the style of ANTHEM-1, implemented as a special case of external equivalence. This type of verification is unique – all other types in the table refer to a form of equivalence between logic programs. Interestingly, ANTHEM-P2P and ANTHEM 2.0, by virtue of their ability to verify external equivalence of logic programs, also support weak equivalence. Weak (or answer set) equivalence between programs Π_1 and Π_2 can be verified by comparing them under a user guide without placeholders, assumptions or input declarations, that contains every predicate in $\Pi_1 \cup \Pi_2$ as an output predicate. Note that CCT supports a form of relativized uniform equivalence with projection similar to external equivalence when placeholders are forbidden, in addition to a generalized form of strong equivalence relativized to a propositional signature.

Recall that ANTHEM 2.0 was an effort to integrate, stabilize, and extend the capabilities of previous prototypes developed for one type of equivalence only. As such, there is at most one competitor to ANTHEM 2.0 for each problem in Table 2. The problems considered are a subset of the repository's **res/examples** folder, excluding only trivial parsing tests. Times are given in seconds, the m parameter represents the number of cores allocated to VAMPIRE, and the "PO" column indicates if a proof outline was provided to aid in solving the problem within the 5-minute time limit. All experiments were conducted on Ubuntu 24.04.2 LTS, 13th Gen Intel(R) Core(TM) i7-1370P, 32 GB RAM.

These comparisons demonstrate that ANTHEM 2.0 is considerably more powerful than its predecessors when applied to non-trivial problems, both in terms of capabilities and

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

performance. In particular, the new features of proof outlines (definitions and inductive lemmas) have enabled us to verify problems that previous systems could not address in a reasonable amount of time. Consider the challenging Frame problem discussed at the end of Section 5. While even ANTHEM 2.0 was initially unable to address this problem within the 5-minute timeout, the use of a proof outline with the new inductive lemma feature brought the runtime down to 7.88 s. Indeed, Division, Frame, and Primes (2v3) all rely on proof outlines with features specific to ANTHEM 2.0.

For certain trivial problems (Bounds, Successor, Transitive), ANTHEM-SE outperforms ANTHEM 2.0, although the differences for the Transitive problem are negligible. For the other two problems, this is likely due to a shortcut employed by ANTHEM-SE when it encounters *positive programs*. A positive rule is a basic rule or constraint whose body does not contain negation. For simple programs of this nature, the γ transformation can be omitted when checking strong equivalence (Lifschitz *et al.* 2019, Proposition 6).

7. Conclusions and future work

This paper provides an overview of the ANTHEM 2.0 system. We strongly encourage interested readers to also visit the ANTHEM repository, where a user manual and set of examples with expected outputs may be found. The systems developed in the ANTHEM project are unique within the landscape of (proof-based) ASP verification tools due to their support for programs with variables and integer arithmetic. ANTHEM 2.0 subsumes previous ANTHEM systems and provides powerful new features such as natural translation for strong equivalence tasks and enhanced proof outlines for external equivalence problems. Proof outlines, for instance, make previously unsolved problems such as the external equivalence of Listings 1 and 4 tractable. Additionally, the (optional) decoupling of the translation and verification steps may prove to be a useful tool for studying the relationship between ASP and other logical formalisms such as classical first-order logic.

This tool offers a stable foundation for future innovation. A number of improvements are already planned or in progress, including improved algorithms for simplifying formulas, proof outlines for strong equivalence tasks, employing alternative backend ATP systems, integration of natural translation into external equivalence verification, bypassing tightness restrictions with ordered completion and/or tightening, extending the supported language with conditional literals, supporting counting and/or unrestricted aggregates, and revising the definition of integer division for consistency with CLINGO (Fandinno and Lifschitz 2023a, Footnote 3). The long-term goal is to support the complete ABSTRACT GRINGO language within ANTHEM.

Acknowledgements

We are thankful to the anonymous reviewers for their valuable feedback. This research is partially supported by NSF CAREER award 2338635 and the research development program at the University of Nebraska Omaha. 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.

References

Clark, K. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, 293–322. https://doi.org/10.1007/978-1-4684-3384-5_11.

Fandinno, J., Hansen, Z., Lierler, Y., Lifschitz, V. AND Temple, N. 2023. External behavior of a logic program and verification of refactoring. *Theory and Practice of Logic Programming* 23, 4, 933–947. https://doi.org/10.1017/s1471068423000200.

Fandinno, J. AND Lifschitz, V. 2023b. On Heuer's procedure for verifying strong equivalence. In Proceedings of the Eighteenth European Conference on Logics in Artificial Intelligence (JELIA'23), S. Gaggl, M. Martinez and M. Ortiz, Eds. Vol. 14281 of Lecture Notes in Computer Science. Springer-Verlag, 253–261. https://doi.org/10.1007/978-3-031-43619-2.

Fandinno, J. AND Lifschitz, V. 2023a. Omega-completeness of the logic of here-and-there and strong equivalence of logic programs. *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning* 19, 1, 240–251. https://doi.org/10.24963/kr.2023/24 August, ISSN 2334-1033

Fandinno, J., Lifschitz, V., Lühne, P. AND Schaub, T. 2020. Verifying tight logic programs with anthem and vampire. *Theory and Practice of Logic Programming* 20, 5, 735–750. https://doi.org/10.1017/s1471068420000344.

Fandinno, J., Lifschitz, V. AND Temple, N. 2024. Locally tight programs. *Theory and Practice of Logic Programming* 24, 5, 942–972. https://doi.org/10.1017/S1471068423 00039X.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V. AND Schaub, T. 2015. Abstract gringo. *Theory and Practice of Logic Programming* 15, 4-5, 449–463. https://doi.org/10.1017/S1471068415000150.

Harrison, A., Lifschitz, V., Pearce, D. AND Valverde, A. 2017. Infinitary equilibrium logic and strongly equivalent logic programs. *Artificial Intelligence* 246, 22–33. https://doi.org/10.1016/j.artint.2017.02.002.

Heuer, J. 2020. Automated verification of equivalence properties in advanced logic programs. Bachelor's thesis, University of Potsdam. https://arxiv.org/abs/2310.19806.

Heyting, A. 1930. Die formalen regeln der intuitionistischen logik. In *Sitzungsberichte der Preussischen Akademie der Wissenschaften*. Deutsche Akademie der Wissenschaften zu Berlin, 42–56.

HUTCHISON, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y. AND Weikum, G. 2005. SELP—a system for studying strong equivalence between logic programs. In Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05), C. Baral, G. Greco, N. Leone and G. Terracina, Eds. Vol. 3662 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 442–446. https://doi.org/10.1007/11546207_43.

Janhunen, T. AND Oikarinen, E. 2004. LPEQ and DLPEQ — translators for automated equivalence testing of logic programs. In Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04), V. Lifschitz and I. Niemelä, Eds. Vol. 2923 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 336–340. https://doi.org/10.1007/978-3-540-24609-1_30.

Kovács, L. AND Voronkov, A. 2013. First-order theorem proving and vampire. In Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013, N. Sharygina and H. Veith, Eds. Vol. 8044 of Lecture Notes in Computer Science, Springer, 1–35. Proceedings, https://doi.org/10.1007/978-3-642-39799-8_1

https://doi.org/10.1017/S1471068425100112 Published online by Cambridge University Press

LIFSCHITZ, V. 2021. Transforming gringo rules into formulas in a natural way. In Proceedings of the Seventeenth European Conference on Logics in Artificial Intelligence (JELIA'21), W. FABER, G. FRIEDRICH, M. GEBSER and M. MORAK, Eds. Vol. 12678 of Lecture Notes in Computer Science, Springer-Verlag, 421–434. https://doi.org/10.1007/978-3-030-75775-5_28.

LIFSCHITZ, V., LÜHNE, P. AND SCHAUB, T. 2019. Verifying strong equivalence of programs in the input language of GRINGO. In Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19), M. BALDUCCINI, Y. LIERLER and S. WOLTRAN, Eds. Vol. 11481 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 270–283. https://doi.org/10.1007/978-3-030-20528-7_20.

LIFSCHITZ, V., PEARCE, D. AND VALVERDE, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic* 2, 4, 526–541. https://doi.org/10.1145/383779.383783.

OETSCH, J., SEIDL, M., TOMPITS, H. AND WOLTRAN, S. 2009. Testing relativised uniform equivalence under answer-set projection in the system cct. In *Applications of Declarative Programming and Knowledge Management*, D. SEIPEL, M. HANUS and A. WOLF , Eds., Springer, Berlin, Heidelberg, 241–246. https://doi.org/10.1007/978-3-642-00675-3_16.

OETSCH, J. AND TOMPITS, H. 2008. Program correspondence under the answer-set semantics: The non-ground case. In Proceedings of the Twenty-Fourth International Conference on Logic Programming (ICLP'08), M. G. DE LA BANDA and E. PONTELLI, Eds. Vol. 5366 of Lecture Notes in Computer Science, Springer-Verlag, 591–605. https://doi.org/10.1007/978-3-540-89982-2_49.

PEARCE, D. 2006. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence* 47, 1-2, 3–41. https://doi.org/10.1007/s10472-006-9028-z.

PEARCE, D., TOMPITS, H. AND WOLTRAN, S. 2001. Encodings for equilibrium logic and logic programs with nested expressions. In Proceedings of the Tenth Portuguese Conference on Artificial Intelligence (EPIA'01), P. BRAZDIL and A. JORGE, Eds. Vol. 2258 of Lecture Notes in Computer Science, Springer-Verlag, 306–320. https://doi.org/10.1007/3-540-45329-6_31.

SUTCLIFFE, G. 2017. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning* 59, 4, 483–502. https://doi.org/10.1007/s10817-017-9407-7.

VALVERDE, A. 2004. tabeql: A tableau based suite for equilibrium logic. In *Logics in Artificial Intelligence*, J. J. ALFERES and J. LEITE, Eds., Springer, Berlin, Heidelberg, 734–737. https://doi.org/10.1007/978-3-540-30227-8_69, 978-3-540-30227-8