DRAFT

SEGMENT
The Halting Problem and Software Verification

Jack K. Horner
Last modified: 10/13/2021 7:42 AM

Abstract

In practice, we cannot test all paths in a typical software system S consisting of more than a few hundred source lines of code. This limit is essentially determined by the limitations of human physiology and light time-of-flight considerations. Are there in-principle non-physical limits to the verification of software? There are several, the best known of which is the Halting Problem. In this section, we describe the Halting Problem and its implications for software verification. As part of this account, we provide what we believe to be a novel proof of that Problem.

1.0 Introduction

For the purpose of this segment, we define a *software system* S to be an executable sequence of instructions written in a Turing-complete language (for the definition of a Turing-complete language, see Boolos, Burgess, and Jeffrey 2007, Chap. 3) and optional[1] non-executable text such as blank lines and comments. A *binary branch* in S is a conditional instruction equivalent to the form "If X, do Y". We define a *path* in S (Symons and Horner 2014) to be an executable sequence of statements S, beginning with system start and ending with system exit (end).[2] We define a *source line of code* (SLOC) as any non-comment, non-blank statement in a software system. (For a more precise definition of SLOC, see Boehm, Abts, Brown et al. 2000, Appendix E, Section 1.1.4.) By *testing*, we mean an activity that, among other things, involves executing S on a Turing machine.

As we argued in Segment TBD, we cannot test all paths in S if S consists of than a few hundred SLOC and has at least one binary branch, on average, per 10 SLOC.[3] This limit is variously determined by contingencies of human performance and light time-of-flight considerations.

Are there in-principle limits to software verification that are not reducible to human performance or physical theory constraints? There are several, the best known of which the *Halting Problem* (Church 1936; Turing 1936):

[1] Optional for compilers, but often not so for human comprehension (see ISO 2017; Boehm, Abts, Brown et al. 2000).
[2] Several definitions of "path" are possible (see, for example, Watson and McCabe 1996).
[3] The average frequency of binary branches in actual well-structured applications is about one per 10 SLOC (see, for example, Watson and McCabe 1996).

(HP) There is no effective procedure[4] *p* that, given *any* Turing machine (Boolos, Burgess,and Jeffrey 2007, Chap. 3) M, and given any positive integer n as input to M, *p* can determine whether M, given n as input, ever halts.[5]

Although as formulated, (HP) might seem to be restricted to software systems that reference positive integers, it can be shown that any software system on a Turing machine is equivalent to a software system that satisfies the specification in (HP), via a procedure called *arithmetization* (see Boolos, Burgess, and Jeffrey 2007, Chap. 3 for details.)[6]

We can, of course, determine by inspection that some programs halt, and that some never will. In other cases, we can't tell whether a program will halt, as the following three examples show.

***Example 1*:**

```
while (true) continue
```

never halts.

***Example 2*:**

```
if (true) halt
```

---

[4] As used here, a procedure or method is *effective* for a class of problems iff (Hunter 1971, pp. 13-15)

- it consists of a finite number of exact, finite instructions
- when applied to a problem from its class, it always finishes (*terminates*) after a finite number of steps
- when applied to a problem from its class, it always produces a correct answer.

[5] Certain non-Turing constructs, called *hypercomputation*, that are claimed to be able to solve the Halting Problem, have been proposed (for a somewhat dated introduction to the topic, see Ord 2006). Hypercomputation lies outside the scope of this book.

[6] In addition to being a fundamental limit to the verifiability of software systems, the Halting Problem has a profound implication for a closely related and fundamental question in logic and epistemology:

> (DP) Is there is an effective procedure for determining whether a sentence D follows from a finite set G of sentences, for any D and any G?

(DP) is called the *decision problem* (*Entscheidungsproblem*). In its modern formulation, it is generally attributed to Hilbert (see, for example, Hilbert and Ackermann 1928). In various ways, Aristotle 350 BCE and Leibniz 1685 (p. 51) anticipate (DP).

It can be shown that, given any Turing machine M and any input *n* to M, we can effectively write a finite set of sentences G and a sentence D such that G implies D if and only if M eventually halts (see Boolos, Burgess, and Jeffrey 2007, Chap. 11). Given that there is no effective procedure for solving the Halting Problem on a Turing machine, *the Decision Problem is not solvable*. This is a fundamental limit on the extent to which information or knowledge can be structured in terms of deductive relations (Hempel 1948; Salmon 1967, pp. 1-5).

always halts.

Example 3 (adapted from Strachey 1965):

A *total* function is a function that is defined for all input values. Suppose there exists a total Turing-computable function *halts(f)* that returns true if the routine *f* halts (when run with no inputs) and returns false otherwise. Consider the following routine:

```
def g():
     if halts(g) :
          loop_forever()
```

halts(g) must either return true or false, because *halts* is assumed to be total. If *halts(g)* returns true, then *g* will call *loop_forever* and never halt, which is a contradiction. If *halts(g)* returns false, then *g* will halt, because it will not call *loop_forever*, which is also a contradiction. Thus, we cannot tell whether the example will halt.[7]

(HP), combined with a slight generalization of an argument in Symons and Horner 2014/2017, has a further sobering implication. They jointly imply that we cannot, in all cases of interest, verify by *testing* whether a typical software system S (in practice, > ~300 SLOC, with, on average, at least one binary branch per 10 SLOC) that executes on a Turing machine has a given property P. More specifically, let P be "contains a segment H whose halting behavior (halts, or not) is not determinable by an effective procedure". It follows from the path explosion catastrophe (see Segment TBD) that we cannot, in almost all cases of interest, verify by testing whether S contains a segment H whose halting behavior is not determinable by an effective procedure. And from this it follows that we cannot verify by testing whether S's halting behavior is determinable by an effective procedure. Put another way, we cannot determine by *testing*, whether most software systems halt (under every possible input).

There are several formulations of the Halting Problem (see Moore and Mertens 2011); some are more rigorous and explicit than others.[8] Rendering the Halting Problem in a first-order language (FOL) is one way to avoid the vagaries of natural languages and to make explicit what inference rules are in play (Church 1956, Section 00).

Burkholder 1987 provides an FOL rendering of (HP). Informally stated, the assumptions of Burkholder's formulation, which we will call "p_n", where n = 1, 2, 3, 4, are (where "program" is assumed to be a program that can execute on a Turing machine) are:

> **p_1**. If there an *algorithm* x that solves the Halting Problem, then there exists a *program* w that solves the Halting Program.

[7] Example 3 can be read as an informal proof of (HP).

[8] Strachey 1965 (see Example 3, above)), for example, is not as rigorous as Church 1936, Turing 1936, or Burkholder 1987.

p2. If program y halts given input z, then y halts given as input the pair <y,z> and prints out "good". If y does not halt given z, then w halts on <y,z> and prints out "bad". (This premise characterizes what is meant by "a program x is able to decide whether y halts or does not halt, given input z".)

p3. If program w can decide whether program y halts, given y as input, then there exists a different program v that can determine only whether a program halts given itself as input. (This premise deals with the special case in which program y is given itself as input.)

p4. If a program v of the kind mentioned in p3 exists, then there exists a program (that is slightly different from v) that loops indefinitely in exactly those cases when v would halt and print out "good".

To be proven (the Halting Problem Theorem**)**: The Halting Problem is unsolvable on a Turing machine.

In order to state Burkholder's 1987 formulation of the Halting Problem rigorously, we need to introduce some notation. Burkholder happens to use the FOL symbology of Pelletier 1986. Accordingly, let:

→ : if-then

~ : not

A : for all

E : there exists

F, G, …, P, Q, … (perhaps with subscripts) : predicate letters

x,y,z,w : individual variables

a,b,c,… : individual constants

In addition, let:

Gx : x is an algorithm

Px : x is a program that can execute on a Turing machine

Dxyz : x is able to decide whether y halts, given input z

H_2xy : x halts, given input y

H_3xyz : x halts, given as input the pair <y,z>[9]

Oxg : x outputs "good" (meaning "x halts")

Oxb : x outputs "bad" (meaning "x doesn't halt")

Given these conventions, Figure 1 shows Burkholder's 1987 formulation of the Halting Problem:

(1) $(Ex)[Gx \ \& \ (Ay)(Py \rightarrow (Az)Dxyz)] \rightarrow (Ew)[Pw \ \& \ (Ay)(Py \rightarrow (Az)Dwyz)]$

(2) $(Aw)([Pw \ \& \ (Ay)(Py \rightarrow (Az)(Dwyz)] \rightarrow (Ay)(Az)([Py \ \& \ H_2yz) \rightarrow (H_3wyz \ \& \ Owg)] \ \& \ [(Py \ \& \ \sim H_2yz) \rightarrow (H_3wyz \ \& \ Owb)]))$

(3) $(Ew)(Pw \ \& \ (Ay)([Py \ \& \ H_2yy) \rightarrow (H_3wyy \ \& \ Owg)] \ \& \ [(Py \ \& \ \sim H_2yy) \rightarrow (H_3wyy \ \& \ Owb)])] \rightarrow (Ev)[Pv \ \& \ (Ay)([Py \ \& \ H_2yy) \rightarrow (H_2vy \ \& \ Ovg)] \ \& \ [(Py \ \& \ \sim H_2yy) \rightarrow (H_2vy \ \& \ Ovb)])]$

(4) $(Ev)[Pv \ \& \ (Ay)([Py \ \& \ H_2yy) \rightarrow (H_2vy \ \& \ Ovg] \ \& \ [(Py \ \& \ \sim H_2yy) \rightarrow (H_2vy \ \& \ Ovb)])] \rightarrow (Eu)[Pu \ \& \ (Ay)([Py \ \& \ H_2yy) \rightarrow \sim H_2yy] \rightarrow \sim H_2uy] \ \& \ [(Py \ \& \ \sim H_2yy) \rightarrow (H_2uy \ \& \ Oub)])]$

The Halting Problem Theorem is

$\sim(Ex)[Gx \ \& \ (Ay)(Py \rightarrow (Az)(Dxyz)]$

Figure 1. Burkholder's 1987 FOL formulation of the Halting Problem.

2.0 Method and results

Let X and Y be sentences (say, in a FOL) and suppose that Y can be derived from X in some derivation system.[10] An *automated deduction framework* (ADF; see, for example, Quaife 1992, Chap. 1; Wos 1988, Chap. 1; Kovács and Voronkov 2013; Sutcliffe and Suttner 2021) is a procedure/program that can, in favorable circumstances (in any case, excluding cases subject to the Halting Problem), produce a derivation (deduction) of Y, given X. Several ADFs can, in

[9] In Burkholder 1987, several instances of predicates that should be of the form" H_3xyz" are erroneously written as "H_2xyz". It's clear from context that Burkholder intended "H_2" to denote an arity-2 predicate and "H_3" to denote an arity-3 predicate. We have corrected those subscripting errors in the present discussion.

[10] Rigorously specifying what a derivation system is requires specifying the language, axioms, and the set of derivation rules of that system. See, for example, Church 1956.

favorable cases, also prove that a finite set, F, of sentences, has a model, allowing us to investigate the consistency and independence of F.

Rendering a problem in an FOL in principle makes the solution of the problem accessible to an FOL ADF and that is the approach we take in the following.

Why would one want to take an ADF approach to proving the Halting Problem, when we already have widely accepted proofs of (HP) in Church 1936 and Turing 1936? There are good reasons for doing so:

A. First, as published, many derivations (proofs) in mathematics/mathematical logic are partially elliptical; Church 1936 and Turing 1936 are no exception. That is, almost all such proofs implicitly assume some derivation steps that informed readers are assumed to know. Such implicit steps may or may not be as robust or as well-known as they may seem. The only way to address this problem is to make the implicit steps explicit. This often requires recasting the proofs in a more fundamental representation than the original proofs used.[11] Doing so without the assistance of automation, however, can and often does greatly increase the length of the proof. Lengthening a proof without automated assistance can, and often does, have one or more problematic consequences:
    a. The longer a "manually"-generated proof is, the greater the probability that it contains transcription errors.
    b. The effort may be intractable for humans.

    To help mitigate (1a) – (1b), we need to augment human efforts with automation that can generate the requisite proof details at least as reliably and faster than a human can, i.e., we need to use an ADF.

B. Novel proofs help to corroborate existing proofs. ADF-generated proofs are often novel compared to non-ADF-generated proofs.

C. Because ADFs can often produce proofs of (or counterexamples to) a conjecture, they allow us to explore, in favorable cases, some sensitivities of formalized theories much faster than we could without ADFs.

D. ADFs have in fact proven useful for investigating many problems in finitely axiomatized FOL systems (see Sutcliffe and Suttner 2021 for thousands of examples).

E. Several famous longstanding conjectures in mathematics, including the Four Color (map) Conjecture (Appel and Haken 1989), and Kepler's (sphere-packing) Conjecture (Hales and Ferguson 2011), have to date been solved only with extensive ADF assistance

Even if we grant (A) – (E), it could be objected, ADFs are not a panacea, because:

i. They involve software, and, as this book argues, we cannot, in all cases of interest, determine whether software is error-free.

[11] Compare, for example, many of the ADF proofs of the theorems in Sutcliffe and Suttner 2021 with the proofs in the references listed in the prologues of those problems in Sutcliffe and Suttner 2021.

ii. They can generate proofs so long or complicated that humans cannot understand them.
iii. To be useful, they often require humans to first formulate good conjectures.
iv. They are not guaranteed to prove all sentences known to be theorems.

Although (i) – (iv) might seem to diminish the force of the reasons given above ((A) - (E)) for using ADFs, they do not, for at least two reasons. First, properly understood, ADFs are tools that are intended to assist, not replace, human efforts (Wos 1988, esp. Chap. 1). Second, (i) – (iv) are not unique to ADFs: they are also limitations of non-ADF-assisted human effort.

VAMPIRE (Kovács 2021; Kovács and Voronkov 2013) is a FOL ADF.[12] In the following, we use VAMPIRE and Burkholder's 1987 first-order-language (FOL) formalization[13] of the Halting Problem shown in Section 1.0 to prove that

a. Premises (1) – (4) are independent (Chang and Keisler 2012, Exercise 1.2.18).
b. Premises (1) – (4) are consistent (Tarski 1953, p. 12; Chang and Keisler 2012, Table 1.3.1).
c. the Halting Problem cannot be solved (on a Turing machine)

To achieve these objectives, we must first translate the FOL formulation of (HP) shown in Figure 1 to symbology that is recognized by VAMPIRE. Figure 2 shows the translation of the FOL symbology of Figure 1 to a symbology that satisfies VAMPIRE's input requirements.[14] In this list, the FOL symbology appear on the left-hand-side of the colon; the corresponding VAMPIRE symbology, on the right-hand-side of the colon.

u : U

v : V

x : X

y : Y

w : W

z : Z

Ex : ? [X]

Ax : ! [X]

~ : ~

[12] VAMPIRE has repeatedly won "best-in-class" awards in the annual CADE Automated Theorem Proving (ATP) (CASC) competitions (see Sutcliffe and Suttner 2021, http://tptp.cs.miami.edu/CASC/).
[13] Bruschi 1991 appears to be the earliest ADF-based proof of the Halting Problem based on Burkholder's 1987 FOL formulation. Some variants of Bruschi's 1991 formulation can be found in Sutcliffe and Suttner 2021.
[14] The resulting symbology is in the TPTP language. For a BNF definition of that language, see Sutcliffe and Suttner 2021.

& : &

→ : =>

Gx : algorithm(X)

Px : program(X)

Dxyz : decides(X,Y,Z)

H_2xy : halts2(X,Y)

H_3xyz : halts3(X,Y,Z)

Oxg : outputs(X,good)

Oxb : outputs(X,bad)

Figure 2. Translation of the symbology of Figure 1 to VAMPIRE symbology.

The build system for VAMPIRE version 4.5.1 was downloaded from Kovács 2021 and used to create a VAMPIRE executable under each of *Windows 10/Cygwin64* (Cygwin authors 2021)[15] and *Red Hat Fedora* (Red Hat 2021). The VAMPIRE input file (file `COM003+1.p`) corresponding to the first-order formalization of Burkholder 1987 was downloaded from Sutcliffe and Suttner 2021, edited as noted in the caption for Figure 3, and executed under VAMPIRE running under each of

I. *Windows 10/Cygwin64* (a Linux look-alike) version CYGWIN_NT-10.0 3.2.0(0.340/5/3) on a Dell Inspiron 545 with an Intel Core2 Quad CPU Q8200 @ 2.33 GHz, 8.00 GB RAM, and ~0.5 TB free electromechanical-disk space
II. *Windows 10/Cygwin64* (a Linux look-alike) version CYGWIN_NT-10.0 3.2.0(0.340/5/3) on a Dell Inspiron 7506 2n1 with an Intel i7-1165 4-core CPU clocked at 2.8 GHz, 16 GB RAM, and ~0.8 TB free solid-state-disk space
III. *Red Hat Fedora* version 4.18.19-100.fc27.x86_64 on a Dell Inspiron 3650 with an Intel Core i5-6400 4-core CPU clocked at 2.7 GHz, 12 GB RAM, and ~0.7 TB free electromechanical-disk space

using the command

[15] The README files bundled in Kovács 2021 provide a high-level description of how to build VAMPIRE from the source code in Kovács 2021. Building VAMPIRE from Kovács 2021 requires a step that is not documented those README files. In particular, one must change the working directory from "`build`" to "`vampire-4.5.1`" before running "`make -j8`". In addition, when the VAMPIRE is built using the cmake script contained in Kovács 2021, the Cygwin C++ compiler issues hundreds of non-fatal warning messages (all of the same kind) about the source. The "same" build procedure, run under Red Hat Fedora (a variety of Linux) defaults, produces the same compiler warning messages. A cmake script in Kovács 2021 can be modified to suppress those warning messages via a compiler flag.

```
$ ../bin/vampire_rel.exe COM003+1.p > COM003+1.p.txt
```

where

$ is the [Cygwin | Linux|] prompt
`vampire_rel.exe` is the VAMPIRE executable
`COM003+1.p.txt` (shown in Appendix 1) is the output of the execution

```
%------------------------------------------------------------------------------
% File     : Derived from COM003+1 : TPTP v7.5.0. Released v2.0.0.
% Domain   : Computing Theory
% Problem  : The halting problem is undecidable
% Version  : [Bur87b] axioms.
% English  :

% Refs     : [Bur87a] Burkholder (1987), The Halting Problem
%          : [Bur87b] Burkholder (1987), A 76th Automated Theorem Proving Pr
% Source   : [Bur87b]
% Names    : - [Bur87b]

% Status   : Theorem
% . . . [comments deleted here by JKH, 12 September 2021]

%------------------------------------------------------------------------------
fof(p1,axiom,
    ( ? [X] :
        ( algorithm(X)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(X,Y,Z) ) )
   => ? [W] :
        ( program(W)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(W,Y,Z) ) ) )).

fof(p2,axiom,
    ( ! [W] :
        ( ( program(W)
          & ! [Y] :
              ( program(Y)
             => ! [Z] : decides(W,Y,Z) ) )
       => ! [Y,Z] :
            ( ( ( program(Y)
                & halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,bad) ) ) ) ) )).

fof(p3,axiom,
```

```
    ( ? [W] :
        ( program(W)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,bad) ) ) ) )
   => ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) ) )).

fof(p4,axiom,
    ( ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) )
   => ? [U] :
        ( program(U)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ~ halts2(U,Y) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(U,Y)
                & outputs(U,bad) ) ) ) )).

fof(prove_this,conjecture,
    ( ~ ( ? [X1] :
            ( algorithm(X1)
            & ! [Y1] :
                ( program(Y1)
               => ! [Z1] : decides(X1,Y1,Z1) ) ) ) )).

%----------------------------------------------------------------------------
```

Figure 3. Input script (adapted from file COM003+1.p of Sutcliffe and Suttner 2021) for the VAMPIRE proof of "The halting problem is undecidable on a Turing machine." The script uses Prolog-style (Ed-Dbali, Deransart, and Cervoni 1996) variables (for example, U, V, W, X, X1, Y, Y1, Z, Z1). For further details of the script syntax, see Sutcliffe and Suttner 2021.

Proof of the Halting Theorem (see Appendix 1 for details)

On the platforms described above, the execution wall-clock time of the script shown in Figure 3 was less than 1 millisecond. The resulting (~400-line) proof is shown in Appendix 1. The proof is by refutation. Although the proof may seem formidable, it has a relatively simple high-level structure, a description of which is at the end of Appendix 1.

Proofs of the independence and consistency of the axioms in Figure 3

In order to prove that the axioms in Figure 3 are independent, it suffices to show that the negation of p*i*, $i = 1, 2, 3, 4$, respectively, conjoined with all p*j*, $j = 1, 2, 3, 4$, $j \neq i$, has a model (Chang and Keisler 2012, Exercise 1.2.18). Appendices 2 – 5 contain VAMPIRE proofs of the independence of axioms p1 – p4 of Figure 3.

In order to prove that the axioms in Figure 3 are consistent, it suffices to show that axioms p1 – p4 of Figure 3 have a model (Tarski 1953, p. 12; Chang and Keisler 2012, Table 1.3.1).[16] Appendix 6 contains a VAMPIRE proof of the consistency of p1 – p4.

3.0 Discussion and conclusions

The VAMPIRE ADF and a VAMPIRE-compatible formulation (file COM003-1.p of Sutcliffe and Suttner 2021) of Burkholder's 1987 formalization of the Halting Problem rigorously prove that

a. p1 – p4 are independent.
b. p1 – p4 are consistent.
c. the Halting Problem is undecidable on a Turing machine.

[16] The TPTP Library (Sutcliffe and Suttner 2021) documentation for file COM003+1.p (which contains p1 – p4, plus the statement of the Halting Problem Theorem

```
fof(prove_this,conjecture,
    ( ~ ( ? [X1] :
            ( algorithm(X1)
            & ! [Y1] :
                ( program(Y1)
               => ! [Z1] : decides(X1,Y1,Z1) ) ) ) )).
```

states that file COM003+1.p is not satisfiable. In the context of the TPTP Library conventions, this means that none of ADFs the TPTP Library curators used to formally test COM003+1.p produced a finite model of that file. It does not mean that there is no possible finite model of COM003+1.p. If we remove the Halting Problem Theorem from COM003+1.p (leaving just p1 – p4), VAMPIRE produces the finite model shown in Appendix 6. This implies that p1 – p4 are consistent.

as shown in Appendices 1-6. These proofs, it's worth noting, have a somewhat paradoxical bent. They use programs (the software identified in Section 2.0), which we *know* halt (under at least one input) on a Turing machine, to show that *there is no effective method for determining* whether all programs that can run on a Turing machine halt.

The proofs of (a) – (c) shown in Appendices 1-6 may be novel: we are not aware that they have been published elsewhere.

The Halting Problem is evidently a fundamental limit on whether all software that could run on Turing machine can be fully verified by an effective procedure (Clarke, Henzinger, and Veith 2018, p. 2). The results of Segment TBD (see also Symons and Horner 2017) imply, furthermore, that we cannot even determine by *testing*, in all cases of interest, whether a software system contains a segment whose halting behavior cannot be determined by an effective method. This result, together with (HP), implies that we cannot, in all cases of interest, determine by testing whether a software system has the Halting Problem.

REFERENCES

Appel K and Haken W. (1989). *Every Planar Map is Four Colorable.* American Mathematical Society. (The first version of the proof was published in 1976).

Aristotle. (~350 BCE). *Prior Analytics.* Trans. by A. J. Jenkinson. In R. McKeon, ed. *The Basic Works of Aristotle.* Random House, 1941.

Boehm BW, Abts C, Brown AW, Chulani S, Clark BK, Horowitz E, Madachy R, Reifer, D, and Steece B. (2000). *Software Cost Estimation with COCOMO II*. Prentice-Hall.

Boolos GS, Burgess JP, and Jeffrey RC. (2007). *Computability and Logic.* Fifth Edition. Cambridge.

Bruschi M. (1991). The Halting Problem. *Association for Automated Reasoning Newsletter* 17, 7-12. http://aarinc.org/Newsletters/017-1991-03.pdf. Accessed 11 September 2021.

Chang CC and Keisler HJ. (2012). *Model Theory*. Third Edition. Dover.

Church A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 345–363.

Church A. (1956). *Introduction to Mathematical Logic*. Vol I. Princeton.

Cygwin authors. (2021). *Cygwin*. https://www.cygwin.com/. Accessed 26 July 2021.

Burkholder L. (1987). A 76th Automated Theorem Proving Problem. *Association for Automated Reasoning Newsletter* 8, 6-7. http://aarinc.org/Newsletters/008-1997-04.pdf. (Note that the rightmost segment of this address (unintentionally) suggests the year of this publication is 1997, not 1987.) Accessed 9 September 2021.

Ed-Dbali A, Deransart P, and Cervoni L. (1996). *Prolog: the standard: reference manual.* Springer.

Hales TC and Ferguson SP. (2011). *The Kepler Conjecture*. Springer. (The first version of this proof was published in 2005.)

Hempel CG. (1948). Studies in the Logic of Explanation. In C. G. Hempel. *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science.* Free Press, 1965. pp. 245-295.

Hilbert D and Ackermann W. (1928). *Grundzüge der theoretischen Logik* (*Principles of Mathematical Logic*). Springer.

Hunter G. (1971). *Metalogic: An Introduction to the Metatheory of Standard First-Order Logic*. University of California Press.

Kovács L. (2021). *VAMPIRE*. https://vprover.github.io/download.html. Accessed 26 July 2021.

Kovács L and Voronkov A. (2013). First-Order Theorem Proving and Vampire. *Proceedings of the 25th International Conference on Computer Aided Verification.* 13-19 July 2013, Saint Petersburg, Russia.

Leibniz GWF. (1685). The Art of Discovery. In P. P. Wiener, ed. *Leibniz: Selections*. Scribners, 1951.

Moore C and Mertens S. (2011). *The Nature of Computation*. Oxford University Press.

Ord T. (2006). The many forms of hypercomputation. *Applied mathematics and computation* 178.1, 143–153.

Pelletier FJ. (1986). Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning* 2, 191-216.

Quaife A. (1992). *Automated Development of Fundamental Mathematical Theories.* Kluwer.

Red Hat, Inc. (2021). Fedora and Red Hat Enterprise Linux. https://docs.fedoraproject.org/en-US/quick-docs/fedora-and-red-hat-enterprise-linux/. Accessed 13 October 2021.

Salmon WC. (1967). *Foundations of Scientific Inference.* University of Pittsburgh Press.

Strachey C. (1965). An impossible program. *The Computer Journal* 7, 313–313.

Sutcliffe G. (2008). The SZS Ontologies for Automated Reasoning Software. In Rudnicki P and Sutcliffe G, eds. *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants,* and *The 7th International Workshop on the Implementation of Logics (Doha, Qattar), CEUR Workshop Proceedings* 418, 38-49.

Sutcliffe G and Suttner C. (2021). *The TPTP Library*. http://www.tptp.org/. Accessed 26 July 2021.

Sutcliffe G, Zimmer J, and Schulz S. (2003). Communication Formalisms for Automated Theorem Proving Tools. In Sorge V, Colton S, Fisher M, and Gow J, eds. *Proceedings of the*

Workshop on Agents and Automated Reasoning, 18th International Joint Conference on Artificial Intelligence, (Acapulco, Mexico), 52-57.

Sutcliffe G, Zimmer J, Schulz S. (2004). TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In Zhang and Sorge V, eds. *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, Frontiers in Artificial Intelligence and Applications* 112, 201-215.

Symons JF and Horner JK. (2017). Software Error as a Limit to Inquiry for Finite Agents: Challenges for the Post-human Scientist. In Powers, T. (ed.) *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics.* Springer.

Tarski A. (1953). A general method in proofs of undecidability. In Tarski, A., Mostowski A., & Robinson R. M. *Undecidable Theories*. Dover reprint, pp. 1-35.

Turing A. (1936). On Computable Numbers, with an Application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society* 42, 230–65.

Watson AH, and Thomas J. McCabe TJ. (1996). *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* NIST Special Publication 500-235.

Wos L. (1988). *Automated Reasoning: 33 Basic Research Problems.* Prentice-Hall.

APPENDIX 1. VAMPIRE proof of "The halting problem cannot be solved on a Turing machine". See Sutcliffe 2008; Sutcliffe, Zimmer, and Schulz 2003, 2004; Sutcliffe and Suttner 2021 for syntax and ontology details.

Command line:

```
$ ../bin/vampire_rel.exe COM003+1.p > COM003+1.p.txt
```

Proof. The proof is by refutation. Such a proof first asserts the axioms/assumptions of the theorem of interest, and asserts the negation of the consequent that theorem, then proceeds to draw a contradiction.

```
% Refutation found. Thanks to Tanya!

% SZS status Theorem for COM003+1

% SZS output start Proof for COM003+1

1. ? [X0] : (! [X1] : (program(X1) => ! [X2] : decides(X0,X1,X2)) &
algorithm(X0)) => ? [X3] : (! [X1] : (program(X1) => ! [X2] :
decides(X3,X1,X2)) & program(X3)) [input]

2. ! [X3] : ((! [X1] : (program(X1) => ! [X2] : decides(X3,X1,X2)) &
program(X3)) => ! [X1,X2] : (((~halts2(X1,X2) & program(X1)) =>
(outputs(X3,bad) & halts3(X3,X1,X2))) & ((halts2(X1,X2) & program(X1)) =>
(outputs(X3,good) & halts3(X3,X1,X2))))) [input]

3. ? [X3] : (! [X1] : (((~halts2(X1,X1) & program(X1)) => (outputs(X3,bad) &
halts3(X3,X1,X1))) & ((halts2(X1,X1) & program(X1)) => (outputs(X3,good) &
halts3(X3,X1,X1)))) & program(X3)) => ? [X4] : (! [X1] : (((~halts2(X1,X1) &
program(X1)) => (outputs(X4,bad) & halts2(X4,X1))) & ((halts2(X1,X1) &
program(X1)) => (outputs(X4,good) & halts2(X4,X1)))) & program(X4)) [input]

4. ? [X4] : (! [X1] : (((~halts2(X1,X1) & program(X1)) => (outputs(X4,bad) &
halts2(X4,X1))) & ((halts2(X1,X1) & program(X1)) => (outputs(X4,good) &
halts2(X4,X1)))) & program(X4)) => ? [X5] : (! [X1] : (((~halts2(X1,X1) &
program(X1)) => (outputs(X5,bad) & halts2(X5,X1))) & ((halts2(X1,X1) &
program(X1)) => ~halts2(X5,X1))) & program(X5)) [input]

5. ~? [X6] : (! [X7] : (program(X7) => ! [X8] : decides(X6,X7,X8)) &
algorithm(X6)) [input]
```

```
6. ~~? [X6] : (! [X7] : (program(X7) => ! [X8] : decides(X6,X7,X8)) &
algorithm(X6)) [negated conjecture 5]

7. ? [X0] : (! [X1] : (program(X1) => ! [X2] : decides(X0,X1,X2)) &
algorithm(X0)) => ? [X3] : (! [X4] : (program(X4) => ! [X5] :
decides(X3,X4,X5)) & program(X3)) [rectify 1]

8. ! [X0] : ((! [X1] : (program(X1) => ! [X2] : decides(X0,X1,X2)) &
program(X0)) => ! [X3,X4] : (((~halts2(X3,X4) & program(X3)) =>
(outputs(X0,bad) & halts3(X0,X3,X4))) & ((halts2(X3,X4) & program(X3)) =>
(outputs(X0,good) & halts3(X0,X3,X4))))) [rectify 2]

9. ? [X0] : (! [X1] : (((~halts2(X1,X1) & program(X1)) => (outputs(X0,bad) &
halts3(X0,X1,X1))) & ((halts2(X1,X1) & program(X1)) => (outputs(X0,good) &
halts3(X0,X1,X1)))) & program(X0)) => ? [X2] : (! [X3] : (((~halts2(X3,X3) &
program(X3)) => (outputs(X2,bad) & halts2(X2,X3))) & ((halts2(X3,X3) &
program(X3)) => (outputs(X2,good) & halts2(X2,X3)))) & program(X2)) [rectify
3]

10. ? [X0] : (! [X1] : (((~halts2(X1,X1) & program(X1)) => (outputs(X0,bad) &
halts2(X0,X1))) & ((halts2(X1,X1) & program(X1)) => (outputs(X0,good) &
halts2(X0,X1)))) & program(X0)) => ? [X2] : (! [X3] : (((~halts2(X3,X3) &
program(X3)) => (outputs(X2,bad) & halts2(X2,X3))) & ((halts2(X3,X3) &
program(X3)) => ~halts2(X2,X3))) & program(X2)) [rectify 4]

11. ~~? [X0] : (! [X1] : (program(X1) => ! [X2] : decides(X0,X1,X2)) &
algorithm(X0)) [rectify 6]

12. ? [X0] : (! [X1] : (program(X1) => ! [X2] : decides(X0,X1,X2)) &
algorithm(X0)) [flattening 11]

13. ? [X3] : (! [X4] : (! [X5] : decides(X3,X4,X5) | ~program(X4)) &
program(X3)) | ! [X0] : (? [X1] : (? [X2] : ~decides(X0,X1,X2) & program(X1))
| ~algorithm(X0)) [ennf transformation 7]

14. ! [X0] : (! [X3,X4] : (((outputs(X0,bad) & halts3(X0,X3,X4)) |
(halts2(X3,X4) | ~program(X3))) & ((outputs(X0,good) & halts3(X0,X3,X4)) |
(~halts2(X3,X4) | ~program(X3)))) | (? [X1] : (? [X2] : ~decides(X0,X1,X2) &
program(X1)) | ~program(X0))) [ennf transformation 8]

15. ! [X0] : (! [X3,X4] : (((outputs(X0,bad) & halts3(X0,X3,X4)) |
halts2(X3,X4) | ~program(X3)) & ((outputs(X0,good) & halts3(X0,X3,X4)) |
~halts2(X3,X4) | ~program(X3))) | ? [X1] : (? [X2] : ~decides(X0,X1,X2) &
program(X1)) | ~program(X0)) [flattening 14]

16. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | (halts2(X3,X3) |
~program(X3))) & ((outputs(X2,good) & halts2(X2,X3)) | (~halts2(X3,X3) |
~program(X3)))) & program(X2)) | ! [X0] : (? [X1] : (((~outputs(X0,bad) |
~halts3(X0,X1,X1)) & (~halts2(X1,X1) & program(X1))) | ((~outputs(X0,good) |
~halts3(X0,X1,X1)) & (halts2(X1,X1) & program(X1)))) | ~program(X0)) [ennf
transformation 9]

17. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | halts2(X3,X3) |
~program(X3)) & ((outputs(X2,good) & halts2(X2,X3)) | ~halts2(X3,X3) |
~program(X3))) & program(X2)) | ! [X0] : (? [X1] : (((~outputs(X0,bad) |
~halts3(X0,X1,X1)) & ~halts2(X1,X1) & program(X1)) | ((~outputs(X0,good) |
```

~halts3(X0,X1,X1)) & halts2(X1,X1) & program(X1))) | ~program(X0))
[flattening 16]

18. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | (halts2(X3,X3) |
~program(X3))) & (~halts2(X2,X3) | (~halts2(X3,X3) | ~program(X3)))) &
program(X2)) | ! [X0] : (? [X1] : (((~outputs(X0,bad) | ~halts2(X0,X1)) &
(~halts2(X1,X1) & program(X1))) | ((~outputs(X0,good) | ~halts2(X0,X1)) &
(halts2(X1,X1) & program(X1)))) | ~program(X0)) [ennf transformation 10]

19. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | halts2(X3,X3) |
~program(X3)) & (~halts2(X2,X3) | ~halts2(X3,X3) | ~program(X3))) &
program(X2)) | ! [X0] : (? [X1] : (((~outputs(X0,bad) | ~halts2(X0,X1)) &
~halts2(X1,X1) & program(X1)) | ((~outputs(X0,good) | ~halts2(X0,X1)) &
halts2(X1,X1) & program(X1))) | ~program(X0)) [flattening 18]

20. ? [X0] : (! [X1] : (! [X2] : decides(X0,X1,X2) | ~program(X1)) &
algorithm(X0)) [ennf transformation 12]

21. ! [X0,X1] : (((~outputs(X0,good) | ~halts3(X0,X1,X1)) & halts2(X1,X1) &
program(X1)) | ~sP0(X0,X1)) [predicate definition introduction]

22. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | halts2(X3,X3) |
~program(X3)) & ((outputs(X2,good) & halts2(X2,X3)) | ~halts2(X3,X3) |
~program(X3))) & program(X2)) | ~sP1 [predicate definition introduction]

23. sP1 | ! [X0] : (? [X1] : (((~outputs(X0,bad) | ~halts3(X0,X1,X1)) &
~halts2(X1,X1) & program(X1)) | sP0(X0,X1)) | ~program(X0)) [definition
folding 17,22,21]

24. ! [X0,X1] : (((~outputs(X0,good) | ~halts2(X0,X1)) & halts2(X1,X1) &
program(X1)) | ~sP2(X0,X1)) [predicate definition introduction]

25. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | halts2(X3,X3) |
~program(X3)) & (~halts2(X2,X3) | ~halts2(X3,X3) | ~program(X3))) &
program(X2)) | ~sP3 [predicate definition introduction]

26. sP3 | ! [X0] : (? [X1] : (((~outputs(X0,bad) | ~halts2(X0,X1)) &
~halts2(X1,X1) & program(X1)) | sP2(X0,X1)) | ~program(X0)) [definition
folding 19,25,24]

27. ? [X0] : (! [X1] : (! [X2] : decides(X0,X1,X2) | ~program(X1)) &
program(X0)) | ! [X3] : (? [X4] : (? [X5] : ~decides(X3,X4,X5) & program(X4))
| ~algorithm(X3)) [rectify 13]

28. ? [X0] : (! [X1] : (! [X2] : decides(X0,X1,X2) | ~program(X1)) &
program(X0)) => (! [X1] : (! [X2] : decides(sK4,X1,X2) | ~program(X1)) &
program(sK4)) [choice axiom]

29. ! [X3] : (? [X4] : (? [X5] : ~decides(X3,X4,X5) & program(X4)) => (? [X5]
: ~decides(X3,sK5(X3),X5) & program(sK5(X3)))) [choice axiom]

30. ! [X3] : (? [X5] : ~decides(X3,sK5(X3),X5) =>
~decides(X3,sK5(X3),sK6(X3))) [choice axiom]

31. (! [X1] : (! [X2] : decides(sK4,X1,X2) | ~program(X1)) & program(sK4)) |
! [X3] : ((~decides(X3,sK5(X3),sK6(X3)) & program(sK5(X3))) | ~algorithm(X3))
[skolemisation 27,30,29,28]

32. ! [X0] : (! [X1,X2] : (((outputs(X0,bad) & halts3(X0,X1,X2)) | halts2(X1,X2) | ~program(X1)) & ((outputs(X0,good) & halts3(X0,X1,X2)) | ~halts2(X1,X2) | ~program(X1))) | ? [X3] : (? [X4] : ~decides(X0,X3,X4) & program(X3)) | ~program(X0)) [rectify 15]

33. ! [X0] : (? [X3] : (? [X4] : ~decides(X0,X3,X4) & program(X3)) => (? [X4] : ~decides(X0,sK7(X0),X4) & program(sK7(X0)))) [choice axiom]

34. ! [X0] : (? [X4] : ~decides(X0,sK7(X0),X4) => ~decides(X0,sK7(X0),sK8(X0))) [choice axiom]

35. ! [X0] : (! [X1,X2] : (((outputs(X0,bad) & halts3(X0,X1,X2)) | halts2(X1,X2) | ~program(X1)) & ((outputs(X0,good) & halts3(X0,X1,X2)) | ~halts2(X1,X2) | ~program(X1))) | (~decides(X0,sK7(X0),sK8(X0)) & program(sK7(X0))) | ~program(X0)) [skolemisation 32,34,33]

36. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | halts2(X3,X3) | ~program(X3)) & ((outputs(X2,good) & halts2(X2,X3)) | ~halts2(X3,X3) | ~program(X3))) & program(X2)) | ~sP1 [nnf transformation 22]

37. ? [X0] : (! [X1] : (((outputs(X0,bad) & halts2(X0,X1)) | halts2(X1,X1) | ~program(X1)) & ((outputs(X0,good) & halts2(X0,X1)) | ~halts2(X1,X1) | ~program(X1))) & program(X0)) | ~sP1 [rectify 36]

38. ? [X0] : (! [X1] : (((outputs(X0,bad) & halts2(X0,X1)) | halts2(X1,X1) | ~program(X1)) & ((outputs(X0,good) & halts2(X0,X1)) | ~halts2(X1,X1) | ~program(X1))) & program(X0)) => (! [X1] : (((outputs(sK9,bad) & halts2(sK9,X1)) | halts2(X1,X1) | ~program(X1)) & ((outputs(sK9,good) & halts2(sK9,X1)) | ~halts2(X1,X1) | ~program(X1))) & program(sK9)) [choice axiom]

39. (! [X1] : (((outputs(sK9,bad) & halts2(sK9,X1)) | halts2(X1,X1) | ~program(X1)) & ((outputs(sK9,good) & halts2(sK9,X1)) | ~halts2(X1,X1) | ~program(X1))) & program(sK9)) | ~sP1 [skolemisation 37,38]

40. ! [X0,X1] : (((~outputs(X0,good) | ~halts3(X0,X1,X1)) & halts2(X1,X1) & program(X1)) | ~sP0(X0,X1)) [nnf transformation 21]

41. ! [X0] : (? [X1] : (((~outputs(X0,bad) | ~halts3(X0,X1,X1)) & ~halts2(X1,X1) & program(X1)) | sP0(X0,X1)) => (((~outputs(X0,bad) | ~halts3(X0,sK10(X0),sK10(X0))) & ~halts2(sK10(X0),sK10(X0)) & program(sK10(X0))) | sP0(X0,sK10(X0)))) [choice axiom]

42. sP1 | ! [X0] : ((((~outputs(X0,bad) | ~halts3(X0,sK10(X0),sK10(X0))) & ~halts2(sK10(X0),sK10(X0)) & program(sK10(X0))) | sP0(X0,sK10(X0))) | ~program(X0)) [skolemisation 23,41]

43. ? [X2] : (! [X3] : (((outputs(X2,bad) & halts2(X2,X3)) | halts2(X3,X3) | ~program(X3)) & (~halts2(X2,X3) | ~halts2(X3,X3) | ~program(X3))) & program(X2)) | ~sP3 [nnf transformation 25]

44. ? [X0] : (! [X1] : (((outputs(X0,bad) & halts2(X0,X1)) | halts2(X1,X1) | ~program(X1)) & (~halts2(X0,X1) | ~halts2(X1,X1) | ~program(X1))) & program(X0)) | ~sP3 [rectify 43]

45. ? [X0] : (! [X1] : (((outputs(X0,bad) & halts2(X0,X1)) | halts2(X1,X1) | ~program(X1)) & (~halts2(X0,X1) | ~halts2(X1,X1) | ~program(X1))) &

program(X0)) => (! [X1] : (((outputs(sK11,bad) & halts2(sK11,X1)) | halts2(X1,X1) | ~program(X1)) & (~halts2(sK11,X1) | ~halts2(X1,X1) | ~program(X1))) & program(sK11)) [choice axiom]

46. (! [X1] : (((outputs(sK11,bad) & halts2(sK11,X1)) | halts2(X1,X1) | ~program(X1)) & (~halts2(sK11,X1) | ~halts2(X1,X1) | ~program(X1))) & program(sK11)) | ~sP3 [skolemisation 44,45]

47. ! [X0,X1] : (((~outputs(X0,good) | ~halts2(X0,X1)) & halts2(X1,X1) & program(X1)) | ~sP2(X0,X1)) [nnf transformation 24]

48. ! [X0] : (? [X1] : (((~outputs(X0,bad) | ~halts2(X0,X1)) & ~halts2(X1,X1) & program(X1)) | sP2(X0,X1)) => (((~outputs(X0,bad) | ~halts2(X0,sK12(X0))) & ~halts2(sK12(X0),sK12(X0)) & program(sK12(X0))) | sP2(X0,sK12(X0)))) [choice axiom]

49. sP3 | ! [X0] : ((((~outputs(X0,bad) | ~halts2(X0,sK12(X0))) & ~halts2(sK12(X0),sK12(X0)) & program(sK12(X0))) | sP2(X0,sK12(X0))) | ~program(X0)) [skolemisation 26,48]

50. ? [X0] : (! [X1] : (! [X2] : decides(X0,X1,X2) | ~program(X1)) & algorithm(X0)) => (! [X1] : (! [X2] : decides(sK13,X1,X2) | ~program(X1)) & algorithm(sK13)) [choice axiom]

51. ! [X1] : (! [X2] : decides(sK13,X1,X2) | ~program(X1)) & algorithm(sK13) [skolemisation 20,50]

52. program(sK4) | program(sK5(X3)) | ~algorithm(X3) [cnf transformation 31]

53. program(sK4) | ~decides(X3,sK5(X3),sK6(X3)) | ~algorithm(X3) [cnf transformation 31]

54. decides(sK4,X1,X2) | ~program(X1) | program(sK5(X3)) | ~algorithm(X3) [cnf transformation 31]

55. decides(sK4,X1,X2) | ~program(X1) | ~decides(X3,sK5(X3),sK6(X3)) | ~algorithm(X3) [cnf transformation 31]

56. halts3(X0,X1,X2) | ~halts2(X1,X2) | ~program(X1) | program(sK7(X0)) | ~program(X0) [cnf transformation 35]

57. halts3(X0,X1,X2) | ~halts2(X1,X2) | ~program(X1) | ~decides(X0,sK7(X0),sK8(X0)) | ~program(X0) [cnf transformation 35]

58. outputs(X0,good) | ~halts2(X1,X2) | ~program(X1) | program(sK7(X0)) | ~program(X0) [cnf transformation 35]

59. outputs(X0,good) | ~halts2(X1,X2) | ~program(X1) | ~decides(X0,sK7(X0),sK8(X0)) | ~program(X0) [cnf transformation 35]

60. halts3(X0,X1,X2) | halts2(X1,X2) | ~program(X1) | program(sK7(X0)) | ~program(X0) [cnf transformation 35]

61. ~decides(X0,sK7(X0),sK8(X0)) | halts2(X1,X2) | ~program(X1) | halts3(X0,X1,X2) | ~program(X0) [cnf transformation 35]

62. outputs(X0,bad) | halts2(X1,X2) | ~program(X1) | program(sK7(X0)) | ~program(X0) [cnf transformation 35]

63. outputs(X0,bad) | halts2(X1,X2) | ~program(X1) | ~decides(X0,sK7(X0),sK8(X0)) | ~program(X0) [cnf transformation 35]

64. program(sK9) | ~sP1 [cnf transformation 39]

65. halts2(sK9,X1) | ~halts2(X1,X1) | ~program(X1) | ~sP1 [cnf transformation 39]

66. outputs(sK9,good) | ~halts2(X1,X1) | ~program(X1) | ~sP1 [cnf transformation 39]

67. halts2(sK9,X1) | halts2(X1,X1) | ~program(X1) | ~sP1 [cnf transformation 39]

68. outputs(sK9,bad) | halts2(X1,X1) | ~program(X1) | ~sP1 [cnf transformation 39]

69. ~sP0(X0,X1) | program(X1) [cnf transformation 40]

70. ~sP0(X0,X1) | halts2(X1,X1) [cnf transformation 40]

71. ~halts3(X0,X1,X1) | ~outputs(X0,good) | ~sP0(X0,X1) [cnf transformation 40]

72. sP1 | program(sK10(X0)) | sP0(X0,sK10(X0)) | ~program(X0) [cnf transformation 42]

74. sP1 | ~outputs(X0,bad) | ~halts3(X0,sK10(X0),sK10(X0)) | sP0(X0,sK10(X0)) | ~program(X0) [cnf transformation 42]

75. program(sK11) | ~sP3 [cnf transformation 46]

76. ~halts2(sK11,X1) | ~halts2(X1,X1) | ~program(X1) | ~sP3 [cnf transformation 46]

77. halts2(sK11,X1) | halts2(X1,X1) | ~program(X1) | ~sP3 [cnf transformation 46]

79. ~sP2(X0,X1) | program(X1) [cnf transformation 47]

81. ~sP2(X0,X1) | ~halts2(X0,X1) | ~outputs(X0,good) [cnf transformation 47]

82. sP3 | program(sK12(X0)) | sP2(X0,sK12(X0)) | ~program(X0) [cnf transformation 49]

83. sP3 | ~halts2(sK12(X0),sK12(X0)) | sP2(X0,sK12(X0)) | ~program(X0) [cnf transformation 49]

84. sP3 | ~outputs(X0,bad) | ~halts2(X0,sK12(X0)) | sP2(X0,sK12(X0)) | ~program(X0) [cnf transformation 49]

85. algorithm(sK13) [cnf transformation 51]

86. decides(sK13,X1,X2) | ~program(X1) [cnf transformation 51]

88. 1 <=> ! [X3] : (~decides(X3,sK5(X3),sK6(X3)) | ~algorithm(X3)) [avatar definition]

89. ~decides(X3,sK5(X3),sK6(X3)) | ~algorithm(X3) <- (1) [avatar component clause 88]

91. 2 <=> ! [X1,X2] : (decides(sK4,X1,X2) | ~program(X1)) [avatar definition]

92. decides(sK4,X1,X2) | ~program(X1) <- (2) [avatar component clause 91]

93. 1 | 2 [avatar split clause 55,91,88]

95. 3 <=> ! [X3] : (program(sK5(X3)) | ~algorithm(X3)) [avatar definition]

96. program(sK5(X3)) | ~algorithm(X3) <- (3) [avatar component clause 95]

97. 3 | 2 [avatar split clause 54,91,95]

99. 4 <=> program(sK4) [avatar definition]

101. program(sK4) <- (4) [avatar component clause 99]

102. 1 | 4 [avatar split clause 53,99,88]

103. 3 | 4 [avatar split clause 52,99,95]

105. 5 <=> ! [X1,X2] : (halts2(X1,X2) | ~program(X1)) [avatar definition]

106. halts2(X1,X2) | ~program(X1) <- (5) [avatar component clause 105]

108. 6 <=> ! [X0] : (outputs(X0,bad) | ~program(X0) |
~decides(X0,sK7(X0),sK8(X0))) [avatar definition]

109. ~decides(X0,sK7(X0),sK8(X0)) | ~program(X0) | outputs(X0,bad) <- (6)
[avatar component clause 108]

110. 5 | 6 [avatar split clause 63,108,105]

112. 7 <=> ! [X0] : (outputs(X0,bad) | ~program(X0) | program(sK7(X0)))
[avatar definition]

113. outputs(X0,bad) | ~program(X0) | program(sK7(X0)) <- (7) [avatar
component clause 112]

114. 5 | 7 [avatar split clause 62,112,105]

116. 8 <=> ! [X1,X2] : (~halts2(X1,X2) | ~program(X1)) [avatar definition]

117. ~halts2(X1,X2) | ~program(X1) <- (8) [avatar component clause 116]

119. 9 <=> ! [X0] : (outputs(X0,good) | ~program(X0) |
~decides(X0,sK7(X0),sK8(X0))) [avatar definition]

120. ~decides(X0,sK7(X0),sK8(X0)) | ~program(X0) | outputs(X0,good) <- (9)
[avatar component clause 119]

121. 8 | 9 [avatar split clause 59,119,116]

123. 10 <=> ! [X0] : (outputs(X0,good) | ~program(X0) | program(sK7(X0)))
[avatar definition]

124. outputs(X0,good) | ~program(X0) | program(sK7(X0)) <- (10) [avatar
component clause 123]

125. 8 | 10 [avatar split clause 58,123,116]

126. ~decides(X0,sK7(X0),sK8(X0)) | ~program(X1) | halts3(X0,X1,X2) |
~program(X0) [subsumption resolution 57,61]

127. halts3(X0,X1,X2) | ~program(X1) | program(sK7(X0)) | ~program(X0) [subsumption resolution 56,60]

129. 11 <=> sP1 [avatar definition]

133. 12 <=> ! [X1] : (halts2(X1,X1) | ~program(X1)) [avatar definition]

134. halts2(X1,X1) | ~program(X1) <- (12) [avatar component clause 133]

136. 13 <=> outputs(sK9,bad) [avatar definition]

139. ~11 | 12 | 13 [avatar split clause 68,136,133,129]

141. 14 <=> ! [X1] : (halts2(sK9,X1) | ~program(X1) | halts2(X1,X1)) [avatar definition]

142. halts2(sK9,X1) | ~program(X1) | halts2(X1,X1) <- (14) [avatar component clause 141]

143. ~11 | 14 [avatar split clause 67,141,129]

145. 15 <=> ! [X1] : (~halts2(X1,X1) | ~program(X1)) [avatar definition]

146. ~halts2(X1,X1) | ~program(X1) <- (15) [avatar component clause 145]

148. 16 <=> outputs(sK9,good) [avatar definition]

150. outputs(sK9,good) <- (16) [avatar component clause 148]

151. ~11 | 15 | 16 [avatar split clause 66,148,145,129]

153. 17 <=> ! [X1] : (halts2(sK9,X1) | ~program(X1) | ~halts2(X1,X1)) [avatar definition]

154. ~halts2(X1,X1) | ~program(X1) | halts2(sK9,X1) <- (17) [avatar component clause 153]

155. ~11 | 17 [avatar split clause 65,153,129]

157. 18 <=> program(sK9) [avatar definition]

159. program(sK9) <- (18) [avatar component clause 157]

160. ~11 | 18 [avatar split clause 64,157,129]

162. 19 <=> ! [X0] : (~outputs(X0,bad) | ~program(X0) | sP0(X0,sK10(X0)) | ~halts3(X0,sK10(X0),sK10(X0))) [avatar definition]

163. ~halts3(X0,sK10(X0),sK10(X0)) | ~program(X0) | sP0(X0,sK10(X0)) | ~outputs(X0,bad) <- (19) [avatar component clause 162]

164. 19 | 11 [avatar split clause 74,129,162]

169. sP1 | program(sK10(X0)) | ~program(X0) [subsumption resolution 72,69]

171. 21 <=> ! [X0] : (program(sK10(X0)) | ~program(X0)) [avatar definition]

172. program(sK10(X0)) | ~program(X0) <- (21) [avatar component clause 171]

173. 21 | 11 [avatar split clause 169,129,171]

175. 22 <=> sP3 [avatar definition]

184. 24 <=> ! [X1] : (halts2(sK11,X1) | ~program(X1) | halts2(X1,X1)) [avatar definition]

185. halts2(sK11,X1) | halts2(X1,X1) | ~program(X1) <- (24) [avatar component clause 184]

186. ~22 | 24 [avatar split clause 77,184,175]

188. 25 <=> ! [X1] : (~halts2(sK11,X1) | ~program(X1) | ~halts2(X1,X1)) [avatar definition]

189. ~halts2(sK11,X1) | ~program(X1) | ~halts2(X1,X1) <- (25) [avatar component clause 188]

190. ~22 | 25 [avatar split clause 76,188,175]

192. 26 <=> program(sK11) [avatar definition]

194. program(sK11) <- (26) [avatar component clause 192]

195. ~22 | 26 [avatar split clause 75,192,175]

197. 27 <=> ! [X0] : (~outputs(X0,bad) | ~program(X0) | sP2(X0,sK12(X0)) | ~halts2(X0,sK12(X0))) [avatar definition]

198. ~halts2(X0,sK12(X0)) | ~program(X0) | sP2(X0,sK12(X0)) | ~outputs(X0,bad) <- (27) [avatar component clause 197]

199. 27 | 22 [avatar split clause 84,175,197]

201. 28 <=> ! [X0] : (~halts2(sK12(X0),sK12(X0)) | ~program(X0) | sP2(X0,sK12(X0))) [avatar definition]

202. ~halts2(sK12(X0),sK12(X0)) | ~program(X0) | sP2(X0,sK12(X0)) <- (28) [avatar component clause 201]

203. 28 | 22 [avatar split clause 83,175,201]

204. sP3 | program(sK12(X0)) | ~program(X0) [subsumption resolution 82,79]

206. 29 <=> ! [X0] : (program(sK12(X0)) | ~program(X0)) [avatar definition]

207. program(sK12(X0)) | ~program(X0) <- (29) [avatar component clause 206]

208. 29 | 22 [avatar split clause 204,175,206]

209. halts2(sK9,X1) | ~program(X1) <- (14, 17) [subsumption resolution 142,154]

213. ~program(X1) <- (5, 8) [subsumption resolution 117,106]

214. $false <- (4, 5, 8) [resolution 213,101]

216. $false <- (5, 8, 26) [resolution 213,194]

217. ~5 | ~8 | ~26 [avatar contradiction clause 216]

219. ~4 | ~5 | ~8 [avatar contradiction clause 214]

226. halts2(sK11,sK11) | ~program(sK11) <- (24) [factoring 185]

248. ~program(sK4) | outputs(sK4,good) | ~program(sK7(sK4)) <- (2, 9) [resolution 120,92]

261. 32 <=> program(sK7(sK4)) [avatar definition]

263. ~program(sK7(sK4)) <- (~32) [avatar component clause 261]

265. 33 <=> outputs(sK4,good) [avatar definition]

267. outputs(sK4,good) <- (33) [avatar component clause 265]

269. ~program(X0) | program(sK7(X1)) | ~program(X1) | ~outputs(X1,good) | ~sP0(X1,X0) [resolution 127,71]

270. program(sK7(X1)) | ~program(X1) | ~outputs(X1,good) | ~sP0(X1,X0) [subsumption resolution 269,69]

271. ~sP0(X1,X0) | ~program(X1) | program(sK7(X1)) <- (10) [subsumption resolution 270,124]

272. ~program(X0) | sP2(X0,sK12(X0)) | ~program(sK12(X0)) <- (5, 28) [resolution 202,106]

274. sP2(X0,sK12(X0)) | ~program(X0) <- (5, 28, 29) [subsumption resolution 272,207]

275. ~program(X0) | ~halts2(X0,sK12(X0)) | ~outputs(X0,good) <- (5, 28, 29) [resolution 274,81]

278. ~outputs(X0,good) | ~program(X0) <- (5, 28, 29) [subsumption resolution 275,106]

302. ~program(X2) | halts3(sK4,X2,X3) | ~program(sK4) | ~program(sK7(sK4)) <- (2) [resolution 126,92]

305. ~program(sK4) <- (5, 28, 29, 33) [resolution 267,278]

306. $false <- (4, 5, 28, 29, 33) [subsumption resolution 305,101]

307. ~4 | ~5 | ~28 | ~29 | ~33 [avatar contradiction clause 306]

310. 34 <=> ! [X3,X2] : (~program(X2) | halts3(sK4,X2,X3)) [avatar definition]

311. halts3(sK4,X2,X3) | ~program(X2) <- (34) [avatar component clause 310]

316. ~algorithm(sK13) | ~program(sK5(sK13)) <- (1) [resolution 89,86]

318. ~program(sK5(sK13)) <- (1) [subsumption resolution 316,85]

321. ~program(X1) | sP0(X1,sK10(X1)) | ~outputs(X1,bad) | ~program(sK10(X1)) | program(sK7(X1)) | ~program(X1) <- (19) [resolution 163,127]

322. ~program(X1) | sP0(X1,sK10(X1)) | ~outputs(X1,bad) | ~program(sK10(X1)) | program(sK7(X1)) <- (19) [duplicate literal removal 321]

324. ~algorithm(sK13) <- (1, 3) [resolution 318,96]

325. $false <- (1, 3) [subsumption resolution 324,85]

326. ~1 | ~3 [avatar contradiction clause 325]

336. ~program(X1) | ~outputs(X1,bad) | ~program(sK10(X1)) | program(sK7(X1)) <- (10, 19) [subsumption resolution 322,271]

337. ~program(X1) | ~outputs(X1,bad) | program(sK7(X1)) <- (10, 19, 21) [subsumption resolution 336,172]

346. 38 <=> halts2(sK11,sK11) [avatar definition]

348. halts2(sK11,sK11) <- (38) [avatar component clause 346]

349. ~26 | 38 | ~24 [avatar split clause 226,184,346,192]

350. ~program(sK4) | outputs(sK4,good) <- (2, 9, 10) [subsumption resolution 248,124]

351. 33 | ~4 | ~2 | ~9 | ~10 [avatar split clause 350,123,119,91,99,265]

352. ~program(X2) | halts3(sK4,X2,X3) | ~program(sK4) <- (2) [subsumption resolution 302,127]

353. ~4 | 34 | ~2 [avatar split clause 352,91,310,99]

354. program(sK7(X0)) | ~program(X0) <- (7, 10, 19, 21) [subsumption resolution 113,337]

355. ~program(sK4) <- (7, 10, 19, 21, ~32) [resolution 263,354]

356. $false <- (4, 7, 10, 19, 21, ~32) [subsumption resolution 355,101]

357. ~4 | ~7 | ~10 | ~19 | ~21 | 32 [avatar contradiction clause 356]

362. ~program(sK9) | ~program(sK9) <- (14, 15, 17) [resolution 146,209]

365. ~program(sK9) <- (14, 15, 17) [duplicate literal removal 362]

367. $false <- (14, 15, 17, 18) [subsumption resolution 365,159]

368. ~14 | ~15 | ~17 | ~18 [avatar contradiction clause 367]

370. ~program(sK4) | outputs(sK4,bad) | ~program(sK7(sK4)) <- (2, 6) [resolution 109,92]

371. ~program(sK9) | sP2(sK9,sK12(sK9)) | ~outputs(sK9,bad) | ~program(sK12(sK9)) <- (14, 17, 27) [resolution 198,209]

373. sP2(sK9,sK12(sK9)) | ~outputs(sK9,bad) | ~program(sK12(sK9)) <- (14, 17, 18, 27) [subsumption resolution 371,159]

376. 39 <=> program(sK12(sK9)) [avatar definition]

377. program(sK12(sK9)) <- (39) [avatar component clause 376]

378. ~program(sK12(sK9)) <- (~39) [avatar component clause 376]

380. 40 <=> sP2(sK9,sK12(sK9)) [avatar definition]

381. ~sP2(sK9,sK12(sK9)) <- (~40) [avatar component clause 380]

382. sP2(sK9,sK12(sK9)) <- (40) [avatar component clause 380]

384. ~program(sK9) <- (29, ~39) [resolution 378,207]

385. $false <- (18, 29, ~39) [subsumption resolution 384,159]

386. ~18 | ~29 | 39 [avatar contradiction clause 385]

389. ~halts2(sK9,sK12(sK9)) | ~outputs(sK9,good) <- (40) [resolution 382,81]

392. ~halts2(sK9,sK12(sK9)) <- (16, 40) [subsumption resolution 389,150]

393. ~program(sK12(sK9)) <- (14, 16, 17, 40) [resolution 392,209]

394. $false <- (14, 16, 17, 39, 40) [subsumption resolution 393,377]

395. ~14 | ~16 | ~17 | ~39 | ~40 [avatar contradiction clause 394]

396. sP2(sK9,sK12(sK9)) | ~outputs(sK9,bad) <- (14, 17, 18, 27, 39) [subsumption resolution 373,377]

397. ~13 | 40 | ~14 | ~17 | ~18 | ~27 | ~39 [avatar split clause 396,376,197,157,153,141,380,136]

401. ~program(sK12(X2)) | ~program(X2) | sP2(X2,sK12(X2)) <- (12, 28) [resolution 134,202]

403. sP2(X2,sK12(X2)) | ~program(X2) <- (12, 28, 29) [subsumption resolution 401,207]

404. ~program(X0) | ~outputs(sK4,good) | ~sP0(sK4,X0) <- (34) [resolution 311,71]

406. ~program(X0) | ~sP0(sK4,X0) <- (33, 34) [subsumption resolution 404,267]

407. ~sP0(sK4,X0) <- (33, 34) [subsumption resolution 406,69]

411. ~program(sK9) <- (12, 28, 29, ~40) [resolution 403,381]

412. $false <- (12, 18, 28, 29, ~40) [subsumption resolution 411,159]

413. ~12 | ~18 | ~28 | ~29 | 40 [avatar contradiction clause 412]

416. ~program(sK11) | ~halts2(sK11,sK11) <- (25, 38) [resolution 189,348]

420. ~halts2(sK11,sK11) <- (25, 26, 38) [subsumption resolution 416,194]

421. $false <- (25, 26, 38) [subsumption resolution 420,348]

422. ~25 | ~26 | ~38 [avatar contradiction clause 421]

423. outputs(sK4,bad) | ~program(sK7(sK4)) <- (2, 4, 6) [subsumption resolution 370,101]

425. 41 <=> outputs(sK4,bad) [avatar definition]

426. ~outputs(sK4,bad) <- (~41) [avatar component clause 425]

427. outputs(sK4,bad) <- (41) [avatar component clause 425]

428. ~32 | 41 | ~2 | ~4 | ~6 [avatar split clause 423,108,99,91,425,261]

433. ~program(sK4) | sP0(sK4,sK10(sK4)) | ~outputs(sK4,bad) | ~program(sK10(sK4)) <- (19, 34) [resolution 163,311]

442. sP0(sK4,sK10(sK4)) | ~outputs(sK4,bad) | ~program(sK10(sK4)) <- (4, 19, 34) [subsumption resolution 433,101]

443. ~outputs(sK4,bad) | ~program(sK10(sK4)) <- (4, 19, 33, 34) [subsumption resolution 442,407]

444. ~program(sK10(sK4)) <- (4, 19, 33, 34, 41) [subsumption resolution 443,427]

445. ~program(sK4) <- (4, 19, 21, 33, 34, 41) [resolution 444,172]

446. $false <- (4, 19, 21, 33, 34, 41) [subsumption resolution 445,101]

447. ~4 | ~19 | ~21 | ~33 | ~34 | ~41 [avatar contradiction clause 446]

450. 42 <=> program(sK10(sK4)) [avatar definition]

451. program(sK10(sK4)) <- (42) [avatar component clause 450]

452. ~program(sK10(sK4)) <- (~42) [avatar component clause 450]

454. 43 <=> sP0(sK4,sK10(sK4)) [avatar definition]

456. sP0(sK4,sK10(sK4)) <- (43) [avatar component clause 454]

480. halts2(sK9,X1) | ~program(X1) <- (8, 14) [subsumption resolution 142,117]

481. ~program(X0) | ~program(sK9) <- (8, 14) [resolution 480,117]

483. ~program(X0) <- (8, 14, 18) [subsumption resolution 481,159]

488. $false <- (8, 14, 18) [resolution 483,159]

495. ~8 | ~14 | ~18 [avatar contradiction clause 488]

507. ~program(sK4) <- (21, ~42) [resolution 452,172]

508. $false <- (4, 21, ~42) [subsumption resolution 507,101]

509. ~4 | ~21 | 42 [avatar contradiction clause 508]

511. halts2(sK10(sK4),sK10(sK4)) <- (43) [resolution 456,70]

514. ~program(sK10(sK4)) <- (8, 43) [resolution 511,117]

515. $false <- (8, 42, 43) [subsumption resolution 514,451]

516. ~8 | ~42 | ~43 [avatar contradiction clause 515]

519. sP0(sK4,sK10(sK4)) | ~outputs(sK4,bad) <- (4, 19, 34, 42) [subsumption resolution 442,451]

520. ~41 | 43 | ~4 | ~19 | ~34 | ~42 [avatar split clause 519,450,310,162,99,454,425]

521. ~program(sK4) | program(sK7(sK4)) <- (7, ~41) [resolution 426,113]

522. program(sK7(sK4)) <- (4, 7, ~41) [subsumption resolution 521,101]

523. $false <- (4, 7, ~32, ~41) [subsumption resolution 522,263]

```
524. ~4 | ~7 | 32 | 41 [avatar contradiction clause 523]

525. $false [avatar sat refutation
93,97,102,103,110,114,121,125,139,143,151,155,160,164,173,186,190,195,199,203
,208,217,219,307,326,349,351,353,357,368,386,395,397,413,422,428,447,495,509,
516,520,524]

% SZS output end Proof for COM003+1

% ------------------------------

% Version: Vampire 4.5.1 (commit )

% Termination reason: Refutation


% Memory used [KB]: 5117

% Time elapsed: 0.001 s

% ------------------------------

% ------------------------------
```

Summary of proof

Although the proof in this Appendix may seem formidable, it has a high-level intelligible structure:

Steps 1-4 assert axioms p1 – p4.

Step 5 asserts the Halting Theorem.

Step 6 is the negation of the Halting Theorem.

Steps 7-11 partially orthogonalize and standardize (a translation VAMPIRE calls “rectifying”) the naming of the variables occurring in Steps 1-6.

Step 12 removes the double negation at the front end of Step 11 (this is one of several translations VAMPIRE calls “flattening”.)

Steps 13, 14, 16, 18, and 20 replace the symbol “=>” with logical equivalents, i.e., replace expressions of the form X => Y with ~X | Y.

Steps 15, 17, and 19 flatten the clauses indicated.

Steps 21-26 rename/consolidate some recurring subclauses.

Steps 27-51 rectify and skolemize (see Chang and Keisler 2012, Chap. 3) various sentences.

Steps 52-86 transform various sentences to Clause Normal Form (cnf), a form that is required by some of VAMPIRE's inference rules.

Steps 88-267 rename various clauses and subclauses.

Steps 269-306, 307-326, 336-357, 362-368, 370-386, 389-395, 396-413, 416-422, 423-447, 450-495, 507-509, 511-516, and 519-524 collectively show that there is a partitioning of the proof space such that each inference chain in that space yields a contradiction.

Step 525 says that all the chains in the partition mentioned in Step 524 yield a contradiction. Thus, by proof by contradiction, Step 6 is false, i.e., the Halting Theorem follows from p1 – p4.

APPENDIX 2. Proof of independence of p1. See Sutcliffe 2008; Sutcliffe, Zimmer, and Schulz 2003, 2004; Sutcliffe and Suttner 2021 for syntax and ontology details.

Command line:

```
$ ../bin/vampire_rel.exe -saturation_algorithm fmb input_file > output_file
```

```
VAMPIRE input file (negates premise p1).

%------------------------------------------------------------------------------
fof(p1,axiom,
    (  ~  (? [X] :
        ( algorithm(X)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(X,Y,Z) ) )
   => ? [W] :
        ( program(W)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(W,Y,Z) ) ) )).

fof(p2,axiom,
    ( ! [W] :
        ( ( program(W)
          & ! [Y] :
              ( program(Y)
             => ! [Z] : decides(W,Y,Z) ) )
       => ! [Y,Z] :
            ( ( ( program(Y)
                & halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,bad) ) ) ) ) )).

fof(p3,axiom,
    ( ? [W] :
        ( program(W)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
```

```
            => ( halts3(W,Y,Y)
                & outputs(W,bad) ) ) ) )
   => ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) ) )).

fof(p4,axiom,
    ( ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) )
   => ? [U] :
        ( program(U)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ~ halts2(U,Y) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(U,Y)
                & outputs(U,bad) ) ) ) ) )).


%------------------------------------------------------------------------------
```

```
VAMPIRE output file (model showing independence of p1)


WARNING! Could not set resource limit: Virtual memory.
TRYING [1]
Finite Model Found!
% SZS status Satisfiable for COM003_negp1
% SZS output start FiniteModel for COM003_negp1
tff(declare_$i,type,$i:$tType).
tff(declare_$i1,type,good:$i).
tff(finite_domain,axiom,
      ! [X:$i] : (
         X = good
      ) ).

tff(declare_bad,type,bad:$i).
tff(bad_definition,axiom,bad = good).
tff(declare_algorithm,type,algorithm: $i > $o ).
tff(predicate_algorithm,axiom,
%          algorithm(good) undefined in model

).

tff(declare_program,type,program: $i > $o ).
tff(predicate_program,axiom,
           ~program(good)

).

tff(declare_decides,type,decides: $i * $i * $i > $o ).
tff(predicate_decides,axiom,
           ~decides(good,good,good)

).

tff(declare_halts2,type,halts2: $i * $i > $o ).
tff(predicate_halts2,axiom,
           ~halts2(good,good)

).

tff(declare_halts3,type,halts3: $i * $i * $i > $o ).
tff(predicate_halts3,axiom,
           ~halts3(good,good,good)

).

tff(declare_outputs,type,outputs: $i * $i > $o ).
tff(predicate_outputs,axiom,
           outputs(good,good)

).
```

```
% SZS output end FiniteModel for COM003_negp1
% ------------------------------
% Version: Vampire 4.5.1 (commit )
% Termination reason: Satisfiable

% Memory used [KB]: 4861
% Time elapsed: 0.001 s
% ------------------------------
% ------------------------------
```

APPENDIX 3. Proof of independence of p2. See Sutcliffe 2008; Sutcliffe, Zimmer, and Schulz 2003, 2004; Sutcliffe and Suttner 2021 for syntax and ontology details.

Command line:

```
$ ../bin/vampire_rel.exe -saturation_algorithm fmb input_file > output_file
```

```
VAMPIRE input file (negates premise p2).

%------------------------------------------------------------------------------
fof(p1,axiom,
    ( ? [X] :
        ( algorithm(X)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(X,Y,Z) ) )
   => ? [W] :
        ( program(W)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(W,Y,Z) ) ) )).

fof(p2,axiom,
    ( ~ ( ! [W] :
        ( ( program(W)
          & ! [Y] :
              ( program(Y)
             => ! [Z] : decides(W,Y,Z) ) )
       => ! [Y,Z] :
            ( ( ( program(Y)
                & halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,bad) ) ) ) ) ))).

fof(p3,axiom,
    ( ? [W] :
        ( program(W)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
```

```
            => ( halts3(W,Y,Y)
                & outputs(W,bad) ) ) ) )
   => ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) ) )).

fof(p4,axiom,
    ( ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) )
   => ? [U] :
        ( program(U)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ~ halts2(U,Y) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(U,Y)
                & outputs(U,bad) ) ) ) ) )).


%------------------------------------------------------------------------------
```

```
VAMPIRE output file (model showing independence of p2)



WARNING! Could not set resource limit: Virtual memory.
TRYING [1]
Finite Model Found!
% SZS status Satisfiable for COM003_negp2
% SZS output start FiniteModel for COM003_negp2
tff(declare_$i,type,$i:$tType).
tff(declare_$i1,type,good:$i).
tff(finite_domain,axiom,
      ! [X:$i] : (
         X = good
      ) ).

tff(declare_bad,type,bad:$i).
tff(bad_definition,axiom,bad = good).
tff(declare_algorithm,type,algorithm: $i > $o ).
tff(predicate_algorithm,axiom,
%         algorithm(good) undefined in model

).

tff(declare_program,type,program: $i > $o ).
tff(predicate_program,axiom,
           program(good)

).

tff(declare_decides,type,decides: $i * $i * $i > $o ).
tff(predicate_decides,axiom,
%         decides(good,good,good) undefined in model

).

tff(declare_halts2,type,halts2: $i * $i > $o ).
tff(predicate_halts2,axiom,
           ~halts2(good,good)

).

tff(declare_halts3,type,halts3: $i * $i * $i > $o ).
tff(predicate_halts3,axiom,
           ~halts3(good,good,good)

).

tff(declare_outputs,type,outputs: $i * $i > $o ).
tff(predicate_outputs,axiom,
           ~outputs(good,good)
```

```
).

% SZS output end FiniteModel for COM003_negp2
% ------------------------------
% Version: Vampire 4.5.1 (commit )
% Termination reason: Satisfiable

% Memory used [KB]: 4861
% Time elapsed: 0.140 s
% ------------------------------
% ------------------------------
```

APPENDIX 4. Proof of independence of p3. See Sutcliffe 2008; Sutcliffe, Zimmer, and Schulz 2003, 2004; Sutcliffe and Suttner 2021 for syntax and ontology details.

Command line:

```
$ ../bin/vampire_rel.exe -saturation_algorithm fmb input_file > output_file
```

```
VAMPIRE input file (negates premise p3).


%------------------------------------------------------------------------------
fof(p1,axiom,
    ( ? [X] :
        ( algorithm(X)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(X,Y,Z) ) )
   => ? [W] :
        ( program(W)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(W,Y,Z) ) ) )).

fof(p2,axiom,
    ( ! [W] :
        ( ( program(W)
          & ! [Y] :
              ( program(Y)
             => ! [Z] : decides(W,Y,Z) ) )
       => ! [Y,Z] :
            ( ( ( ( program(Y)
                  & halts2(Y,Z) )
                => ( halts3(W,Y,Z)
                  & outputs(W,good) ) )
              & ( ( program(Y)
                  & ~ halts2(Y,Z) )
                => ( halts3(W,Y,Z)
                  & outputs(W,bad) ) ) ) ) ) )).

fof(p3,axiom,
    ( ~ ( ? [W] :
        ( program(W)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
              => ( halts3(W,Y,Y)
                & outputs(W,good) ) )
            & ( ( program(Y)
```

```
                & ~ halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,bad) ) ) ) )
   => ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) ) ))).

fof(p4,axiom,
    ( ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) )
   => ? [U] :
        ( program(U)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ~ halts2(U,Y) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(U,Y)
                & outputs(U,bad) ) ) ) ) )).


%------------------------------------------------------------------------------
```

```
VAMPIRE output file (model showing independence of p3)


%----------------------------------------------------------------------------
fof(p1,axiom,
    ( ? [X] :
        ( algorithm(X)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(X,Y,Z) ) )
   => ? [W] :
        ( program(W)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(W,Y,Z) ) ) )).

fof(p2,axiom,
    ( ! [W] :
        ( ( program(W)
          & ! [Y] :
              ( program(Y)
             => ! [Z] : decides(W,Y,Z) ) )
       => ! [Y,Z] :
            ( ( ( program(Y)
                & halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,bad) ) ) ) ) )).

fof(p3,axiom,
    ( ~ ( ? [W] :
        ( program(W)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,bad) ) ) ) )
   => ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
```

```
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) ) ))).

fof(p4,axiom,
    ( ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) )
   => ? [U] :
        ( program(U)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ~ halts2(U,Y) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(U,Y)
                & outputs(U,bad) ) ) ) ) )).


%----------------------------------------------------------------------------
```

APPENDIX 5. Proof of independence of p4. See Sutcliffe 2008; Sutcliffe, Zimmer, and Schulz 2003, 2004; Sutcliffe and Suttner 2021 for syntax and ontology details.

Command line:

```
$ ../bin/vampire_rel.exe -saturation_algorithm fmb input_file > output_file
```

```
VAMPIRE input file (negates premise p4).

%----------------------------------------------------------------------------
fof(p1,axiom,
    ( ? [X] :
        ( algorithm(X)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(X,Y,Z) ) )
   => ? [W] :
        ( program(W)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(W,Y,Z) ) ) )).

fof(p2,axiom,
    ( ! [W] :
        ( ( program(W)
          & ! [Y] :
              ( program(Y)
             => ! [Z] : decides(W,Y,Z) ) )
       => ! [Y,Z] :
            ( ( ( program(Y)
                & halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,bad) ) ) ) ) )).

fof(p3,axiom,
    ( ? [W] :
        ( program(W)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,bad) ) ) ) )
```

```
   => ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) ) )).

fof(p4,axiom,
    ( ~ ( ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) )
   => ? [U] :
        ( program(U)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ~ halts2(U,Y) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(U,Y)
                & outputs(U,bad) ) ) ) ) )).


%----------------------------------------------------------------------------
```

```
VAMPIRE output file (model showing independence of p4)


WARNING! Could not set resource limit: Virtual memory.
TRYING [1]
Finite Model Found!
% SZS status Satisfiable for COM003_negp4
% SZS output start FiniteModel for COM003_negp4
tff(declare_$i,type,$i:$tType).
tff(declare_$i1,type,good:$i).
tff(finite_domain,axiom,
      ! [X:$i] : (
         X = good
      ) ).

tff(declare_bad,type,bad:$i).
tff(bad_definition,axiom,bad = good).
tff(declare_algorithm,type,algorithm: $i > $o ).
tff(predicate_algorithm,axiom,
%          algorithm(good) undefined in model

).

tff(declare_program,type,program: $i > $o ).
tff(predicate_program,axiom,
           program(good)

).

tff(declare_decides,type,decides: $i * $i * $i > $o ).
tff(predicate_decides,axiom,
%          decides(good,good,good) undefined in model

).

tff(declare_halts2,type,halts2: $i * $i > $o ).
tff(predicate_halts2,axiom,
           halts2(good,good)

).

tff(declare_halts3,type,halts3: $i * $i * $i > $o ).
tff(predicate_halts3,axiom,
           ~halts3(good,good,good)

).

tff(declare_outputs,type,outputs: $i * $i > $o ).
tff(predicate_outputs,axiom,
           outputs(good,good)

).
```

```
% SZS output end FiniteModel for COM003_negp4
% ------------------------------
% Version: Vampire 4.5.1 (commit )
% Termination reason: Satisfiable

% Memory used [KB]: 4861
% Time elapsed: 0.031 s
% ------------------------------
% ------------------------------
```

APPENDIX 6. Proof of consistency of Burkholder 1987 axioms. See Sutcliffe 2008; Sutcliffe, Zimmer, and Schulz 2003, 2004; Sutcliffe and Suttner 2021 for syntax and ontology details.

Command line:

```
$ ../bin/vampire_rel.exe -saturation_algorithm fmb input_file > output_file
```

```
VAMPIRE input file (same as Figure 3).

%----------------------------------------------------------------------------
fof(p1,axiom,
    ( ? [X] :
        ( algorithm(X)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(X,Y,Z) ) )
   => ? [W] :
        ( program(W)
        & ! [Y] :
            ( program(Y)
           => ! [Z] : decides(W,Y,Z) ) ) )).

fof(p2,axiom,
    ( ! [W] :
        ( ( program(W)
          & ! [Y] :
              ( program(Y)
             => ! [Z] : decides(W,Y,Z) ) )
       => ! [Y,Z] :
            ( ( ( program(Y)
                & halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Z) )
             => ( halts3(W,Y,Z)
                & outputs(W,bad) ) ) ) ) )).

fof(p3,axiom,
    ( ? [W] :
        ( program(W)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts3(W,Y,Y)
                & outputs(W,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
```

```
            => ( halts3(W,Y,Y)
               & outputs(W,bad) ) ) ) )
   => ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) ) )).

fof(p4,axiom,
    ( ? [V] :
        ( program(V)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,good) ) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(V,Y)
                & outputs(V,bad) ) ) ) )
   => ? [U] :
        ( program(U)
        & ! [Y] :
            ( ( ( program(Y)
                & halts2(Y,Y) )
             => ~ halts2(U,Y) )
            & ( ( program(Y)
                & ~ halts2(Y,Y) )
             => ( halts2(U,Y)
                & outputs(U,bad) ) ) ) ) )).


%------------------------------------------------------------------------------
```

```
VAMPIRE output file (model showing consistency of Burkholder 1987 axioms)



WARNING! Could not set resource limit: Virtual memory.
TRYING [1]
Finite Model Found!
% SZS status Satisfiable for COM003
% SZS output start FiniteModel for COM003
tff(declare_$i,type,$i:$tType).
tff(declare_$i1,type,good:$i).
tff(finite_domain,axiom,
      ! [X:$i] : (
         X = good
      ) ).

tff(declare_bad,type,bad:$i).
tff(bad_definition,axiom,bad = good).
tff(declare_algorithm,type,algorithm: $i > $o ).
tff(predicate_algorithm,axiom,
%         algorithm(good) undefined in model

).

tff(declare_program,type,program: $i > $o ).
tff(predicate_program,axiom,
           ~program(good)

).

tff(declare_decides,type,decides: $i * $i * $i > $o ).
tff(predicate_decides,axiom,
%         decides(good,good,good) undefined in model

).

tff(declare_halts2,type,halts2: $i * $i > $o ).
tff(predicate_halts2,axiom,
           halts2(good,good)

).

tff(declare_halts3,type,halts3: $i * $i * $i > $o ).
tff(predicate_halts3,axiom,
           ~halts3(good,good,good)

).

tff(declare_outputs,type,outputs: $i * $i > $o ).
tff(predicate_outputs,axiom,
```

```
            ~outputs(good,good)

).

% SZS output end FiniteModel for COM003
% ------------------------------
% Version: Vampire 4.5.1 (commit )
% Termination reason: Satisfiable

% Memory used [KB]: 4861
% Time elapsed: 0.041 s
% ------------------------------
% ------------------------------
```