WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

Master Thesis

Certainly Terminating Wikifunctions

Eddie Begović

Date of Birth: 15.07.1996
Student ID: 11801549

Subject Area: Digital Economy

Program Code: 066/960

Supervisor: Univ.-Prof. Dr. Axel Polleres

Date of Submission: September 14, 2026

Department of Information Systems & Operations Management, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR WIRTSCHAFTSINFORMATIK UND OPERATIONS MANAGEMENT DEPARTMENT OF INFORMATION SYSTEMS AND OPERATIONS MANAGEMENT

Contents

List of Figures

List of Tables

Listings

Abstract

Wikifunctions is a collaborative Wikimedia project that provides an open library of functions with implementations in multiple programming languages. Since anyone can contribute code, the question arises whether these functions are guaranteed to terminate, which is a question that cannot be answered in general. While several termination analysis tools exist, their applicability to such a crowd-sourced repository has barely been addressed. In this thesis, we therefore extract 1,968 Python implementations from the official Wikifunctions dump, categorize them by structure and then translate the 43 translatable integer-loop functions into the input formats of five established tools, validating each translation through differential testing. Combined, these tools decide 33 of the 43 functions without ever contradicting each other which underlines how strongly complementary these tools are. For 20 of these functions, only a single tool actually provides a verdict. The analysis also uncovers a genuine non-termination bug in a live Wikifunctions function.

1 Introduction

Recent developments in large language models have produced systems that no longer only generate text but also call external tools and executable functions on their own. In such agentic settings, a repository like Wikifunctions becomes more than a reference work. Wikifunctions is an openly editable library of functions operated by the Wikimedia Foundation [52], launched as part of the Abstract Wikipedia initiative, whose long term vision is to generate Wikipedia articles from language independent content [51]. Its functions are meant to be found, composed, and executed automatically, by humans and increasingly by machines.

Executing a function that someone else wrote raises a basic question, mainly if this call will ever terminate. A function that does not terminate on some input can block whatever system called it and waste computational resources, and in a setting where functions are selected and executed automatically, no human is there to inspect the code first. If repositories like Wikifunctions are to serve as building blocks for automated systems, it would be valuable to know in advance which of their functions are guaranteed to terminate.

From a theoretical point of view, this question cannot be answered in general. Turing showed that no algorithm can decide for every program whether it halts [47]. Decades of research have shown, however, that for restricted classes of programs, in particular loops over integer variables, termination can often be proven automatically, most prominently by synthesizing ranking functions [21, 39]. This research has produced mature tools such as AProVE [28], MuVal [49], T2 [19], KoAT [36], and VeryMax [16], which compete annually in the International Termination Competition [7].

These tools, however, are developed and evaluated on curated benchmark sets. Wikifunctions is a different environment, as its content is crowd sourced, written in general purpose languages such as Python, whereas none of the tools accepts python. Whether the tools can be applied to such a repository at all, what it takes to prepare real world functions for them and how much they can actually prove there, has to our knowledge not been investigated. This thesis addresses that gap. We deliberately do not develop a new termination tool. Instead, we ask how far one can get with the tools that already exist, and structures this investigation around three research questions:

RQ1 Which existing termination analysis tools are suitable for analyzing implementations extracted from Wikifunctions?

RQ2 To what extent can these tools automatically prove termination or detect non-termination for Wikifunctions python implementations?

RQ3 How do the results of different termination analysis tools compare on the same set of Wikifunctions implementations?

To answer these questions, we built a pipeline that connects the raw Wikifunctions data to the tools. It extracts all Python implementations from the

official Wikifunctions XML dump, categorizes them by structure using abstract syntax tree analysis, translates the integer loop functions into C programs and integer transition systems from a single shared intermediate representation and validates every translation against the original Python through a differential test. The five tools are then run in the same Docker setup used by the Termination Competition [7] and their outputs are mapped onto a common set of verdicts so that the results become comparable.

The evaluation leads to findings on three levels. First, the reachable portion of the repository turns out to be small, as only a fraction of the extracted Python implementations can be translated faithfully into the tools input formats. This already says as much about the nature of Wikifunctions content as about the tools. Second, on the functions that can be reached the tools prove more than one might expect. They reach a verdict on most of them, they also never contradict each other where their results overlap and they turn out to be strongly complementary rather than redundant, with the two C based tools clearly outperforming the three that operate on dedicated transition system formats. Third, the analysis did not merely confirm termination but uncovered a genuine non-termination on a live Wikifunctions function that fails to terminate on a specific input. Together these findings show that automated termination analysis can provide real value for a collaborative function repository, beyond confirming what is already known.

The remainder of this thesis is structured as follows. Section 2 introduces the necessary background on termination, transition systems and ranking functions. Section 3 reviews related work on termination analysis techniques and describes the five tools used in this study. Section 4 presents the pipeline, from the extraction of the dataset to the execution of the tools and the design of the evaluation. Section 5 reports the results along the three research questions. Section 6 discusses the limitations of the study and directions for future work, and Section 7 concludes.

2 Preliminaries

This section introduces the fundamental concepts on which this study is based. It aims to provide the necessary background for understanding the main ideas and discussions presented throughout this thesis.

2.1 Wikifunctions

Wikifunctions is an open, collaborative platform that provides a shared repository of functions which can be used, modified, and extended by anyone. Each function includes descriptions, parameters, test cases, and multiple implementations in different programming languages, making it reusable across a wide range of applications. Similar to projects like Wikipedia, it aims to build an open, collaboratively maintained global library of code that can be easily accessed and combined [52].

The platform was originally conceived as a central component of a multilingual Wikipedia architecture, where its primary role is to host rendering functions that transform language independent abstract content into natural language text. The broader vision is to allow Wikipedia content to be written once and automatically rendered in hundreds of languages, rather than requiring independent editorial communities to maintain separate articles for each language edition [51]. In its collaborative nature, it has turned into a general repository of broader variety of functions.

A function in Wikifunctions can have multiple implementations, either written in a language such as Python or JavaScript, or composed from other existing functions. This makes the platform somewhat different from a traditional code repository. The focus is on what a function computes rather than how it is implemented, and different implementations of the same function can be compared and tested against each other. Functions can be called from a variety of environments, including RESTful APIs, command-line interfaces, and Jupyter notebooks, which makes the platform relatively easy to integrate into automated workflows [51].

As an example, take the factorial function, which multiplies all positive integers from 1 up to some number n. On Wikifunctions this function takes a single natural number as input and like most functions on the platform, is not tied to one programming language, as it can have several independent implementations, for instance in Python or JavaScript, which are all expected to return the same result for the same input. This already shows one of the key ideas behind Wikifunctions, namely that a functions definition is kept separate from how it is actually implemented, so that different language specific versions can exist side by side under the same function. We come back to the Python implementation of this function later in this thesis, where it is used to show how a loop can be modeled as a transition system (Section 2.5.1).

A second, contrasting example is the *Ackermann function*, shown in Figure 1. It takes two non-negative integers and is defined recursively as

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ A(m-1,\,1) & \text{if } m > 0 \text{ and } n = 0, \\ A(m-1,\,A(m,\,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases} \tag{1}$$

Unlike the factorial, whose computation is a simple loop, the Ackermann function is defined by deeply nested recursion and is a well known example of a function that is not primitively recursive. It always terminates, yet its values grow extremely quickly even for tiny inputs. On Wikifunctions it is stored as the function `Z14742`. Because its termination rests on nested recursion rather than on a simple integer loop, it lies outside the class of functions analyzed in this thesis, where recursion is excluded (Section 4.2). In this thesis, Wikifunctions is used as a source of functions that can be automatically retrieved and analyzed.

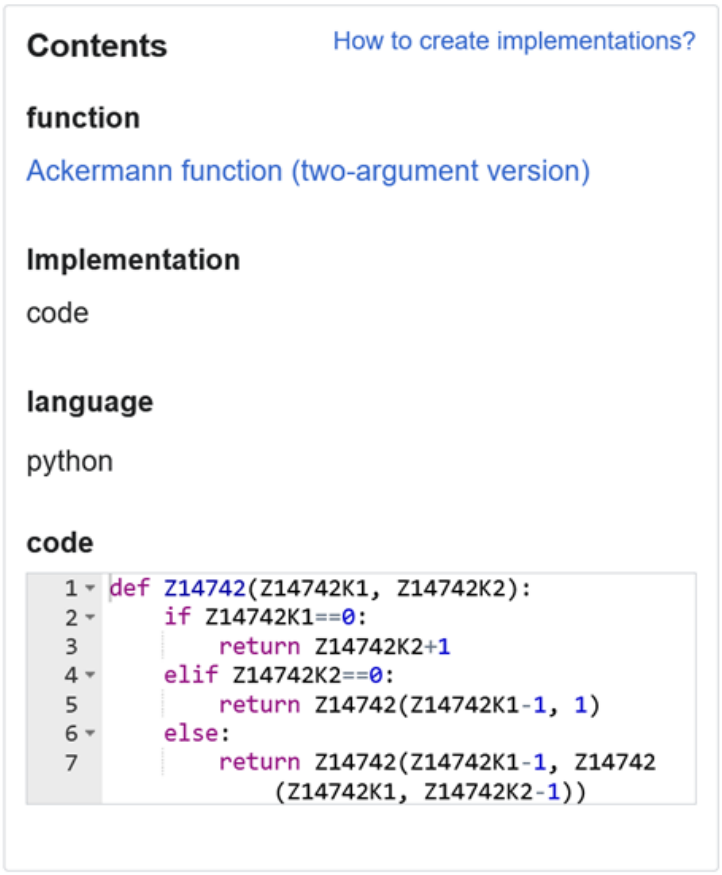

Figure 1: The Ackermann function on Wikifunctions (function `Z14742`), shown with its recursive Python implementation.

2.2 Termination

One of the central properties of functions (and computation in general) is termination. A function terminates if its execution always halts and returns a result, no matter which input it is given. A function is called non-terminating if there is at least one input for which it keeps running forever. This matters because a function that never halts also never produces an answer, which is of little use in practice.

The simplest way a function can fail to terminate is a loop whose condition never becomes false. Listing 1 shows such a case. Because the condition *True* always holds, the loop body keeps repeating and the program never leaves the loop.

```
def loop():
    while True:
        pass
```

Listing 1: A trivially non-terminating loop.

Recursion can fail to terminate in much the same way, namely when the base case is never reached. The function in Listing 2 does have a stopping condition, `n == 0`, but every recursive call increases `n`, so for any positive input the condition is never met and the function keeps calling itself.

```
def count_up(n):
    if n == 0:
        return 0
    return count_up(n + 1)
```

Listing 2: A recursive function whose base case is never reached.

Deciding whether an arbitrary program terminates is generally undecidable, which is the well known halting problem. There is no single algorithm that can tell, for every possible program, whether it halts [47]. For this reason termination analysis tools do not try to solve the problem in full generality. Instead they aim to prove termination for as many programs as they can, usually by finding some quantity related to the loop condition that becomes smaller with every step, which is known as a ranking function, which will be discussed in Section 3.1.

2.3 Transition Systems

For this subsection we follow the definitions of [21]. Programs can be formally modeled as transition systems. A transition system provides a mathematical representation of how a program evolves during execution by describing the possible program states and the transitions between them.

A transition system is typically defined as a tuple

$$S = \langle V, \Theta, T \rangle$$

With V representing a finite set of variables, Θ representing the initial condition and T representing a finite amount of transitions. The initial condition is not necessarily a concrete value. Instead, it describes which states an execution of the program is allowed to start in. It can require a fixed value like $i = 0$, but it may just as well leave the variables only partially constrained, as in $i \geq 0$ [37]. We will explain this tuple in a following subsection with an example.

2.4 Cycles and Termination

From [21] we understand that to prove that a program eventually terminates, it is not necessary to analyze every possible execution step in detail. Instead,

the focus is placed on cycles in the program. A cycle represents a part of the program that can be executed repeatedly, such as a loop.

The authors points out that proving termination essentially comes down to showing that these cycles cannot be executed forever. In other words, if we can show that every cycle eventually terminates, then the whole system must also terminate. To achieve this, ranking functions are used.

2.5 Ranking Functions

We follow the definition of [21] again in order to explain ranking functions. A ranking function, in its simplest form, is a mathematical function used to prove the termination of a transition system. More precisely, a ranking function maps program states to values in a well-founded domain, for example the domain of natural numbers $\mathbb{N}$, which is also bounded from below. A relation is well-founded if it does not allow an infinite strictly decreasing sequence [40]. This explains why $\mathbb{N}$ is commonly used. Starting from a value such as 3, a strictly decreasing sequence can only be

$$3 > 2 > 1 > 0,$$

after which no smaller natural number exists. In contrast, the integers $\mathbb{Z}$ are not well-founded under $>$, since the sequence

$$3 > 2 > 1 > 0 > -1 > -2 > \cdots$$

can continue indefinitely. Similarly, the non-negative real numbers $\mathbb{R}$ are not well-founded under $>$, since a sequence such as

$$3 > \frac{3}{2} > \frac{3}{4} > \frac{3}{8} > \cdots$$

decreases forever while only approaching 0. Since program termination requires stopping after finitely many transitions, merely approaching a lower bound is not sufficient. Therefore, if each loop iteration decreases a ranking function whose values lie in a well-founded domain, the loop cannot execute forever and must eventually terminate. We will also take a look at different approaches for computing linear ranking functions, for example [39] and [22].

2.5.1 Basic Loop Example

To better understand how loops behave during execution, we consider a simple example inspired by an implementation of the factorial function available on the Wikifunctions platform.

The factorial of a number n is defined as the product of all positive integers from 1 up to n. For example, the factorial of 4 is calculated as

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24.$$

One possible implementation of this function according to Wikifunctions is shown below.

```
def Z13667(Z13667K1):
    k = 1
    for i in range(1, Z13667K1 + 1):
        k *= i
    return k
```

Listing 3: Factorial function from Wikifunctions

This function computes the factorial step by step. At the beginning, the variable k is set to 1. The loop then runs over the variable i, starting at 1 and ending at the input value, i.e. the variable Z13667K1. This ensures that the loop continues and monotonically increases each iteration ensuring that Z13667K1 is eventually reached. In every iteration, the current value of i is multiplied with k, so that k stores the result of the factorial.

Looking at this loop more closely, we can identify its basic components. Starting off with the tuple we discussed before. Here we will quickly show the set of variables, the initial condition as well as the transition τ. Here, primed variables denote the value of a variable after taking a transition. For example, k' is the value of k in the next state, while k is the value in the current state. Here n stands for the input parameter, which is called *Z13667K1* in the code.

$$V = \{n, i, k\}$$

$$\Theta : n \geq 0 \wedge i = 1 \wedge k = 1$$

$$\tau : i \leq n \wedge k' = k \cdot i \wedge i' = i + 1 \wedge n' = n$$

The *loop guard* is $i \leq Z13667K1$. It determines whether the loop body is executed. As long as the current value of i is less than or equal to $Z13667K1$, the loop performs another iteration. When i becomes greater than $Z13667K1$, the loop stops. The loop body consists of the multiplication step k *= i, which can be expressed as $k' = k \cdot i$, where k' represents the value of k after the update.

While the loop executes, certain properties hold before the first iteration and are preserved by every subsequent iteration. Such properties are called *loop invariants*, because they remain valid every time the loop condition is checked [27].

Building on these invariants, a possible ranking function for this loop is $-i + Z13667K1$. It is bounded from below and decreases in each iteration, which means the loop cannot run forever.

In this example, these properties are still relatively easy to understand. However, in more complex programs, such invariants are often not obvious from simply looking at the code. For this reason, automatic invariant generators, as introduced by Cousot and Halbwachs [25], are needed to identify these properties and the construction of ranking functions.

To show a more complex example, we now consider the computation of the greatest common divisor (GCD), inspired by implementation Z13612 available on Wikifunctions.

The greatest common divisor of two integers is defined as the largest positive integer that divides both numbers without leaving a remainder. For example, the greatest common divisor of 48 and 18 is calculated as

$$\gcd(48, 18) = 6,$$

since 6 is the largest natural number that divides both 48 and 18 exactly.

One efficient way to compute the greatest common divisor is the Euclidean algorithm. The key idea of this method is that the GCD of two numbers does not change if the larger number is replaced by the remainder of dividing it by the smaller number. This process is repeated until the remainder becomes zero. At that point, the remaining number is the greatest common divisor.

One possible implementation of this function according to Wikifunctions is shown below.

```
def Z13612(Z13612K1, Z13612K2):
    if (Z13612K1 == 0):
        return Z13612K2
    x = Z13612K1
    y = Z13612K2
    while(y):
        x, y = y, x % y
    return x
```

Listing 4: Greatest common divisor function in Python, reproduced from Wikifunctions

The *loop guard* is $y \neq 0$, written as *while(y)* in Python. It determines whether the loop body is executed and is therefore part of the transition τ, which may only be taken as long as y is non-zero [21]. The loop body consists of simultaneous assignment *x, y = y, x % y*, which is expressed by the two updates $x' = y$ and $y' = x \bmod y$. The case *Z13612K1* $= 0$ is handled before the loop is reached and is therefore excluded by the initial condition Θ.

This function computes the greatest common divisor step by step. At the beginning, the variables x and y are assigned the input values. If the first value is zero, the function immediately returns the second value, since the GCD of 0 and a number is the number itself.

The loop continues as long as the variable y is not equal to zero. Inside the loop body, the values of x and y are updated simultaneously. The variable x takes the value of y, while y is replaced by the remainder of dividing the previous value of x by y. This remainder is always smaller than the previous absolute value of y.

Looking at this loop more closely, we can again identify its components and express them as a transition system in the same way as before.

$$V = \{x, y\}$$

$$\Theta : x > 0 \wedge y \geq 0$$

$$\tau : y \neq 0 \wedge x' = y \wedge y' = x \bmod y$$

Compared to the factorial example, this loop is more complex because the variables do not change in a simple linear manner. Instead of decreasing by a fixed amount, the value of y is replaced by a remainder that depends on both variables. This makes it less obvious how the loop progresses toward termination. Nevertheless, each iteration reduces the value of y, and therefore will once y reaches zero, the loop terminates and the final value of x is returned as the greatest common divisor.

3 Related Work

With the growing interest in termination analysis and ranking functions, a large body of literature has developed that presents different methods for synthesizing ranking functions and evaluating their performance. This section gives an overview of this literature and provides the foundation for the research carried out in this thesis.

3.1 Linear Ranking Function Synthesis

In this section, we focus on methods for computing linear ranking functions. We examine three foundational papers that present different approaches to synthesizing linear ranking functions. These works are considered fundamental in the field and serve as the theoretical basis for many modern termination analysis techniques.

3.1.1 Invariant Based Ranking Function Synthesis

We illustrate the concept using the following example program from [21], shown in Listing 5.

```
local i, j, k : integer
while i <= 100 and j <= k do
    (i, j) := (j, i + 1)
    k := k - 1
end
```

Listing 5: Example loop from [21]

Just by looking at the program it is not immediately obvious to tell if the program terminates or not. It is also not straightforward to derive the invariants or to determine whether the variables are bounded. In the following we will describe the approach proposed by [21], which systematically constructs ranking functions to address these challenges.

In a first step an invariant generator is assumed, that can compute the invariants automatically. For our example, we assume the invariant generator that is based on [25] as well as assuming that the generated invariants are represented as a system of linear inequalities. The next step then is to create auxiliary variables, which are variables that copy and hold the value of all the variables when the program enters the loop for the first time. These are used to make the invariants stronger, as these are always constant throughout the loop cycle. Although the assignment $k := k - 1$ implies that k decreases in every iteration and therefore makes k appear to be a natural candidate for a ranking function, this is not sufficient on its own, as k has no obvious lower bound in the original loop. Based on our example the invariants generated would be

$$i \leq 100 \quad \wedge \quad j - k \leq 0$$

While this invariant bounds i from above, j and k are not bounded. After adding the auxiliary variables, the invariant generator is applied again. Since the auxiliary variables store the values from the moment the loop is entered and remain constant during the loop, they can be used as symbolic bounds for the changing variables.

The augmented invariant are

$$i \leq 100 \ \wedge \ j - k \leq 0 \ \wedge \ k - k_0 \leq 0$$

in which the variables i, j, and k are all bounded from above. In particular, $j \leq k_0$ follows from the inequalities $j - k \leq 0$ and $k - k_0 \leq 0$. These bounds are what make a ranking function possible. The expression $-i - j$ decreases by one with each iteration of the cycle and its value is bounded from below by $-100 - k_0$, so it defines a ranking function for the loop and proves that it terminates.

The second step of the algorithm is to compute linear expressions over the program variables that remain bounded during the execution of the loop. An expression is considered bounded if there exists a constant lower bound such that its value is never smaller than this bound for all reachable states of the loop. To determine such expressions, the algorithm relies on the loop invariants obtained in the previous step.

In the third step of the algorithm, the goal is to compute decreasing expressions over the program variables. The paper specifically requires these expressions to be discretely decreasing because the generated ranking functions map into a rational-valued domain rather than directly into the natural numbers. Therefore, ordinary decrease is not sufficient as the decrease must have a fixed minimum size. An expression f is discretely decreasing if there exists a positive

constant $\Delta > 0$ such that, whenever the program transitions from a state s to a successor state s', the following inequality holds:

$$f(s) \geq f(s') + \Delta.$$

Thus, the value of f must decrease by at least Δ in every transition.

In the final step, the algorithm combines the bounded expressions with the decreasing expressions by computing their intersection. If this intersection contains an expression that is bounded from below and decreases by at least a fixed positive amount during each transition, this expression can be used as a ranking function and provide a proof that the loop cannot run indefinitely. For the loop in Listing 5 this intersection yields $-i - j$ as a ranking function, since it decreases by one with each iteration of the cycle and is bounded from below by $-100 - k_0$, which proves that the loop terminates. While the approach provides a method for automatically constructing linear ranking functions, it relies on the availability of sufficiently strong loop invariants. In practice, the effectiveness of the method therefore depends on the strength of the invariant generator used.

Although the proposed algorithm shows promising results, it still has some limitations. First, as the size and complexity of the program increase, the polyhedral cone representations used by the algorithm can grow very large. A polyhedral cone is used to represent whole sets of linear inequalities and possible linear expressions. For example, in Listing 5 invariants such as $i \leq 100$, $j - k \leq 0$ and $k - k_0 \leq 0$ are first expressed as linear inequalities and then stored as coefficient vectors. A coefficient vector simply lists the coefficients of an inequality, so that $k - k_0 \leq 0$ is stored as the vector with 1 in the position of k, -1 in the position of k_0 and zero everywhere else. These vectors are called generators, because they act as the basic building blocks from which the cone, and therefore further consequences of the inequalities, can be constructed.

The algorithm also computes the polar of a cone, which is the set of all vectors u with $u \cdot v \leq 0$ for every v in the cone. This gives a dual view of the same information, since the vectors v in the cone are the coefficient vectors of the inequalities the system implies, while the vectors u in its polar are the solutions of that system. However, the polar also has to be stored using its own generators. The problem is that, in some cases, the polar may require far more generators than the original cone. Therefore, when programs have many variables or complex transition relations, storing both the cone and the generators of its polar can cause the internal representation to grow rapidly, leading to excessive memory usage.

Second, the algorithm only searches for linear ranking functions. Therefore, it may fail to find a ranking function even if the program does terminate, in cases where a non-linear ranking function would be required.

3.1.2 Constraint Based Ranking Function Synthesis

A different approach was proposed by [39]. Instead of computing bounded and decreasing expressions separately, their method reduces the synthesis of linear ranking functions to solving a system of linear constraints derived directly

from the loop guard and update relations of unnested loops. This method in comparison is complete, meaning that if a linear ranking function exists for the given loop, the algorithm is guaranteed to find it.

This approach focuses on two specific types of programs, so called *simple while loop programs* and *linear arithmetic simple while* (*LASW*) *programs*. A simple while program is a restricted form of loop. Its guard consists only of simple conditions combined with conjunctions, its body contains only variable updates, and these updates are interpreted as simultaneous assignments. For example, in the loop

```
while (Cond1 and ... and Condm) do
    Simultaneous Updates
od
```

Listing 6: General form of a simple while program from [39]

the conditions $Cond_1, \ldots, Cond_m$ form the loop guard, and the loop body consists only of updates that are applied at the same time in each loop iteration.

A *linear arithmetic simple while* (LASW) program is a simple while program in which the loop guard and the update relations can be expressed as linear inequalities over program variables. In the update relations, unprimed variables denote the values before the loop body is executed, while primed variables denote the values after one execution of the loop body.

To illustrate *LASW* programs, we consider the following example from [39].

```
while (i-j >= 1) do
    (i,j) := (i-Nat,j+Pos)
od
```

Listing 7: Example from [39]

This loop contains nondeterministic updates, where Nat denotes any nonnegative integer and Pos denotes any positive integer.

Following the approach, the program can be represented as a system of linear inequalities over the program variables and their primed versions:

$$-i + j \leq -1, \quad -i + i' \leq 0, \quad j - j' \leq -1.$$

The variables i' and j' denote the values of i and j after the execution of the loop body.

These inequalities encode both the loop condition and the update relations. In particular, the nondeterministic updates are captured by the constraints $i' \leq i$ and $j' \geq j + 1$. In this example, the method yields the ranking function

$$\rho(i, j) = \begin{cases} i - j & \text{if } i - j \geq 1, \\ 0 & \text{otherwise,} \end{cases}$$

which returns $i-j$ as long as the loop guard still holds and 0 once it does not. During each iteration, i can only stay the same or decrease, while j increases by at least one. As a result, the difference $i - j$ becomes smaller in every iteration.

Thus, for a given LASW program, the loop guard and update relation are first encoded as a linear transition relation over current-state and next-state variables. The method then derives a constraint problem from this transition relation, where a solution determines the coefficients of a linear ranking function. If this constraint problem is satisfiable, termination of the loop is proven. If it is unsatisfiable, then no linear ranking function exists for the given LASW program. This does not imply that the loop is non-terminating, but only that termination cannot be established by a linear ranking function.

3.1.3 Control Flow Based Ranking Function Synthesis

The third approach of computing a linear ranking function is mentioned in [22], in which 2 methods are being explained. These two methods consist of a more general, but computationally expensive algorithm, and a faster heuristic algorithm that is less general because it restricts the search space of possible ranking functions. We will start explaining the general algorithm first before moving on to the heuristic one.

The algorithm consists of two phases, with the first phase preparing the program for computing the ranking function. This means its generating the invariants and extracting a control flow graph (CFG) from the program. It removes everything in that graph that could not happen in a real execution.

Following the definition of Allen [11], a control flow graph (CFG) can be understood as a way of representing the possible flow of execution in a program. In a CFG, each node represents a basic block, meaning a sequence of instructions with one entry point and one exit point, while the directed edges show how control can move from one block to another. This means that the CFG does not focus on every single instruction individually, but rather on the larger structure of how different parts of the program may be executed. By representing a program in this way, it becomes easier to analyze relationships such as which blocks can follow others, where loops may occur, and how execution paths are connected, which is why CFGs are important for control flow analysis. To illustrate the CFG, we extracted it from the code shown in Listing 3.

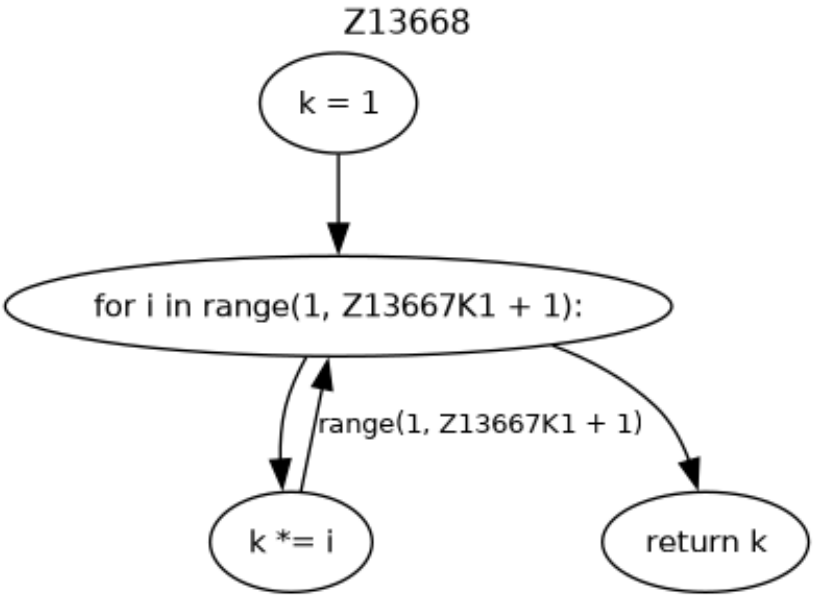

Figure 2: Control flow graph of the factorial function shown in Listing 3.

In the second phase the ranking functions are getting computed by the mutually recursive procedures named $rank1$ and $rank2$. The procedure $rank1$ extracts the maximum strong connected subgraphs (MSCS). If all MSCS are trivial, $rank1$ succeeds immediately. If there are non-trivial MSCS then $rank2$ is called, which is a multistep process. An MSCS is said to be trivial when it consists of a single vertex and no edges. We will now briefly look into the steps of rank2.

First it partitions the program variables into a modified category which are variables that are changed by some transitions and into non-modified variables which are preserved through all transitions. The next step consists generating the tail invariants. Tail invariants differ from ordinary invariants in that they are not required to hold every time a location is reached and instead they must hold eventually along any infinite computation that remains inside the MSCS. For the loop in Listing 3, for example the assertion $i \geq 2$ is not an ordinary invariant at the loop head, since i equals one when the loop is first entered, but it is a tail invariant, because it holds on every later visit. This is what lets the method ignore the first pass through a loop, which matters for structures such as do-while loops where the loop condition is only evaluated at the end of an iteration. Next, it will generate a set of all linear expressions over the modified variables which do not increase under the transitions. Then it computes for each transition the set of expressions that are bounded below and decrease. For each transition, if there exists an expression δ that is decreasing on that transition, bounded below, and non-increasing over the entire MSCS, then δ is

a ranking function and the transition is removed from the MSCS. If there is no such transition, rank2 fails. Otherwise, rank2 calls rank1 and the process repeats.

The heuristic approach works a bit different, as it is based on two observations from the programs the authors have considered. The first observation is that programs the authors considered are written in structured programming languages, meaning they have reducible CFGs. A reducible CFG is one where every loop has a header node that must be passed through to enter the loop. More precisely, a vertex v dominates a vertex w if every proper path to w passes through v and a CFG is reducible if every strongly connected subgraph contains a vertex that dominates it, which is its header. Reducible CFGs are well structured, since their loops are properly nested. Based on this observation, the authors devised an invariant generator that takes advantage of the structure of reducible CFGs. When the algorithm encounters a non-trivial strongly connected subgraph (SCS), it avoids computing invariants through repeated iteration. Instead, it constructs an approximate invariant from the current assertions, which are represented as linear inequalities. It first keeps the information about variables that are not modified inside the SCS. Then it checks whether the existing constraints imply useful bounds on the variables that are modified. If these bounds depend on non-modified variables and remain valid throughout the loop, they are added to strengthen the approximation.

The second observation is that, once variables modified inside an SCS are distinguished from variables that are non-modified, many loops in the programs considered by the authors can be proven terminating with single variable ranking functions. Instead of searching for arbitrary linear combinations of variables, the heuristic algorithm checks whether an individual modified variable, or its negation, is sufficient to show progress. This avoids the computational overhead and can make the search for ranking functions faster. However, the approach is less general than the first algorithm, since it may fail when termination can only be shown using a ranking function that combines multiple variables.

3.2 Extensions Beyond Linear Ranking Functions

Besides simple linear ranking functions, more expressive variants exist. Within the class of linear ranking functions, lexicographic linear ranking functions represent an important extension. These are particularly useful in situations where no single linear ranking function can be found, although the program itself is terminating.

Lexicographic Ranking Function

Instead of relying on a single global ranking function, this approach constructs several ranking functions that are combined into a tuple. This allows termination to be shown in multiple stages and often increases the likelihood of successfully finding a suitable ranking function.

The solving strategy proposed in [17] consists of two main parts. First, a solver determines the numerical values of the coefficients of the ranking func-

tions.

Second, a search procedure determines how multiple ranking functions should be arranged to form a valid lexicographic ranking function. This procedure systematically tests different possible arrangements and is theoretically complete, which means that if a lexicographic ranking function exists, the method will eventually find it.

A lexicographic ranking function consists of an n-tuple of affine functions such that, in each iteration, one component decreases by at least a fixed positive amount, while all earlier components in the tuple do not increase.[17]

Consider the following loop, which is used in [17] to demonstrate a lexicographic ranking function:

```
while (y1 != y2) {
    if (y1 > y2)
        y1 = y1 - y2;
    else
        y2 = y2 - y1;
}
```

Listing 8: GCD loop from [17] in C

Assume that y_1 and y_2 are positive integers. Since the loop only runs while $y_1 \neq y_2$, the else branch is taken exactly when $y_2 > y_1$. In this loop, one of the two variables therefore decreases in every iteration. If $y_1 > y_2$, then y_1 decreases and if $y_2 > y_1$, then y_2 decreases. Since the decrease depends on which branch is taken, the progress of the loop is described by more than one measure [17].

Instead of using a single ranking function, the paper uses the lexicographic ranking function

$$\rho(y_1, y_2) = \langle y_2 - 2,\ y_1 - 2 \rangle.$$

The first component of the ranking tuple is $y_2 - 2$. If the branch $y_2 > y_1$ is executed, then y_2 decreases, and with it the first component of the tuple. If instead the branch $y_1 > y_2$ is executed, then y_2 remains unchanged. In this case, the first component does not increase, while the second component, $y_1 - 2$, decreases.

Hence, each iteration decreases the tuple in lexicographic order, either the first component decreases directly, or if it remains unchanged, the second component decreases. Since this lexicographic measure strictly decreases with every iteration, the loop must eventually terminate.

This approach is particularly useful for loops in which progress cannot be captured by a single linear ranking function, but can still be described by several measures that decrease in a fixed order. [17]

Non-linear termination analysis for polynomial programs

A more expressive approach beyond linear and lexicographic linear ranking functions is presented in [18]. The paper studies the termination of multipath

polynomial programs, that is, loops with multiple execution paths whose guards and assignments may contain polynomial expressions. This class of programs is expressive enough to support practical code abstraction and analysis.

The proposed method is a form of non-linear termination analysis. It is non-linear in two senses, first, the guards and assignments of the loop need not be linear and second, the method can prove termination for loops that do not have a linear ranking function. The approach is sound but not complete. In fact, the authors show that no complete method exists for this class of programs.

To illustrate the considered program class, the paper gives the following example:

```
import random
def multipath(x,y,z):
    while x >= y:
        if random.choice([1,2]) == 1:
            x,y = x + 1, y + x              #tau_1
        else:
            x,y,z = x - z, y + z**2, z-1    #tau_2
    return x,y,z
```

Listing 9: Multipath polynomial program from [18], written in Python

This example shows that the program variables do not necessarily evolve in a simple linear way. Depending on the chosen transition, variables may increase or decrease according to polynomial expressions. As a result, simple linear ranking functions are often not sufficient to prove termination.

Instead of relying on ranking functions in the usual linear sense, the authors analyze the loop using finite differences. Finite differences have long been used in program analysis, for example in approaches that derive invariants, running times, or termination proofs from difference equations. In [18], however, they are used to study the qualitative behavior of expressions over transitions.

The key idea is to identify expressions whose values eventually only decrease during the execution of the loop while at the same time being bounded from below. If such an expression can be found, the loop cannot continue forever and termination follows.

This approach is particularly useful for programs in which variables change according to polynomial expressions, where linear ranking functions are often not sufficient to prove termination.

Non-termination analysis

Beyond just proving termination, there also exists the analysis of non-termination. Here we will follow the work of Gupta et al. [32]. It has become increasingly clear that the practical value of verification tools often lies more in discovering bugs than in proving full program correctness [30]. For software developers, a concrete bug is usually more useful than a correctness proof, since bugs can already be identified during development, whereas full correctness proofs are often only possible at a much later stage. This indicates that the precision and

efficiency of such tools should be geared towards finding bugs instead of finding proofs of termination.

In the context of the work of Gupta et al., the relevant liveness property is termination. A program satisfies this property if every execution eventually finishes. Consequently, a liveness violation is a non-terminating execution, that is, an execution that continues forever. Gupta et al. point out that much of the previous work in liveness verification has focused on proving termination. However, termination proving techniques are incomplete. Therefore, a failed attempt to prove termination does not immediately imply that a program is non-terminating and produces a possible counter example. Instead, it may only show that the chosen proof technique was not strong enough. For this reason, the authors argues that termination proving should be complemented by techniques that can actively demonstrate feasible non-terminating executions.[32]

To search for such counterexamples, the paper focuses on so called lassos. A lasso consists of two finite paths, called the stem and the loop. The stem is a finite path that leads to a loop, whereas the loop is a finite path that starts and ends at the same program location. The method is incomplete, since not every non-terminating execution can be represented in lasso form. In particular, some infinite executions may be non-periodic. Nevertheless, restricting the search to lassos allows the approach to detect many practically relevant non-termination bugs efficiently.[32]

The algorithm proceeds in two phases. In the first phase, candidate lassos are generated by systematically exploring the execution paths of the program until a control location is revisited. If the search needs to explore an alternative branch, it can backtrack and continue with a different execution choice. This has two important advantages. First, the stem followed by one execution of the loop is guaranteed to be feasible, since it originates from an actually explored program execution. Second, the search is exhaustive, meaning that every lasso of this form will eventually be generated.[32]

In the second phase, the algorithm checks whether a candidate lasso indeed induces a non-terminating execution. For this purpose, the paper introduces the notion of a recurrent set. Intuitively, a recurrent set is a reachable set of program states from which the loop can always return to the same set again. If such a set exists, then the loop can be executed indefinitely. Instead of searching for such sets manually, the method uses templates for recurrent sets and reduces their computation to a constraint solving problem. The authors further distinguishes between two levels of precision. The precision of the analysis depends on the chosen constraint theory. A bit-level analysis models machine specific effects such as overflow and therefore provides a very precise view of program behavior. In contrast, an arithmetic level analysis is more abstract and is aimed at detecting non-termination caused by the algorithm itself rather than by low-level implementation details.

An example used in the paper is a broken implementation of binary search, shown in Listing 10. To illustrate the bug, the paper considers the execution

path that follows the first branch of the conditional statement, that is,

$$\{lo < hi\};\ mid = (lo + hi)/2;\ \{a[mid] < k\};\ lo = mid + 1.$$

For the initial values

$$lo = 1, \qquad hi = MAXINT, \qquad a[0] < k,$$

the loop condition is satisfied. However, the calculation overflows and outputs mid = 0. As a result, the update sets lo = mid + 1 = 1, while hi remains unchanged. The program therefore returns to the same loop state as before, namely lo = 1 and hi = MAXINT. Since this cycle can be repeated indefinitely, the execution is non-terminating and has the shape of a lasso.

This example from [32] shows that non-termination detection complements termination proving methods and is essential for identifying concrete infinite executions that cannot be established through termination proofs alone.

```
int bsearch(int a[], int k,
            unsigned int lo,
            unsigned int hi) {
    unsigned int mid;
    while (lo < hi) {
        mid = (lo + hi) / 2;
        if (a[mid] < k) {
            lo = mid + 1;
        } else if (a[mid] > k) {
            hi = mid - 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

Listing 10: Broken binary search example adapted from [32] in C

3.3 Termination Analysis Tools

The approaches described so far are the foundations that termination tools build on. Which tools are actually available and actively developed is documented in the Termination Portal [46], which collects the tools, publications and events of the field as well as hosting the annual Termination Competition, where the tools are compared on a common benchmark set. We use the portal to identify the current state of the art and describe below the five tools that are later used in our Methodology.

AProVE

AProVE (Automated Program Verification Environment) is an automated verification tool for termination and complexity analysis for C, Java, Haskell and Prolog programs as well as term rewrite systems (TRS). To analyze the high level languages, AProVE translates the input program into a so called symbolic execution graph. To explain symbolic execution graphs we follow the works of Ströder et al. [43]. The symbolic execution graph gives an abstract view of all possible ways the input program can run. It is built through symbolic execution, so the program is analyzed using symbolic values instead of concrete inputs. Although it resembles a CFG, it is not one, since its nodes are abstract states rather than program positions, so the same position can occur in several nodes. During this construction, the tool also takes into account features that are specific to the programming language. In the case of C programs, this means handling aspects such as pointer arithmetic, memory allocation, pointer dereferencing and memory safety [43]. In order to analyze the program for termination the graph is turned into a TRS. Figure 3 serves as a graphical overview for how the tool works [28].

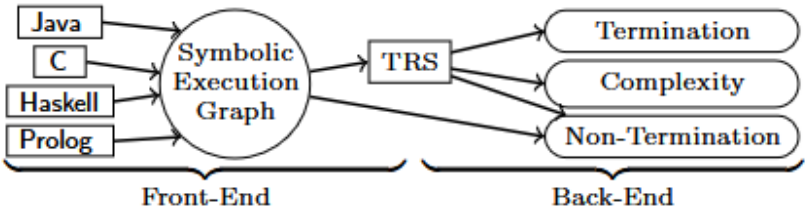

Figure 3: Graphical explanation of AProVE [28]

A TRS is a formal way of describing computation using rewrite rules. In this setting, program states or expressions are represented as terms, which are built from function symbols and variables. A rewrite rule then describes how one term can be replaced by another. For termination analysis, the main question is whether this rewriting process could go on forever. The dependency pair approach, which is implemented in AProVE, focuses on the parts of a TRS that may cause recursion or repeated function calls. In simple terms, a dependency pair represents a possible function call, while a chain of dependency pairs represents a possible sequence of such calls during a reduction. Thus, in the dependency pair framework, termination is shown by ruling out infinite dependency pair chains [29].

To make this more concrete, the paper introduces a small example right at

the start. It is the following TRS [29]:

$$\begin{aligned} \text{minus}(x, 0) &\to x \\ \text{minus}(s(x), s(y)) &\to \text{minus}(x, y) \\ \text{div}(0, s(y)) &\to 0 \\ \text{div}(s(x), s(y)) &\to s(\text{div}(\text{minus}(x, y), s(y))) \end{aligned}$$

In this kind of setting, natural numbers are written symbolically using the constant 0 and the successor function s, so $s(0)$ stands for one, $s(s(0))$ for two, and so on. The rules describe how terms are rewritten, with the right-hand side replacing the left-hand side whenever a match is found.

Before dependency pairs, termination was usually shown using simplification orders, which require the right-hand side of every rule to be structurally smaller than the left-hand side. Looking at the last div rule, this requirement clearly does not hold, as the right-hand side wraps the recursive div call inside an s, which already looks like its growing rather than shrinking. Even worse, as the paper points out, instantiating y with $s(x)$ causes the left-hand side to appear embedded inside the right-hand side, which is exactly the situation simplification orders are unable to handle. As a result, methods based on these orders fail on this TRS, even though it clearly terminates [29].

This is exactly the gap the dependency pair approach is designed to close. Instead of demanding that the entire right-hand side be smaller than the left, it focuses only on the recursive calls inside each rule. For the example above, the relevant dependency pairs are:

$$\begin{aligned} \text{MINUS}(s(x), s(y)) &\to \text{MINUS}(x, y) \\ \text{DIV}(s(x), s(y)) &\to \text{MINUS}(x, y) \\ \text{DIV}(s(x), s(y)) &\to \text{DIV}(\text{minus}(x, y), s(y)) \end{aligned}$$

The capital-letter symbols are tuple symbols, used to mark the function calls being tracked. The first and third pairs describe minus and div calling themselves, while the second one represents the call from div into minus. Once the surrounding s from the original rule is stripped away, the recursion is much easier to analyze. In the third pair, for example, the first argument changes from $s(x)$ to $\text{minus}(x, y)$, which is intuitively smaller. The problem that broke simplification orders simply does not appear here, because we are no longer looking at the rule as a whole.

T2

TERMINATOR is a termination prover developed by Cook, Podelski and Rybalchenko that introduced a new way of structuring termination proofs [23]. Instead of constructing a single global ranking function for an entire loop, TERMINATOR builds up a collection of many small ranking functions, each of which only needs to handle a single path through the program. These small functions are combined by disjunction, and a safety prover is used to check whether the collection together covers all possible program behaviors. If it does not, the

safety prover returns a counterexample path, from which a new small ranking function is synthesized and added to the collection. This process repeats until the safety prover finds no further counterexamples.

T2 (TERMINATOR 2) is an open source verification framework that directly builds upon and extends TERMINATOR [19]. It can process programs written in languages such as C, C++, and Objective C. Internally, T2 represents programs as graphs of program locations that are connected by guarded transition rules over integer variables. For termination analysis, T2 focuses on linear integer arithmetic fragments of C and applies termination as well as non-termination proving techniques on this graph-based representation. Figure 4 will show a graphical overview [19].

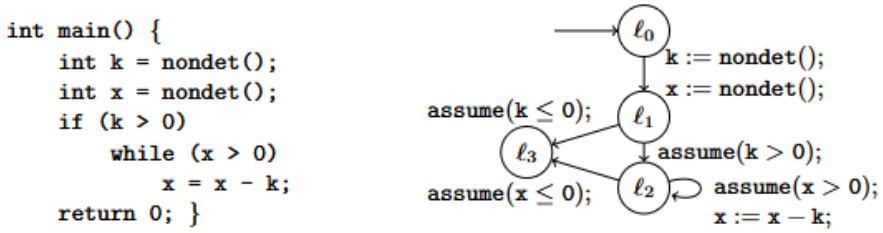

Figure 4: Graphical explanation of T2 [19]

KoAT

KoAT (Komplexitäts Analyse Tool) is an analysis tool for integer programs. Its main focus is the automatic inference of upper bounds on runtime and variable growth. To do so, it uses a modular approach in which individual parts of a program are analyzed separately before the results are combined into an overall bound for the complete program. To analyze C programs, they first need to be translated into an internal representation of integer programs, which is then used for the analysis.

In addition, KoAT is also used as a backend in newer versions of AProVE for the analysis of integer transition systems. The following Figure 5 serves as a bird's-eye view of this approach [42, 36]. We will quickly go over the various frameworks used in KoAT and why they are being used.

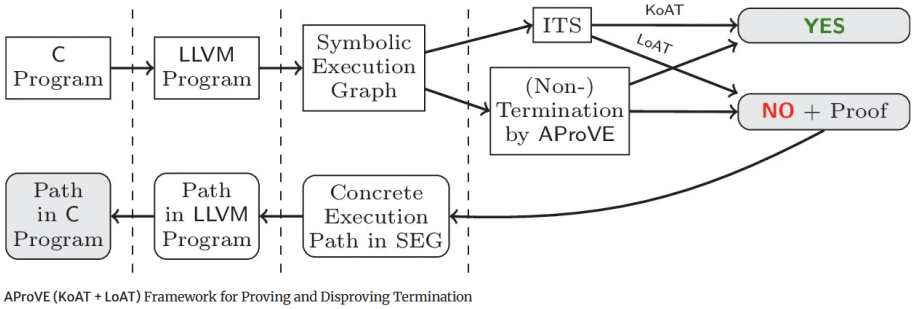

Figure 5: Graphical overview of KoAT [36]

For the explanation of a Low Level Virtual Machine (LLVM), we will take Lattner and Adve's explanation from [35]. LLVM is a compiler framework based on an intermediate program representation. Despite its name, it is not a virtual machine in the sense of a complete execution environment for programs. Rather, LLVM introduces a low level representation between the original source program and the final machine code. Programs, for example written in C or C++, can be translated into this representation, where they can then be analyzed and transformed before being compiled further.

The LLVM representation is designed to be close enough to machine code to represent low-level programs, but still rich enough to support program analysis. This makes LLVM useful as a common basis for compiler optimizations and analysis tools, since these tools can operate on LLVM's uniform intermediate representation instead of directly on the syntax of a particular source language.

In the LLVM based approach used by tools such as KoAT, LLVM can therefore be seen as an intermediate step before the actual analysis. Instead of analyzing the original C program directly, the program is first translated into LLVM's more uniform low-level representation. From this representation, the information needed for the subsequent analysis can be extracted more easily.

Before KoAT can analyze a program, it first translates it into an Integer Transition System (ITS), which serves as the internal representation on which the actual complexity analysis is performed. The need for such a representation becomes clear when looking at a concrete example from [20]. Consider the following two programs:

```
while i > 0 do
  i := i - 1;
  while x > 0 do
    x := x - 1
  done
done
```

```
while i > 0 do
  i := i - 1;
  x := x + i;
  while x > 0 do
    x := x - 1
  done
done
```

Figure 6: Two similar programs with different runtimes [20]

Both programs look structurally very similar and can even be proven terminating using the same lexicographic ranking function $\langle i, x \rangle$. However, their runtime complexities differ fundamentally. The program on the left runs in linear time, while the program on the right runs in quadratic time. The critical difference lies in the additional assignment $x := x + i$ inside the first loop of the right program. Because of this, x grows during the first loop, and since the second loop iterates x times, its runtime now depends on how large x has become by the time the first loop finishes. This kind of interaction between loops and variable sizes is not immediately obvious from reading the source code alone. By first translating the program into an ITS, KoAT can systematically track how variable values evolve across different parts of the program and combine these results to derive a precise upper bound on the overall runtime, which in this case turns out to be quadratic in the input size.

VeryMax

Another tool we will look into is VeryMax [16]. It is a verification framework that proves termination and non-termination. The tool can handle C and C++ programs which includes linear operation with integer variables as well as loop and condition statements by transforming them into LLVMs, which is ideal for our testing purposes. Besides all of that VeryMax can also handle ITS. It heavily depends on Max-SMT constraint solving for proving and disproving termination [34]. In order to understand Max-SMT solver, we will briefly explain SAT, MaxSAT and SMT first. The tool proves program termination by repeatedly finding conditions under which parts of the program terminate, removing those already solved cases, and focusing only on the remaining states where non-termination might still be possible [15].

To define SAT, we use the work of Gong and Zhou [31]. In computer science, the Boolean Satisfiability Problem (SAT) asks whether a given Boolean formula can be made true by assigning truth values to its variables. A SAT problem usually consists of a collection of Boolean variables and a set of clauses built from those variables. The goal is to find an assignment of values that satisfies all of the clauses.

SAT is a fundamental problem in areas such as mathematical logic, automated reasoning, machine learning, and many other theoretical and practical domains. It is historically significant because it was the first problem proven to

be NP-complete, and many other NP-complete problems can be related to it. As a result, even small improvements in SAT-solving algorithms can have an important impact on computer science.

To explain P and NP, we use the work of Cook [24] to describe both terms briefly. Informally, P is the class of decision problems that can be solved by an algorithm in a reasonable amount of time, where reasonable means that the number of steps grows at most polynomially with the size of the input.

The class NP stands for nondeterministic polynomial time. It was originally defined using nondeterministic machines, which are theoretical machines that can have several possible next moves from the same configuration. In simple terms, NP contains decision problems for which a proposed solution can be checked efficiently, even if finding that solution may be difficult.

Moving on to MaxSAT, we take inspiration from the work of Ansótegui et al. and explain it [13]. The MaxSAT problem extends the traditional satisfiability problem by considering cases where it may not be possible to satisfy all constraints. Instead of requiring every clause to be satisfied, the goal is to satisfy as many clauses as possible.

This idea can be extended further to the Weighted Partial MaxSAT problem. In this version, the constraints are divided into two categories, which are the hard clauses, which must always be satisfied and the soft clauses, which are desirable but not strictly required. Soft clauses can also be assigned different weights, where each weight represents the penalty incurred if that clause is not satisfied. This reflects the fact that some constraints are more important than others.

The use of weights makes the problem weighted, while the distinction between hard and soft clauses makes it partial. Therefore, in a weighted partial MaxSAT instance, the aim is to find an assignment that satisfies all hard clauses while minimizing the total weight of the soft clauses that are violated. An assignment that achieves this minimum penalty is considered optimal.

For many real world problems, translating everything in propositional logic is not always the best approach. In many cases, it is more useful to describe the problem using a richer non-propositional logic that takes some background Theory T into consideration. This is called a Satisfiability Modulo Theories (SMT) problem. Given a Formula F, SMT asks whether F is satisfiable with respect to the theory T. In other words it checks whether there is a model that satisfies both the background theory T and Formula F [38].

Now we turn back to [34] to explain MAX-SMT. What Max-SMT does is it combines the ideas behind Max-SAT and SMT. It extends SMT in the same way that MaxSAT extends SAT. Instead of requiring every constraint to be satisfied, Max-SMT allows some constraints to be violated. Each constraint is assigned a weight, which represents how costly or undesirable it is to violate that constraint. The goal is then to find a solution that satisfies the constraints as well as possible by minimizing the total cost of the constraints that are not satisfied.

Some constraints may have an infinite weight. These are treated as hard constraints, meaning they must always be satisfied. Constraints with finite

weights are soft constraints, meaning they should be satisfied if possible, but they may be violated if this leads to a better overall solution.

MuVal

The last tool we are going to be testing is called MuVal and is being developed by Unno et al. [49]. In this approach, the program and the property, that should be checked are encoded together as a formula in a fixpoint logic, called μCLP. Here, property means the specification or requirement that the program is supposed to satisfy.

The advantage of this is that the logic acts as a common intermediate language. Different verification problems can first be translated into this logical form, and then a solver can focus only on checking whether the resulting formula is valid. Fixpoint logic is a logic for describing recursive properties and it is used to express program behavior, especially loops and infinite executions. A least fixpoint describes the smallest behavior that satisfies a rule, while a greatest fixpoint describes the largest behavior that can keep satisfying a rule. [33]

MuVal follows this general idea. It targets a first order fixpoint logic with background theories, called μCLP. The predicates are interpreted using least and greatest fixpoints, allowing the logic to express both finite behaviors, such as termination, and infinite behaviors, such as non-termination. Instead of solving the μCLP validity problem directly, MuVal reduces it to a constraint satisfaction problem in pfwCSP, a predicate constraint language that generalizes constrained Horn clauses (CHC).

In order do understand CHC, we use the work of Bjørner et al. to explain them [14]. CHCs are Horn clauses extended with a background theory called an assertion language, for example integer linear arithmetic, but other theories are also possible. To understand what a Horn Clause is we refer to the work of Angluin et al. [12] and break it down. A Horn clause is a special type of clause where at most one literal is allowed to be unnegated and that restriction is what makes it equivalent to an implication since $\neg a \vee \neg b \vee \neg c \vee d$ says the same as $a \wedge b \wedge c \rightarrow d$. The negated literals therefore act as the condition and the single unnegated literal as the conclusion that follows from it. A clause itself is formed by taking a bunch of literals and joining them in a disjunction over some variable set V. A literal, at its most basic, is just a variable (like v) or its negated counterpart ($\neg v$).

Now that CHCs have been explained, pfwCSP are following. For the explanation we use the work of Unno et al. [50]. Essentially, pfwCSP extends CHCs by lifting the restriction that each clause may contain at most one positively occurring predicate variable, allowing arbitrary clauses that CHCs cannot express.

To illustrate how MuVal works in practice, Figure 7 shows the overall process. Given a μCLP validity problem, MuVal first constructs its logical opposite, called the dual instance. The primal instance asks if this program always terminate, while the dual asks if this program fails to terminate. Both instances are then independently converted into pfwCSP constraint sets. MuVal then runs two solvers in parallel, one for the primal and one for the dual, and lets them

exchange information with each other as they progress. This exchange allows the two solvers to mutually narrow down each others search space, which helps them converge to an answer faster. The process concludes as soon as one of the solvers finds its answer if the primal solver succeeds, the property is valid, and if the dual solver succeeds, it is invalid.

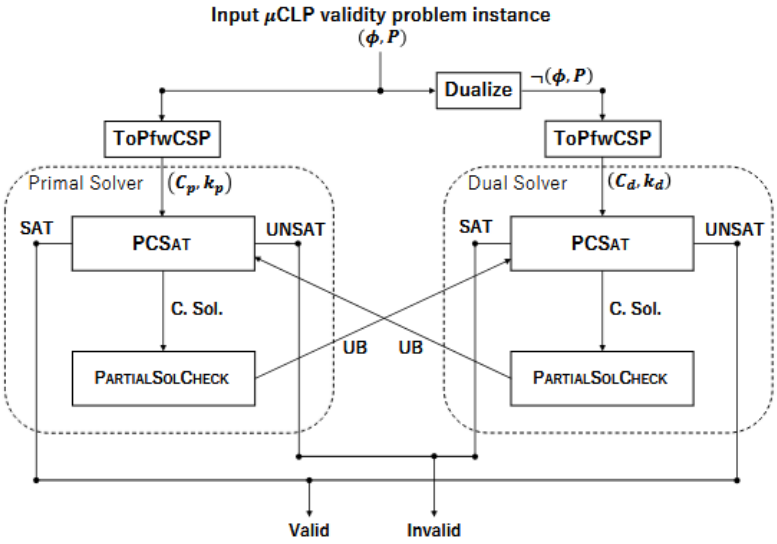

Figure 7: Overview of MuVal [49]

However, MuVal in its original form is not directly capable of analyzing C programs, as it lacks support for features commonly found in real world C code, such as pointers, arrays, data structures, and bitwise operations. To bridge this gap, Fathi et al. [26] developed Athena, a framework that serves as a front-end for MuVal. Athena takes a C program as input and transforms it into a form that MuVal can analyze.

Summary of Input Formats

The five tools do not share a common input language. Tow of them are driven from C, two from T2 transition system and format and one from an ITS in the KoAT format. Table 1 lists which formats each tool accepts and which of them are used.

Tool	Input formats accepted	Used here
AProVE	Java, C, Haskell, Prolog, TRS [28]	C
MuVal	μCLP, C through a front end [49, 26]	C
T2	native T2 format, C via LLVM [19]	T2
VeryMax	T2, C and C++ via LLVM [16]	T2
KoAT	integer programs in the TermCOMP ITS format, C via llvm2kittel [48]	*.koat*

Table 1: Input formats accepted by the five tools and the format used.

The C files are ordinary C programs. The loop is written out as code and the parameters are left open, with only the assumption that they are non-negative. The *.t2* and *.koat* files are not programs at all. They describe the same behavior as a transition system, meaning a set of location and the moves between them.

ARI is the standard format defined by TermCOMP 2025 [8]. The pipeline generates it as well but none of the tools runs it.

4 Methodology

This chapter describes the pipeline developed to investigate how existing termination analysis tools can be applied to functions extracted from Wikifunctions. Since the selected tools do not accept Python code directly, a multi-stage pipeline was developed to bridge the gap between the raw Wikifunctions data and the input formats each tool requires. The pipeline begins by extracting Python implementations from the Wikifunctions XML dump and categorizing them according to their structural properties, using abstract syntax tree analysis to determine which functions are suitable candidates for termination analysis. The functions that pass this step are then translated into the appropriate input formats for each tool. Once the translations are in place, the tools are executed and their outputs are collected and brought into a comparable form, since different tools report their results in different ways. The remainder of this chapter describes each of these stages in detail, followed by an explanation of how the collected results are used to answer the three research questions. Every step of the pipeline described in this chapter, together with the result tables reported in Section 5, is implemented in the notebook. Table 10 in Appendix A states where each of them can be found.

4.1 Dataset

The primary data source for this thesis is the official Wikifunctions XML dump dated May 1, 2026. Like other Wikimedia projects such as Wikipedia, Wikifunctions provides periodic exports of its entire content as structured XML files, made available through the Wikimedia database backup dump service [4]. The dump used in this thesis was obtained directly from this service and contains a total of 79,340 pages.

Before describing the extraction process, we briefly explain how content is structured within Wikifunctions. Every entity on the platform, whether it is a function, a type, an implementation, or a test case, is stored as a ZObject. A ZObject is essentially a typed JSON object and each persistent ZObject receives a unique identifier known as a ZID, which takes the form of the letter Z followed by a positive integer. Two types of ZObjects are particularly relevant for this thesis. The first is a function object, identified by the type Z8, which defines the signature of a computation including its name, arguments and return type. The second is an implementation object, identified by the type Z14, which contains the actual executable code for a function in a specific programming language. A single function can have multiple implementations in different languages. Keys within a ZObject follow the naming convention ZnKm, where n is the ZID of the defining type and m is the key index. For example, Z13667K1 refers to the first key of the first argument of the factorial function Z13667 [10].

Figure 8 illustrates this structure using the factorial function as a concrete example:

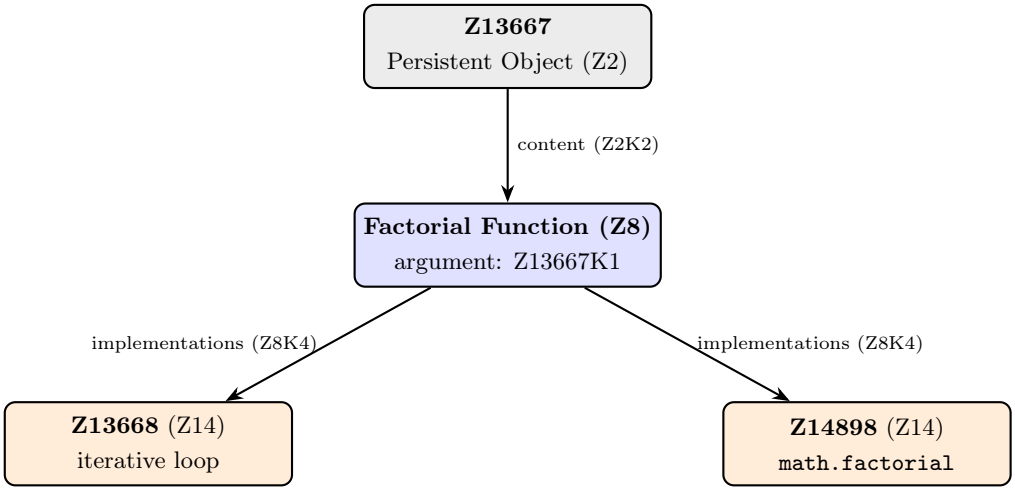

Figure 8: ZObject structure for the factorial function Z13667

We considered only Python implementations for our approach. Implementations in other languages such as JavaScript were excluded. Not every implementation on Wikifunctions carries source code, since more than half of them are compositions built from calls to other functions. Among those that do carry code, Python is the most common with 1,968 implementations, followed by JavaScript with 1,299. To extract the relevant data, the XML dump was processed page by page and each page was checked for the presence of a Z14 implementation containing Python code. When a match was found, the source code and the corresponding function ZID were stored for the next stage of the pipeline. Figure 9 illustrates this process and the results are summarized in Table 2.

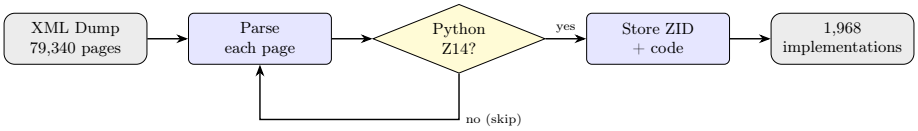

Figure 9: Dataset extraction pipeline. Each page in the XML dump is checked for a Python implementation

Summary	
Date of the dump	May 1, 2026
Total pages scanned	79,340
Functions	4,134
with ≥ 1 Python implementation	1,823
without any Python implementation	2,311
Python implementations found	1,968

Table 2: Summary of the Wikifunctions dataset extraction.

It is important to note that the 1,968 extracted implementations are not all suitable for the pipeline built here. Most of them contain no loop at all and terminate trivially and among those that do loop, the ones that iterate over strings, lists or floating point values cannot be expressed in the target formats, since the ITS formats have no way to represent such data. That restriction comes from the formats rather than from the tools and the C programs follow the same integer only style so that all five tools see the same functions. The analysis is therefore restricted to functions that loop over integer variables and the next section shows how few Wikifunctions functions take that form.

4.2 Categorization

Having extracted 1,968 Python implementations from the dump, the next question is which of them actually can be passed to a termination tool and which of them already terminate trivially, for example because they contain neither a loop nor recursion. The tools are evaluated here on input formats designed for programs that iterate over integer variables, so a function that just converts a string to uppercase or checks whether a number is even has nothing for a ranking function to latch onto. Running such functions through the translation and analysis pipeline would produce no useful output and waste considerable time. To avoid this, each implementation is first inspected statically and assigned to a structural category that determines how it is handled downstream.

The inspection is done by parsing each implementation into an Abstract Syntax Tree using Pythons built in *ast* module and then walking the tree to collect a set of flags about what the function contains whether it has a *while* loop or *for range* loop, whether it performs integer arithmetic, whether it includes strings, lists, or dictionaries, whether it uses recursion and whether it calls any functions not defined locally and not part of the standard Python built ins.

An *Abstract Syntax Tree* (AST) is a tree structured representation of the syntactic content of a program. Each node in the tree corresponds to one construct in the source code like a function definition, a loop, an assignment, an expression without retaining the raw text. Pythons built in *ast* module generates such a tree directly from source code and provides utilities for walking it node by node [41]. Crucially, parsing never executes the code as the analysis is entirely structural.

Figure 10 shows the AST produced for the factorial implementation Z13668, illustrating how each line of code decomposes into typed nodes.

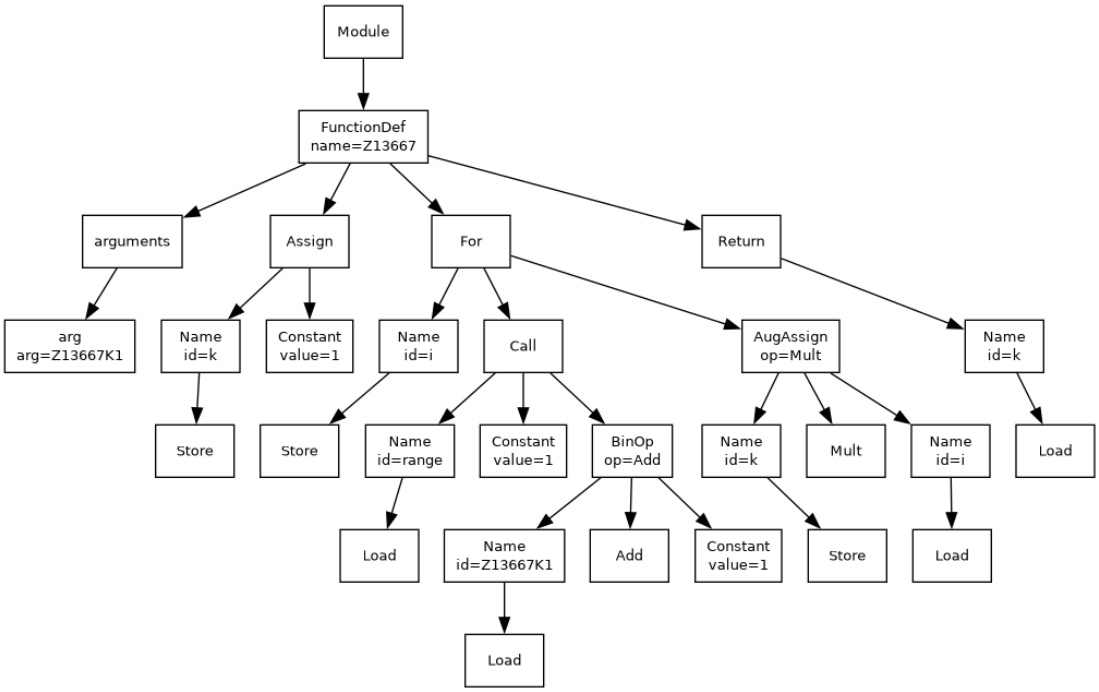

Figure 10: Abstract Syntax Tree of the factorial implementation Z13668, generated by Pythons `ast` module.

Importantly, the code is never executed and all analysis is purely structural, which makes the step both fast and safe regardless of what a function actually computes. For each implementation we record a set of flags. They cover whether the code parses at all and whether it contains a loop and if so what kind of loop it is and how deeply the loops are nested. We also record whether the function uses break or continue, whether it calls itself and whether it does integer arithmetic. Finally we note whether floats, strings, lists or dictionaries appear anywhere and whether the function calls something it does not define itself and that is not a Python built in. Each function is then assigned to one of the categories described below and summarized in Table 3.

Running the categorization over all 1,968 implementations produces the distribution shown in Table 3. The categories classify each implementation by its control flow and data structure. The decision about which categories we attempt to verify follows the program classes of the International Termination Competition as defined on the Termination Portal [7]. The faithfully translatable integer loops (INT_LOOP_TRANSLATABLE) are expressed in two target formats. They are written as C integer programs, which matches the C Integer

Programs category [3], and as integer transition systems, which matches the Integer Transition Systems category [5]. The same 43 functions are therefore represented in both formats. Recursive functions (Recursion) like the Ackermann Function in Figure 1 would instead require the Term Rewrite Systems pathway [9], or the native support for recursive C that AProVE offers [28]. Both would be possible in principle and Section 6 explains why we do not take either route.

Category	Description	Count	%
No_Loop	No loop and no recursion	1,435	72.9
Non_Integer_Loop	Loops over strings, lists, floats	288	14.6
Recursion	Uses recursion	156	8.0
Int_Loop_Translatable	Integer loops translated and tested	43	2.2
Int_Loop_Complex	Integer loops too complex to translate	26	1.3
Parse_Error	Does not parse	20	1.0
Total		1,968	100.0

Table 3: Category distribution of the Python implementations extracted from Wikifunctions.

The largest category by far is No_Loop, which contains nearly three quarters of all implementations. Functions without a loop and no recursion trivially terminate and offer nothing for a termination tool to analyze. Looking at what these functions actually do, this makes sense, as Wikifunctions is a collaborative repository of general purpose functions, not a collection of algorithms. A large proportion of its content consists of unit conversions and simple arithmetic checks, none of which need iteration [52].

Non_Integer_Loop captures all functions that do loop but not over integer variables. This includes `for` loops over strings, lists, or dictionaries as well as `while` loops whose progress depends on floating point conditions. The input formats used in this thesis only deal with integer arithmetic, so these functions cannot be expressed in them directly.

Recursion covers functions that use recursive calls rather than explicit loops. Translating recursive Python into an integer transition system requires a different strategy than the loop based translator implemented here and is therefore out of scope.

Parse_Error contains implementations that could not be parsed at all by Pythons *ast* module, typically due to Python 2 syntax such as `print` statements or `unicode` literals.

The remaining 69 functions contain at least one loop over integer variables. Of these, 43 were successfully translated into a faithful integer transition system and form the analysis set for this thesis. A translation counts as faithful when it produces the same results as the original Python function on every input it was tested on. How this is checked is described in the next section. The other 26 were excluded, as 21 contained integer loops that were too complex to translate faithfully and 5 used operations the target formats cannot express, such as the bitwise shift and exclusive or operators. Some of these functions call code that the translator cannot see, either a library function or another Wikifunctions function. The call cannot be written into the target format,

so they end up among the excluded functions as well. These 43 functions are labeled Int_Loop_Translatable and the 26 excluded functions are grouped together as Int_Loop_Complex in Table 3.

The most striking takeaway from this categorization is how small the set turns out to be with only 43 out of 1,968 functions (2.2%) could be translated into an integer program that accurately reflects the behavior of the original Python code.

4.3 Translation Pipeline

The 43 functions in Int_Loop_Translatable cannot simply be handed to a termination tool as they are. All tools in the evaluation expect either a C program or an integer transition system in a tool specific format none of them accept Python directly. Before any analysis can take place, each function therefore has to be translated into a representation the tools can actually work with.

The translation is built around a single intermediate representation. Each function is first converted into a CFG. From this single CFG, all tool specific formats are then generated. AProVE and MuVal receive a C program, KoAT receives a `.koat` file and both T2 and VeryMax receive a `.t2` file. This means that all five tools are evaluated on the same underlying functions, just in the format each tool was designed for. Since all five tools expect different input formats, the pipeline needs to produce several representations of the same function without translating each one independently. The solution is to build one shared intermediate representation first and then generate all formats from it. Figure 11 gives an overview of how this works.

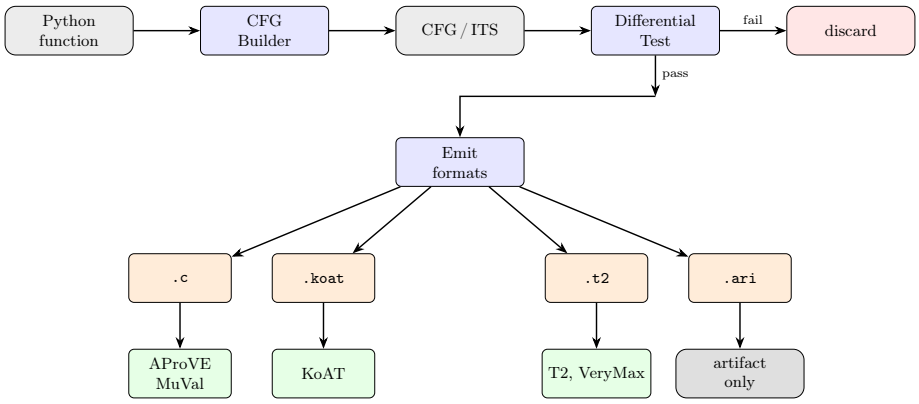

Figure 11: Translation pipeline overview.

The starting point is a CFG that captures the loop structure and variable behavior of the Python function. Once this representation exists, generating the tool specific formats is straightforward since they all describe the same

underlying program. Algorithm 1 describes how this conversion is carried out in practice.

Algorithm 1 Building a CFG from a Python function

Require: A Python function with one or more integer loops
Ensure: A set of locations and a set of transitions T over the variables V

1: Collect the integer variables of the function into V.
2: Create an entry location where the function begins and an exit location where it halts.
3: **for** each assignment, branch and loop of the function in order **do**
4: Create a new location for the point reached after it.
5: Add a transition τ to that location and express an assignment as the updates of τ, written with primed variables.
6: Add the condition of a branch or a loop as the guard of one transition and its negation as the guard of a second transition.
7: Lead a loop back to the location at which its guard is checked and a return to the exit location.
8: **end for**
9: Collect the transitions into T.

The algorithm is implemented as the function *build_cfg* in the notebook, together with the four emitters that write the C, T2, *.koat* and ARI Files. Table A maps every algorithm in this chapter onto the corresponding notebook function. The translation is deterministic. Each function is parsed into an AST, converted into a single CFG and written out by four separate emitters.

Translating a function into a CFG does not on its own guarantee that the translation is correct. A mistake in how the guard or the variable updates are extracted could produce a transition system that behaves differently from the original Python. To catch such errors, each translation is checked using a differential test before it is included in the analysis. Algorithm 2 describes this procedure.

Algorithm 2 Checking that a translation is faithful

Require: The original Python function and the CFG built from it
Ensure: A decision to keep or to discard the translation

1: Generate a set of test inputs, some random integers and some fixed small values.
2: **for** each input x **do**
3: Run the original Python function on x and record the result.
4: Run the CFG on x in an interpreter and record the result.
5: **if** the two results differ **then**
6: The translation does not match the original. Discard it.
7: **end if**
8: **end for**
9: Both versions agreed on every comparable input. Accept the translation.

Wikifunctions also stores its own test cases for many functions. Using their inputs here as well is discussed in Section 6. This algorithm is implemented as the function *difftest* in the notebook, see Appendix A.

To illustrate what a validated translation looks like in practice, Listings 11 and 12 show the C and T2 translations produced for the factorial function Z13668. This function passed the differential test and is part of the 43 functions that form the analysis set. The C translation represents each CFG location as a labeled block connected by *goto* statements, with the loop guard expressed as an *if* condition and the input modeled as a nondeterministic integer bounded by an *assume* call. The T2 translation expresses the same structure as explicit $FROM/TO$ transition blocks containing the guard and variable updates.

```
extern int __VERIFIER_nondet_int(void);
extern void __VERIFIER_assume(int);
int main() {
  long Z13667K1 = __VERIFIER_nondet_int();
  __VERIFIER_assume(Z13667K1 >= 0);
  long k = 0;
  long i = 0;
  goto Lloc2;
Lloc1: ;
  return 0;
Lloc2: ;
  k = 1;
  goto Lloc3;
Lloc3: ;
  i = 1;
  goto Lloc5;
Lloc4: ;
  goto Lloc1;
Lloc5: ;
  goto Lloc6;
Lloc6: ;
  if ((i < ((Z13667K1) + (1)))) {
    goto Lloc7;
  }
  if ((!((i < ((Z13667K1) + (1)))))) {
    goto Lloc4;
  }
  return 0;
Lloc7: ;
  k = ((k) * (i));
  goto Lloc9;
Lloc8: ;
  i = ((i) + (1));
  goto Lloc6;
Lloc9: ;
  goto Lloc8;
Lloc10: ;
  goto Lloc1;
}
```

Listing 11: C translation of the factorial function Z13668, produced by the CFG builder.

```
START: 0;

FROM: 0;
TO: 2;

FROM: 2;
k := 1;
TO: 3;

FROM: 3;
i := 1;
TO: 5;

FROM: 5;
TO: 6;

FROM: 7;
k := ((k) * (i));
TO: 9;

FROM: 9;
TO: 8;

FROM: 8;
i := ((i) + (1));
TO: 6;

FROM: 6;
assume((i < ((Z13667K1) + (1))));
TO: 7;

FROM: 6;
assume((!((i < ((Z13667K1) + (1))))));
TO: 4;

FROM: 4;
TO: 1;

FROM: 10;
TO: 1;
```

Listing 12: T2 translation of the factorial function Z13668, produced by the CFG builder.

Before a translated function is included in the analysis, it is checked with a differential test. The original Python function and the CFG built from it are run on the same inputs by a small interpreter, so that the two results can be compared. The inputs are ten randomly generated vectors with values between 0 and 9, produced from a fixed seed so the test can be reproduced, together with a set of structured inputs where each of 0 to 8, together with -1 and -2, is given to every parameter. Both sides run under a budget of 40,000 steps. If they ever disagree, either because they return different values or because one halts while the other does not, the function is thrown out. If both sides run past the step budget on the same input, that counts as agreement, so a translation is also accepted when it reproduces the originals non-terminating behavior. Only functions that pass are kept. This gives strong practical evidence that the 43 translations behave like the original Python code, though it is not a full proof of equivalence, since as discussed in Section 6 the two are only compared on finitely many inputs Tools that compile Python to C exist, but they would only cover the C translation but not the ITS translation. The ITS Termination Tools would then no longer be compared on the same program and since comparing the tools is one of the research questions, all four formats are generated from a single CFG instead.

The 2025 Termination and Complexity Competition [8] introduced ARI [6] as a standard exchange format for integer transition systems, and the pipeline also produces *.ari* files as an artifact following the ARI format. Driving the tools from ARI through the official *its conversion* [45] converter was attempted but turned out not to be feasible. For these reasons the tools are driven via their supported formats instead, and this is discussed further in the Limitation Section 6.

4.4 Tool Execution

Once all formats have been generated, the five tools are run on their respective input files. Each tool is available as a Docker image published on the official *termcomp* DockerHub repository [2], which contains the tool versions submitted to the 2025 Termination and Complexity Competition [8]. Using these images ensures that the tools reflect their current competition state and that the execution environment is fully reproducible. A timeout of 60 seconds is applied per function to prevent any single analysis from running indefinitely. Table 4 gives an overview of the tools and the format each one receives.

Tool	Input format	Timeout (s)
AProVE	`.c`	60
MuVal	`.c`	60
KoAT	`.koat`	60
T2	`.t2`	60
VeryMax	`.t2`	60

Table 4: Tools used in the evaluation with their input format and timeout per function.

Rather than calling each tool once per function, all 43 input files for a given tool are processed in a single container run. Inside the container, a shell loop iterates over the files one by one, runs the tool on each with the 60 second timeout and writes the output to a shared buffer. The combined output is then split back into per function sections and stored, so that each section can be mapped onto one of the five common verdict categories introduced below. This batched approach avoids the overhead of starting a new container for every file. Algorithm 3 gives an overview of this process.

Algorithm 3 Batched tool execution

Require: A tool, its Docker image, and the set of input files
Ensure: A verdict for each input file

1: Start a single Docker container for the tool.
2: **for** each input file f **do**
3: Run the tool on f inside the container with a 60 second timeout.
4: Capture the output and mark it with the filename so it can be identified later.
5: **end for**
6: Split the combined output back into individual per-file sections.
7: **for** each per-file output **do**
8: Parse the output and map it to one of the five verdict categories.
9: Store the verdict together with the function identifier.
10: **end for**

This algorithm is implemented as the function *run_ batch_ tier* in the notebook, see Appendix A. Each tool reports its findings in its own way, so before any comparison can be made, the raw output of each run is translated into one of five common verdict categories:

- *terminating*: the tool successfully proved that a function terminates.
- *non-terminating*: the tool found evidence that a function does not terminate.
- *unknown*: the tool finished but could not reach a conclusion either way.
- *timeout*: the tool was still running after 60 seconds and was stopped.

- *tool_error*: the tool crashed or produced output that could not be read.

The specific keywords used to detect each verdict differ per tool, as AProVE and MuVal output *YES* or *NO*, T2 prints whether a termination or non-termination proof succeeded, KoAT outputs a complexity bound, and VeryMax states whether the program terminates in plain text. Mapping everything to the same five categories is what makes it possible to compare all five tools fairly on the same set of functions.

To make the verdict classification more concrete, the following listings show the actual output produced by two of the tools when analyzing the factorial function Z13668, which serves as the running example throughout this thesis.

Listing 13 shows the proof chain AProVE produces for Z13668 in the C representation. AProVE proves termination by transforming the program into different problems until one of them can be decided. In this case, the output lists the chain as follows. The C program becomes an LLVM program, then a symbolic execution graph and finally it becomes a set of arithmetic rules. At the last step it prints out a *YES*, indicating that it found a proof of the program terminating.

```
YES
proof of /benchmarks/Z13668.c

Termination of the given C Problem could be proven:

(0) C Problem
(1) CToLLVMProof [EQUIVALENT, 214 ms]
(2) LLVM problem
(3) LLVMToTerminationGraphProof [EQUIVALENT, 3128 ms]
(4) LLVM Symbolic Execution Graph
(5) SymbolicExecutionGraphToSCCProof [SOUND, 0 ms]
(6) LLVM Symbolic Execution SCC
(7) SCC2IRS [SOUND, 203 ms]
(8) IntTRS
(9) IntTRSCompressionProof [EQUIVALENT, 0 ms]
(10) IntTRS
(11) PolynomialOrderProcessor [EQUIVALENT, 7 ms]
(12) YES
```

Listing 13: AProVE output for Z13668 (factorial, C input)

VeryMax works on the ITS representation and proves termination by finding a ranking function for each loop in the program. For Z13668, it finds the ranking function $-i + Z13667K1$, which is simply the distance between the current loop counter and the upper bound. This value decreases by one in every iteration and reaches zero when the loop ends, as shown in Listing 14. At the end of the output it reports that the program terminates.

```
Checking conditional termination of SCC {l6}...

LOG: CALL solveLinear

LOG: RETURN solveLinear - Elapsed time: 0.000651s
Ranking function: -_t2_i + _t2_Z13667K1
New Graphs:
Proving termination of subgraph 2
Analyzing SCC {l1}...
No cycles found.

Program Terminates
```

Listing 14: VeryMax output for Z13668 (factorial, T2 input)

Both tools therefore reach the same verdict for Z13668, even though they read different input formats and use different methods to get there.

4.5 Evaluation Design

The evaluation is built around the 43 translated functions that passed the differential test. All five tools are run on every function and for each run the raw output is mapped to one of the five verdict categories introduced in the previous section. The three research questions are then answered from this result set. Table 5 gives an overview of how each question is implemented and Table 10 in Appendix A states where each of them is computed.

Research question	Metric / operationalization
RQ1 Tool applicability	A tool is considered applicable if it can be called through its Docker container, accepts the translated input without failing at the parsing stage, and produces output that maps to one of the five verdict categories. Any obstacles encountered during input preparation are documented as part of this assessment.
RQ2 Coverage	Per tool count across the five verdict categories (terminating, non-terminating, unknown, timeout, tool error) over the 43 functions. Overall coverage is the number of functions for which at least one tool reaches a conclusive (terminating or non-terminating) verdict.
RQ3 Tool comparison	Comparison of the tools on the same 43 functions along several dimensions, which are: how many functions each decides, how much their verdicts overlap, pairwise agreement where both tools decide, and a qualitative comparison of how the tools fail. Contradictory verdicts, if any, are examined individually.

Table 5: Operationalization of the three research questions.

For RQ1 the goal was to find out whether each tool could actually be set up, called and have it produce useful output for the translated functions. One obstacle already worth noting here is the ARI exchange format. Even though the pipeline produces *.ari* files following the ARI format, driving the tools from them through the official *its-conversion* converter [45] was attempted but abandoned, as the converter did not handle the integer division and modulo in these files. As a result, no tool was driven from ARI and each tool was instead run on its supported format. Further obstacles specific to individual tools are discussed in Section 5.

For RQ2 coverage is looked at from two angles. On the per tool level, each tools verdicts are counted across the five categories. On the overall level the focus is on how many of the 43 functions receive a conclusive verdict from at least one tool, regardless of which tool provided it.

For RQ3 all five tools are compared directly on the same 43 functions. The comparison considers how many functions each tool decides, how much their verdicts overlap, how often each pair of tools agrees where both reach a verdict and how the tools differ qualitatively in the way they fail. Agreement is measured pairwise. Each of the ten possible pairs of tools is compared only on the functions where both tools reached a conclusive verdict. Any case where one tool proves termination and another proves non-termination for the same function would stand out as a contradiction and be examined individually, however in this evaluation no such case occurred.

5 Results

A first observation is that integrating and running the tools was largely unproblematic, as all five could be configured and executed on every function. Whether a run yielded a usable result, however, varied considerably. Of the 215 runs, a substantial proportion returned no conclusive answer. KoAT rejected the majority of its inputs already during parsing, VeryMax rejected most of its inputs at its front end, and AProVE mostly returned no output at all. Despite these failures, 32 of the 43 functions were proven terminating by at least one tool, with MuVal alone accounting for 29 of them, making it the strongest performer in this evaluation.

A more notable finding is that the tools never contradicted one another. Wherever two tools both produced a definite result for the same function, those results agreed. Across all five tools and all 43 functions, no case arose in which one tool reported termination and another non-termination. Only a single non-terminating result was produced in the entire evaluation, which was by MuVal on function Z14962. No other tool produced a verdict for this function, but manual inspection confirms the result that the implementation loops forever on input 0, because 0 is divisible by every i and the loop counter grows without bound. The termination analysis thus uncovered a genuine defect in a live Wikifunctions function, which the four other tools missed only because they failed on the input, not because they disagreed. The tools thus differ less in the correctness of their results than in their coverage, that is, how many functions each is able to decide and in the manner in which they fail when they cannot. These two dimensions, coverage and failure behavior, are examined more closely in RQ2 and RQ3.

5.1 RQ1 - Tool Applicability

The aim of RQ1 was to establish whether each tool could be set up, invoked and made to read the translated inputs and return an output. A tool is treated as applicable to a given function if it can be called through its Docker container, accepts the translated input without failing at the parsing stage and produces output that can be mapped to one of the five verdict categories defined in Section 4.4.

At the level of integration, all five tools could be configured and called. Reading the translated inputs, however, succeeded to very different degrees and this is where the main applicability differences appear. AProVE and MuVal, the latter after the fix described below, accepted all 43 *.c* inputs. T2 accepted 41 of 43, rejecting two with parse errors. VeryMax rejected the same two functions with a sort error and aborted a further 27 with a semantic error, so it fully processed only 14 of the 43. The semantic error is what VeryMax prints when a division or modulo does not have a constant divisor. KoAT, by contrast, could parse only 6 of its 43 *.koat* inputs, as its front end rejected the remaining 37 with parsing errors. Table 6 summarizes this.

Tool	**Supported format**	**Applicability notes**
AProVE	*.c*	All 43 inputs accepted, 21 runs produced no usable output.
MuVal	*.c*	Applicable after adding the declaration in Listing 15, all 43 inputs accepted thereafter.
KoAT	*.koat*	Only 6 of 43 inputs parsed, 37 rejected.
T2	*.t2*	41 of 43 accepted, 2 parse errors.
VeryMax	*.t2*	14 of 43 fully processed, 2 rejected with a sort error, 27 aborted with a semantic error.

Table 6: Tool applicability across the 43 faithfully translated functions.

Three practical obstacles came up during input preparation and are worth documenting.

The first concerns the ARI exchange format. The 2025 TermCOMP standard specifies ARI as the common intermediate format from which tools should be driven via the official *its-conversion* converter [45]. The pipeline produces *.ari* files as an artifact following the ARI format, so this route was attempted first, but it did not work in practice. The converter aborted with an unhandled exception while parsing the files and it provided no output target for T2 at all. Notably, integer division and modulo caused KoAT's native parser to reject most of its inputs and VeryMax's front end to abort on most of its inputs with a semantic error, so the difficulty is not specific to the converter but to how these tools handle non-linear integer operators in general. A way of avoiding these operators altogether is discussed in Section 6. As a result, no tool was

driven from ARI. Instead, each tool was run on its supported format generated directly from the same validated CFG, so the comparison still rests on the same underlying functions. The *.ari* files are kept as an artifact but play no role in the analysis.

The second obstacle concerned MuVal. The C translator adds a *__VERIFIER_assume(p >= 0)* guard for each parameter *p*, constraining the analysis to non-negative integer inputs. Initially these calls were emitted without a corresponding declaration at the top of the file. MuVal's front end rejects implicit declarations and on encountering the undeclared call it printed *Unsuitable instructions detected* and returned no verdict, so MuVal initially produced no result on any of the 43 functions despite the translations being otherwise correct. The fix was a single line added to the C template, shown in Listing 15:

```
extern void __VERIFIER_assume(int);
```

Listing 15: Declaration required by MuVal's LLVM frontend.

With the declaration in place, MuVal returned a conclusive verdict on 30 of the 43 functions (29 terminating, 1 non-terminating) and produced no verdict on the remaining 13.

The third obstacle affected T2 and VeryMax on the two functions mentioned above. Their *.t2* encodings contained an expression, which T2 reported as a parse error and VeryMax as a sort error. These two functions are therefore not interpretable by either *.t2* based tool.

Taken together, these observations show that applicability is not simply a question of whether a format is correct in principle, but of whether the generated code meets the specific expectations of each tools front end. This is most visible for KoAT and VeryMax, whose front ends accept only a small fraction of the inputs, rejecting the non-linear operators these functions contain.

5.2 RQ2 - Coverage

Combining all five tools, 33 of the 43 functions receive a conclusive verdict, that is, a terminating or non-terminating result from at least one tool. Of these, 32 are proven terminating and one function (Z14962) receives a non-terminating verdict. The remaining ten functions are left undecided by every tool, producing only genuine "unknown" results, timeouts, or tool errors across all five. Complete coverage of the benchmark is therefore not achieved, even though the combined tools still decide the large majority of functions. Table 7 and Figure 12 give the per tool breakdown using the five verdict categories.

Tool	Format	Term.	Non-term.	Unknown	Timeout	Tool err.
MuVal	`.c`	29	1	0	0	13
AProVE	`.c`	15	0	7	0	21
T2	`.t2`	11	0	30	0	2
VeryMax	`.t2`	11	0	2	1	29
KoAT	`.koat`	6	0	0	0	37

Table 7: Per tool verdicts across the 43 translated functions, using the five verdict categories.

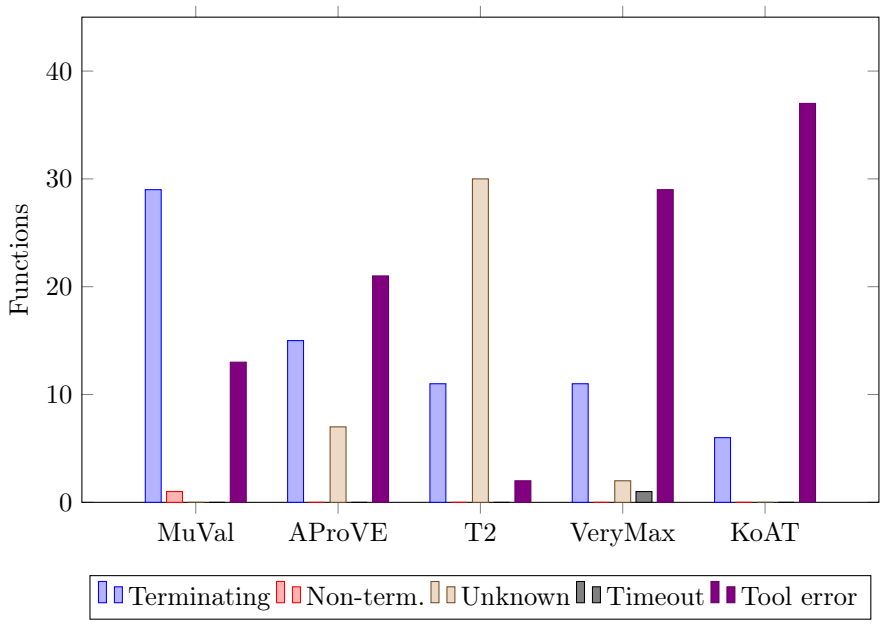

Figure 12: Verdict distribution per tool over the 43 functions.

The strongest individual tool is MuVal, which proves 29 of the 43 functions terminating. This is a noteworthy result, because MuVal operates on the C translation rather than on a dedicated transition system format, yet it outperforms all three ITS based tools, including T2, which is a dedicated termination prover for integer transition systems. AProVE follows with 15, then T2 and VeryMax with 11 each, and KoAT with 6.

T2's relatively modest score is explained by the kind of arithmetic in the benchmark rather than by any setup issue. T2 searches for linear ranking functions and what limits it here is integer division and modulo. Every one of its 30 inconclusive results involves one of these two operators, although 5 of the 11 functions it does prove contain them as well. Division and modulo can be rewritten as loops using only addition, subtraction and comparison, so the difficulty

lies in how the translation encodes them rather than the arithmetic itself, as discussed further in Section 6. Since many Wikifunctions implement arithmetic such as greatest common divisor or divisibility checks, these two operators are frequent, which limits how many functions T2 can handle.

KoAT's score of 6 is not primarily a question of proving power. As discussed in Section 5.1, its parser rejected 37 of the 43 inputs, so those functions were never analyzed. Of the six inputs it could parse, KoAT proved all six terminating. Its low coverage is therefore an applicability limitation rather than an analysis one.

A useful control comes from comparing T2 and VeryMax directly, since both read the same `.t2` files, so any difference in outcome is attributable to the tool rather than the format. The two tools prove exactly the same number of functions terminating, eleven each. What differs is their behavior on the functions they cannot prove. T2 parses non-linear operators inside guards and reports an honest "Termination/nontermination proof failed" when its proof strategies are exhausted, which we classify as an unknown. VeryMax rejects division and modulo with a variable denominator outright before any analysis took place. The format controlled comparison thus shows comparable proving power, but VeryMax's low coverage is, like KoAT's, primarily an applicability limitation rather than an analysis one, a point taken up again in Section 5.3.

Non-termination plays only a marginal role. Exactly one function, Z14962, receives a non-terminating verdict and only from MuVal. No other tool returns a verdict for this function, so the result cannot be independently confirmed by another tool, but manual inspection of the Python source confirms it that the implementation loops forever on input 0. The other tools produced no verdict on this function because they failed on the input and not because they disagreed.

Finally, no single tool covers the whole benchmark. Even the strongest, MuVal gives no verdict on 13 of the 43 functions. Coverage improves substantially when tools are combined, rising to 33 of 43, but ten functions remain undecided by all five. This supports a portfolio approach, in which several tools are run together, while also showing that even a portfolio of five current tools does not achieve complete coverage on this benchmark.

5.3 RQ3 - Comparison of Tools

RQ3 compares the tools against one another on the same 43 functions, both quantitatively, in terms of how many functions each decides, how much their results overlap and how often they agree, as well as qualitatively, in terms of how they behave when they cannot decide a function. The per-tool figures underlying this comparison are those reported in Section 5.2.

Quantitatively, the tools span a wide range. MuVal decides the most functions (29 terminating, plus the single non-terminating verdict), followed by AProVE with 15, then T2 and VeryMax with 11 each, and KoAT with 6. The two strongest tools are both C based, which is notable given that three of the five tools operate on dedicated transition system formats.

The tools are strongly complementary rather than redundant. Table 8 shows how many of the 43 functions are decided by exactly n tools. Only six functions are decided by all five tools, whereas 20 are decided by a single tool and ten by none. Most of the combined coverage reported in Section 5.2 therefore comes from different tools deciding different functions, which is the main quantitative argument for running the tools as a portfolio.

Decided by	**Number of functions**
No tools	10
1 tool	20
2 tools	2
3 tools	1
4 tools	4
5 tools	6

Table 8: How many of the 43 functions receive a definite verdict from exactly n tools.

A clear pattern emerges when the tools are grouped by input category. The two C based tools, AProVE and MuVal together prove 32 functions terminating, while the three ITS based tools, T2, KoAT and VeryMax together prove only 12. Moreover, every function proven terminating by an ITS tool is also proven by a C tool. The ITS tools prove nothing that the C tools miss, whereas the C tools prove 20 functions that no ITS tool decides. On this benchmark the C based route therefore strictly dominates the ITS based one. This gap is not purely a matter of proving strength, however. Part of it stems from applicability and setup, since KoATs parser rejected most of its inputs and VeryMax's front end aborted on most of its inputs (Section 5.1) and part may stem from the C tools analyzing a more constrained input domain of non-negative integers.

A cleaner tool versus tool comparison is possible for T2 and VeryMax, which read the identical *.t2* files, so any difference between them is due to the tool rather than the format. The two prove exactly the same number of functions terminating and they fail on almost exactly the same functions, as 29 of T2's 30 inconclusive cases are functions on which VeryMax also fails. What differs is how that failure comes about as T2 parses the non-linear operators and gives up during proof search, whereas VeryMax rejects them at its front end before any search begins. For these two tools, differences in outcome are therefore driven by how each handles non-linear arithmetic, not by the input format.

Where the tools do reach conclusions on the same functions, they never disagree. Agreement was measured pairwise, over the functions on which both tools in a pair returned a definite verdict. Genuine unknowns, timeouts and tool errors were treated as no comparable verdict. Across all ten pairs the agreement is 100%, with no contradiction anywhere in the benchmark (Table 9). Two caveats apply, as the number of jointly decided functions per pair is small, between 6 and 12 and since almost every definite verdict is "terminating", this is best read as a consistency result showing the absence of contradictions rather

than as strong mutual confirmation.

Tool pair	**Jointly decided**	**Agreement**
MuVal–AProVE	12	100%
AProVE–T2	11	100%
AProVE–VeryMax	11	100%
MuVal–VeryMax	11	100%
MuVal–T2	10	100%
T2–VeryMax	10	100%
MuVal–KoAT	6	100%
AProVE–KoAT	6	100%
T2–KoAT	6	100%
VeryMax–KoAT	6	100%

Table 9: Pairwise agreement over functions decided by both tools.

Qualitatively, the tools differ most in how they fail. On the hard, non-linear functions that make up most of the undecided cases, each tool fails in a characteristic way, for example T2 exhausts its linear ranking-function strategies and gives up, VeryMax aborts with a semantic error on non-linear updates, AProVE returns no output or runs out of memory, MuVal produces no output, and KoAT never gets past its parser. The tools thus fail on a largely shared set of difficult functions but express that failure very differently. Taken together, the comparison shows that the tools agree wherever they overlap, differ mainly in coverage and failure behavior rather than in correctness and that on this benchmark the C-based tools clearly outperform the ITS-based ones.

6 Limitations and Future Work

Several limitations constrain how far the results of this thesis can be generalized and they also point towards natural directions for future work.

The most important limitation concerns the scope of the study. Only functions that take the form of a plain integer loop were analyzed. Of the 1,968 Python implementations extracted from the dump, 1,435 contain neither a loop nor recursion of their own and therefore terminate trivially, which leaves 533 further implementations. Of these, 288 loop over strings, lists or floating point values, 156 are recursive and 20 could not be parsed, leaving only 69 candidates of which only 43 passed both the categorization and the differential test and could therefore be handed to the tools. These 43 are implementations rather than distinct functions and they correspond to 40 functions, since two functions contributed more than one implementation. Recursive functions and functions that operate on strings, lists, or floating point values were excluded, because the pipeline only translates integer loops, which it encodes as C programs and integer transition systems. How far this restriction can be relaxed, in particular for recursive functions, is taken up among the directions for future work below.

A second limitation concerns the faithfulness of the translations. Whether a translation behaves like its original Python function was checked with a differential test, which runs both versions on a set of randomly generated and systematically chosen inputs and keeps the translation only if they agree. That gives good practical confidence, but it is not a proof that the two are semantically the same, so strictly speaking a tool's verdict applies to the translated program rather than directly to the original Python function. Z14962 marks the boundary of this check. It fails to terminate only on the input 0 and on that input both the Python original and the translation exceeded the test's step budget, which the test counts as agreement.

In this case that judgement was correct, as manual inspection confirmed. It does show, however, that for non-terminating behavior the test can only observe that both versions ran past the step budget, which is weaker evidence of agreement than comparing actual return values and the test still says nothing about inputs that were never tried. For a function that terminates on no input at all, such as the loop in Listing 1, the check loses its power completely, because every input exceeds the budget on both sides and any endlessly looping translation is accepted regardless of what it actually computes. No such implementation occurs in the benchmark and Z14962 is still compared by return value on every input other than 0. Wikifunctions stores test cases for its functions with curated inputs together with expected outputs for the Python implementations. Their inputs were not used here, although they could be integrated into the differential test as an additional source of inputs, which is taken up as a direction for future work. In addition, no existing tool translates Python directly into the formats the termination tools expect, so a custom translator was implemented for this thesis and Anthropic's LLM Claude [1] assisted in writing that code. The differential test is therefore the translations only safeguard to the best of our knowledge. Finally, the 2025 TermCOMP standard defines ARI as a com-

mon intermediate format but the official converter could not handle the integer division and modulo present in many of the functions, so each tool had to be run on its own native format instead.

A further limitation concerns the input domain. The C translations constrain every parameter to non-negative integers, while the *.t2* and *.koat* translations carry no precondition. A terminating verdict from a C based tool is therefore a proof for non-negative inputs only, whereas the transition system based tools must prove termination over all integers, which is a strictly harder obligation. All arguments of these functions are declared as natural numbers on Wikifunctions, so negative inputs are outside what they accept. Python and C also agree on integer division and modulo only for non-negative values, since Python rounds a division down while C rounds it towards zero, so the restriction is what keeps the translation faithful.

A fourth limitation concerns the execution environment. The tools were run in the same setup used by the annual Termination Competition, as Docker containers, with a 60 second time limit per function. This keeps the evaluation consistent with how the tools are normally assessed, but it also means the results reflect that particular setup.

A final limitation concerns what the results can claim about non-termination. Only one implementation received a non-terminating verdict from a tool, Z14962, which fails to terminate on input 0. MuVal proved that this implementation does not terminate. The differential test cannot confirm that on its own, since on the input 0 it only sees that both versions ran past the step budget, so we checked the source code manually. The ten functions that no tool decided were inspected in the same way. Seven of them terminate, because every iteration either grows the divisor or shrinks the number. Two of them, Z14065 and Z14069, compute Collatz stopping times, for which termination is a long standing open problem [44]. The last one, Z26991, does not terminate either, since it loops forever when its first argument is 0 or its second argument is 1. Two implementations are therefore known not to terminate on every input and the tools found one of them. For Z14065 and Z14069 the question stays open, since it depends on the Collatz conjecture. The ability of the tools to detect non-termination was therefore exercised, but only by a single function, so no general conclusion about their non-termination capabilities can be drawn from this benchmark.

The perfect agreement between the tools should still be read with some care. It was only measured on the functions that both tools in a pair actually decided and those overlaps are small, at most twelve functions for any pair. On top of that, almost every verdict in the study is a terminating one, so when two tools agree it usually just means both of them found the function to terminate. So while it is reassuring that no tool ever contradicted another, this shows more that they do not disagree than that they strongly confirm each other.

These limitations suggest several directions for future work.

The most direct extension is to the classes of functions that were excluded here, namely recursive and non-numeric functions. This is not primarily a limitation of the tools, but of the translation the pipeline performs. The pipeline builds its integer transition systems and C program out of the loops it finds in

the code. A recursive function repeats through its own call instead of through a loop, so there is nothing for the translation to pick up. The Ackermann function shown in Figure 1 is a good example, since it can be proven to terminate but only when it is written in a form that can handle recursion, which the loop based translation used in this thesis is not able to produce. Some recursive functions, such as tail recursive ones, can be easily turned into an equivalent loop and would then fit the format the pipeline already handles. Adding such a rewriting before the translation step would extend the set of target functions without changing the translation itself. A future extension could therefore translate recursive functions into term rewriting systems, or make use of the native support that tools such as AProVE have for recursive C, rather than into the integer transition systems used here. We do not do this because every target format is generated from a single CFG so that all five tools are compared on the same representation. General recursion has no counterpart in an ITS and a pipeline that targets only C would not have this problem and could handle recursive functions, although at the cost of dropping the ITS approach and the tools used for them. Non-numeric functions are excluded for a related reason, because for an integer loop a ranking function is a numeric expression over the loop variables, whereas functions over strings or lists would need a ranking function defined over those structures, such as the length of a list, which again lies outside the integer-loop setting the pipeline targets. Integer division and modulo could also be removed before the translation, by replacing each occurrence with a small helper loop that computes the quotient and remainder using only addition, subtraction and comparison. This would let KoAT and VeryMax accept the inputs they currently reject.

A second direction concerns the input side of the pipeline. Because the tools accept only C or dedicated transition system formats, while Wikifunctions functions are usually written in languages such as Python or JavaScript, every function first had to be translated. If Wikifunctions were to offer C implementations alongside its existing ones, the tools could analyze them directly, which would remove the translation stage for the C-based tools and reduce the faithfulness questions that come with it.

A further direction concerns the inputs used to validate the translations. Wikifunctions own test cases could be run alongside the generated inputs, so that every translation is additionally checked on the cases the community considered worth recording.

The pool of potential functions that can be analyzed could also be enlarged. Only a small fraction of the extracted functions met the integer loop criteria, so as the platform grows and more functions are added more of them will become suitable for analysis over time. Equally, the community or a targeted effort could contribute more useful functions that fit these criteria, which would give the evaluation a broader and more varied basis.

Since the dataset contains only two implementations that are known not to terminate and the tools proved only one of them, a further direction would be to bring in established non-terminating examples, so that future evaluations can test non-termination detection more systematically than the present study

could.

Another possible direction would be to combine the tools into a single service rather than using them separately. Since the study showed that different tools decide different functions, such a service could run several tools together, for instance the strongest tools from the C and ITS categories of the Termination Competition and return the first conclusive answer. This is only one option, but it would turn the observation that the tools complement each other into something directly usable.

Finally, the results could be fed back into Wikifunctions itself. The platform could attach a metadata flag to each function, for example marking it as verified terminating or verified non-terminating, populated by running the termination tools, ideally on powerful hardware so that timeouts and memory limits play a smaller role. Anyone using a function would then see at a glance that its termination or non-termination had been checked and which of the tools that were run on it were able to confirm it. Implementing this would require Wikifunctions to provide a built in way to run the analysis and record the result. This would bridge the gap between automated termination analysis and open collaborative function repositories such as Wikifunctions.

7 Conclusion

This thesis investigated whether existing automated termination tools can be applied to functions from Wikifunctions. Rather than building a new termination tool ourselves, we developed a pipeline that extracts Python implementations from the Wikifunctions XML dump, identifies the integer loop functions among them, translates those into the input formats of five established tools, validates each translation with a differential test and runs the tools in the same Docker setup used by the Termination Competition. The three research questions asked which tools are suitable for this task, how much they can prove, and how their results compare.

Regarding suitability, all five tools could be set up and executed, but their applicability differed far more than expected. The two C based tools accepted all 43 translated inputs, in MuVal's case after a one line fix to the C template, whereas KoAT's parser rejected 37 of its 43 inputs VeryMax's front end aborted on 27 of them, and the standardized ARI route could not be used at all. Applicability in practice therefore depends less on producing a correct input format than on meeting the specific expectations of each tool's front end.

Regarding coverage, the tools combined reached a conclusive verdict for 33 of the 43 functions, with 32 proven terminating. MuVal alone proved 29, making it the strongest individual tool. The main obstacle is non-linear arithmetic, as integer division and modulo, which are frequent in the number theoretic functions that dominate the benchmark, are precisely where the weaker tools fail. The single non-terminating verdict is the most telling result here. The "is factorial" implementation that MuVal flagged does not loop forever by construction but because of a real error in a live repository function, one that manual inspection confirmed. In this sense, the evaluation uncovered a genuine bug in Wikifunctions.

Regarding the comparison, the tools never contradicted one another, with perfect pairwise agreement on the functions that both tools in a pair decided, although these overlaps are small. More importantly, the tools are complementary rather than redundant, as 20 of the 33 decided functions were decided by exactly one tool. The two C based tools strictly dominated the three transition system based ones on this benchmark and the tools differed characteristically in how they fail, from honest "unknown" answers to front-end rejections, crashes and memory exhaustion.

The broader answer to the question behind this thesis, whether existing automated termination tools can be applied to functions from Wikifunctions, is therefore a qualified yes. Existing termination tools can be applied to Wikifunctions and where they apply they are trustworthy and useful, but only a small part of the repository is currently within reach, with 43 of its 1,968 implementations. That figure reflects the repositories content as much as the pipeline, since nearly three quarters of the implementations neither loop nor are recursive and therefore terminate trivially, posing no termination question in the first place. Among the implementations that do iterate, however the limit is the translation step rather than the tools themselves. Two practical conclusions follow. First,

because the tools strengths barely overlap, they are best used as a portfolio, running several of them on each function and taking the first conclusive answer rather than relying on any single tool. Second, the verdicts they produce are exactly the kind of information a repository like Wikifunctions could show to its users. For instance as a metadata flag marking a function as verified terminating by the tool it got verified by. In this study that analysis was carried out entirely offline, since the tools are not part of the platform and turning it into such a flag would require Wikifunctions to integrate the analysis and record its result. For the integer loop fragment studied here, the results suggest that this is a realistic goal and extending it to recursive and non-numeric functions and to a broader set of non-terminating examples, is the natural direction for future work.

References

[1] Anthropic. www.anthropic.com. Accessed 13-07-2026.

[2] TermComp. https://hub.docker.com/u/termcomp. Docker Hub. Accessed 20 June 2026.

[3] C integer programs. https://termination-portal.org/wiki/C_Integer_Programs, 2026. Accessed 16 June 2026.

[4] Data dumps. https://meta.wikimedia.org/wiki/Data_dumps, 2026. Accessed 14 June 2026.

[5] Integer transition systems. https://termination-portal.org/wiki/Integer_Transition_Systems, 2026. Accessed 16 June 2026.

[6] Term Rewriting. https://termination-portal.org/wiki/Term_Rewriting#Integer_Transition_Systems, 2026. Accessed 18 June 2026.

[7] Termination Competition. https://termination-portal.org/wiki/Termination_Competition, 2026. Accessed 16-06-2026.

[8] Termination Competition. https://termination-portal.org/wiki/Termination_Competition_2025, 2026. Accessed 20-08-2026.

[9] TRS standard. https://termination-portal.org/wiki/TRS_Standard, 2026. Accessed 16 June 2026.

[10] ZObjects. https://meta.wikimedia.org/wiki/Abstract_Wikipedia/ZObject, 2026. Accessed 14 June 2026.

[11] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery.

[12] Dana Angluin, Michael Frazier, and Leonard Pitt. Learning conjunctions of horn clauses. *Machine Learning*, 9(2):147–164, 1992.

[13] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Sat-based maxsat algorithms. *Artificial Intelligence*, 196:77–105, 2013.

[14] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51. Springer, 2015.

[15] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination through conditional termination. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–117. Springer, 2017.

[16] Cristina Borralleras, Daniel Larraz, Albert Oliveras, José Miguel Rivero, Enric Rodríguez-Carbonell, and Albert Rubio. Verymax: tool description for termcomp 2016. In *15th International Workshop on Termination*, volume 18, 2016.

[17] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 491–504, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[18] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, pages 113–129, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[19] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 387–393. Springer, 2016.

[20] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(4):1–50, 2016.

[21] Michael A. Colón and Henny B. Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–81, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[22] Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 442–454, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[23] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. *ACM Sigplan Notices*, 41(6):415–426, 2006.

[24] Stephen Cook. The p versus np problem. *Clay Mathematics Institute*, 2(6):3, 2000.

[25] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. 01 1978.

[26] NEGAR FATHI, HIROSHI UNNO, TACHIO TERAUCHI, and RAHUL PURANDARE. Sound termination and non-termination analysis of c programs with bit-precise bounded semantics and advanced constructs. 2026.

[27] Carlo A Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys (CSUR)*, 46(3):1–51, 2014.

[28] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, et al. Proving termination of programs automatically with aprove. In *International Joint Conference on Automated Reasoning*, pages 184–191. Springer, 2014.

[29] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[30] Patrice Godefroid. The soundness of bugs is what matters (position statement). In *BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, 2005.

[31] Weiwei Gong and Xu Zhou. A survey of sat solver. In *AIP Conference Proceedings*, volume 1836, page 020059. AIP Publishing LLC, 2017.

[32] Ashutosh Gupta, Thomas A Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–158, 2008.

[33] Naoki Kobayashi, Takeshi Nishikawa, Atsushi Igarashi, and Hiroshi Unno. Temporal verification of programs via first-order fixpoint logic. In *International Static Analysis Symposium*, pages 413–436. Springer, 2019.

[34] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Minimal-model-guided approaches to solving polynomial constraints and extensions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 333–350. Springer, 2014.

[35] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[36] Nils Lommen and Jürgen Giesl. Aprove (koat + loat). In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 205–211, Cham, 2025. Springer Nature Switzerland.

[37] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

[38] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[39] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 239–251, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[40] ProofWiki. Infinite sequence property of strictly well-founded relation. `https://proofwiki.org/wiki/Infinite_Sequence_Property_of_Strictly_Well-Founded_Relation`, 2026. Accessed 5 May 2026.

[41] Python Software Foundation. Abstract syntax trees. `https://docs.python.org/3/library/ast.html`, 2026. Accessed 16 June 2026.

[42] RWTH Aachen University. KoAT. `https://koat.verify.rwth-aachen.de/`, 2026. Accessed 16 June 2026.

[43] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. Proving termination and memory safety for programs with pointer arithmetic. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 208–223, Cham, 2014. Springer International Publishing.

[44] Terence Tao. Almost all orbits of the collatz map attain almost bounded values. *Forum of Mathematics, Pi*, 10:e12, 2022.

[45] TermCOMP. ITS-Conversion. `https://github.com/TermCOMP/its-conversion`, 2026. GitHub repository. Accessed 27 June 2026.

[46] TermCOMP. Termination portal. `https://termination-portal.org/`, 2026. Accessed 16 June 2026.

[47] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42:230–265, 1936.

[48] RWTH Aachen University. KoAT - input format. `https://koat.verify.rwth-aachen.de/getting_started/input_format/`. Accessed 25-08-2026.

[49] Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. Modular primal-dual fixpoint logic solving for temporal verification. *Proceedings of the ACM on Programming Languages*, 7(POPL):2111–2140, 2023.

[50] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. Constraint-based relational verification. In *International Conference on Computer Aided Verification*, pages 742–766. Springer, 2021.

[51] Denny Vrandečić. Building a multilingual wikipedia. *Commun. ACM*, 64(4):38–41, March 2021.

[52] Wikimedia Foundation. Wikifunctions: About. `https://www.wikifunctions.org/wiki/Wikifunctions:About`, 2026. Accessed 16 June 2026.

A Code and Data Availability

All code, translations, tool outputs, and result tables produced for this thesis are publicly available in the following repository:

`https://github.com/16x29/Master-Thesis---Certainly-Terminating-Wikifunctions`

The repository contains the complete analysis notebook, the SQLite database of record (the original Python implementations, the generated translations, and the verbatim outputs of every tool run), the translated `.c`, `.t2`, and `.koat` and `.ari` files and the result tables reported in Section 5. Its `README` documents how to reproduce every result. Using the provided database, all tables in this thesis can be regenerated without re-running the tools, and a full re-run of the five tools in their official Termination Competition Docker images is documented as well.

The algorithms given in Section 4 are implemented in the notebook *wikifunctions_thesis_pipeline.ipynb* in that repository. Table 10 states where each of them can be found.

Thesis element	**Implementation in the notebook**
Algorithm 1	*build_cfg*, with the emitters *emit_c*, *emit_t2*, *emit_koat*, *emit_ari*
Algorithm 2	*difftest*
Algorithm 3	*run_batch_tier*
Verdict categories	*classify*
Table 7	Stage 6, RQ2 tool-results summary
Table 9	Stage 6, RQ3 tool-agreement matrix

Table 10: Where each element of Sections 4 and 5 is implemented in the notebook.

B Use of AI Assistance

In the preparation of this thesis, a large language model was used for grammatical and linguistic correction, as well as for coding the notebook.