Polynomial Postconditions via mwp-Bounds

Anonymous author
Anonymous affiliation

Abstract

Formal specifications are necessary for guaranteeing software meets critical safety properties, but retrofitting specifications to existing software requires considerable resources and expertise. Although inference techniques ease specifications discovery, existing methods are limited in targeting different specification conditions. In particular, inference of postconditions—partial specification conditions that must hold after program execution—has received little attention. In this paper, we present a static program analysis for automatically inferring postconditions from program syntax. Our solution is a compositional, sound technique for bounding variable values in imperative programs. For each variable, it finds one approximative postcondition of at most polynomial form, if a postcondition is expressible in the underlying calculus. The technique is applicable in imprecise contexts, in absence of exact variable values, and without reliance on external solvers or annotations. The technique is based on the complexity-theoretic flow calculus of mwp-bounds, but required us to enhance the theory in several ways to obtain a solution for postcondition inference. We have implemented our analysis, mwp$_\ell$, on numerical loops in C. Our experiments show the analysis generalizes to classic algorithms and finds postconditions for 69% or more variables across the evaluated benchmarks. The results suggest the technique can assist software engineers in specification tasks at development-time, when complete program details are still uncertain.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Theory of computation → Complexity theory and logic; Theory of computation → Logic and verification

Keywords and phrases Formal Methods, Static Program Analysis, Program Verification, Postcondition Inference

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Software engineers have an aphorism that warns against publishing releases on Fridays. It is shared lightheartedly, but embeds serious commentary about the normalcy of software instability. Formal methods provides the techniques to improve and achieve rigorous software quality guarantees. Unfortunately, integrating formal methods to mainstream software development workflows remains challenging due to, e.g., lack of training and tool-specific issues [43]. Continued research effort must be dedicated to reducing the entry barriers.

Our results take a step toward making formal methods more accessible by introducing a solution for partial specifications inference. More precisely, we focus on automatic inference of loop *postconditions*; assertions that must hold after a loop terminates. Obtaining a complete formal specification requires combining postconditions with preconditions and loop invariants. The specification can then be verified by a theorem prover. While there has been considerable research on automatic inference of specification conditions—particularly of invariants, reviewed in the related works of Sect. 8—postcondition inference has been largely overlooked, with a few exceptions [35, 28]. But the study is warranted because postconditions assist discovery of other verification conditions [16]. Furthermore, as generalized assertions, postconditions offer benefits in other software development activities, e.g., debugging, testing, and maintenance [38, 49, 2].

Defining formal specifications is a nontrivial task and requires expertise. Specifications must be consistent, precise, and complete descriptions of functional behavior. When expressed in natural language, ambiguity and omissions arise. We avoid the issue by applying inference directly to program syntax. In this view, a program is assumed to implement its intended behavior, but misses a formal proof. Our solution is a fully static analysis to infer approximative postconditions of variables in numerical loops. When paired with complementary inference techniques and verification tools, it leads to full formal guarantees.

1.1 Overview

We analyze deterministic imperative loop programs with the goal of automatically inferring postconditions. A postcondition is an assertion about the loop's variable values that holds after iteration, if the loop terminates. Our postconditions are approximative, but optimal in expressiveness of the underlying theory. The loop structure is unrestricted: arbitrary termination and update conditions, loop types, jump statements, data-flows etc., are allowed. A program manipulates a fixed number of natural number variables and, beyond type, we make no assumptions about initial values. Reducing contextual information is practically motivated since such information is commonly absent in unverified realistic programs.

▶ **Example 1** (*LucidLoop*). Listing 1 demonstrates a canonical verification problem. Given a specification with a precondition (`assume`) and a loop command (`for`), the goal of formal verification is to prove the postcondition (`assert`) is satisfiable. Listing 2 shows the problem variant we address in this paper. When precise variable values, precondition, and iteration count are unknown, what postcondition (?) can we infer from the syntax?

Listing 1 Precise context

```
assume(X2==1∧X4==2∧X5==4);
for(i=0;i<10;i++) {
  X3=X2*X2;
  X3=X3+X5;
  X4=X4+X5; }
assert(X4==42);
```

Listing 2 Imprecise context

```
for(i=0;i<X1;i++) {
  X3=X2*X2;
  X3=X3+X5;
  X4=X4+X5; }
assert(?);
```

Our goal is to infer a partial specification conditions (postconditions) that then assist discovery of complete and precise specifications and permit formal verification. ◻

The analysis we present derives *mwp-bounds* [22], a kind of value growth bounds of program variables. An mwp-bound represent a variable's final value relative to initial variable values, omitting constants. We categorize variables by the mwp-bound form as linear, iteration-independent, iteration-dependent, or inconclusive; to reflect the maximal value growth (cf. Sect. 5.4). A critical idea is that the **mwp-bounds are postconditions**.

▶ **Example 1** (cont.). At the program point ?, our technique gives the following result.

- Values of variables `X1`, `X2`, and `X5` have grown at most linearly from the initial values.
- Variable `X3` value is iteration-independent and bounded by $\max(\texttt{X3}, \texttt{X2} + \texttt{X5})$.
- Variable `X4` value is iteration-dependent and bounded by $\texttt{X4} + \texttt{X1} \times \texttt{X5}$.

We can determine manually the precise postconditions for comparison. The linear variables do not change from the initial values. If the loop iterates, final value of `X3` is $\texttt{X2}^2 + \texttt{X5}$, and remains unchanged otherwise. Variable `X4` postcondition is precise as expressed. ◻

In the remainder of the introduction, we give a teaser of how we achieve this result and highlight the key insights.

Connecting complexity and verification. Loop analysis is a classic challenge connecting computational complexity and verification communities. In complexity theory, loops hold such a pivotal role it suffices to concentrate complexity analysis to just loop commands [5]. In verification, inferring inductive loop invariants—conditions implied by a loop's precondition and preserved in each iteration—is among the most challenging problems [14, 42, 47]. Although the communities differ in their motivations for studying loops, we focus on the intersection. Complexity-theoretic analyses seem naturally suited for postcondition inference, yet we have found only a few publications making related observations [32, 34]. By demonstrating that insights from complexity theory transfer to program verification, we hope to inspire similarly motivated future investigations.

From implicit complexity theory to explicit static analyses. The field of *implicit computational complexity* [13] differs from traditional complexity theory in its aims to discover machine-independent characterizations of complexity classes. A foundational concept is introducing *restrictions* at the level of a programming language, such that any program satisfying the restriction is guaranteed to have desirable runtime behavior. Implicit complexity is implicit in at least two ways: computational model and explicit program bounds need not be specified [30]. Our results demonstrate how the syntactic orientation makes implicit complexity a natural candidate for integrating complexity-theoretic solutions into practical applications.

Small core languages to reason about big programs. Although static program analysis has introduced a variety of rich analyses, constructing a perfect analyzer is impossible [36]. This creates endless opportunity in designing increasingly useful approximative analyses. Typically, the analyses trade between precision and termination, but rarely sacrifice expressiveness [29, p. 4]. Our approach counters this norm. We start with a small core language, then map it to a subset of a Turing-complete language. The justification is, if sufficient program fragments can be analyzed, the technique yields useful feedback. This is possible thanks to built-in compositionality that avoids common scalability issues [41, 6].

1.2 Contributions

Our technical contributions are summarized below.

- We present a solution for automatic postcondition inference based on the flow calculus of mwp-bounds. It supports software engineers in program verification by reducing manual effort in specifications generation. The technique is applicable in imprecise contexts and in absence of exact variable values.
- We extend the flow calculus of mwp-bounds with two new capabilities: locating optimal variable bounds and bounding variables in presence of whole-program derivation failure. These extensions produce a strictly more expressive system than its predecessors.
- We implement mwp_ℓ, a static analysis of loops in C, that materializes our theory. mwp_ℓ is already integrated into a public static analyzer pymwp, extending the utility of our results beyond the presentation of this paper.
- We demonstrate the uniqueness and effectiveness of our postcondition inference in an experimental evaluation. The results validate the theoretical claims, provide evidence of practical efficiency, and show the technique generalizes to finding postconditions in natural algorithms. We are unaware of other static tools with comparable behavior.

2 Preliminaries

2.1 Loop specifications for obtaining formal guarantees

In formal methods programs are defined as precise mathematical models through specifications. Formally verifying a program involves providing a proof that the program satisfies its specification. In contrast to tests and inspection, a proof conclusively ensures behavioral correctness at the modeled level of abstraction.

A *specification* is a formal contract of three components. **1.** *Precondition* P, the initial logic expressions assumed to hold before entering the loop. **2.** Program, `loop b C`, performing command `C` until the boolean control expression `b` becomes false. **3.** *Postcondition* Q, the final logic expressions asserted to hold after the loop terminates. Using a Hoare triple [18], we can express the specification as $\{P\}$ `loop b C` $\{Q\}$. A loop is correct if we can construct a proof that the loop satisfies its specification in all program states. More precisely, a correctness proof requires verifying that every computation terminates, every call to another procedure satisfies its preconditions, and the postcondition holds at loop termination [16].

In this paper, we focus on postconditions. A postcondition is a higher-level view describing the goal of the program. Informally, we express the postcondition inference problem as follows. Given a precondition P (in our formulation a constant true, $\top$), and a program `loop b C`, the inference problem involves finding a postcondition Q that satisfies the inference rule $P = \top, \{\mathtt{b}\}\, \mathtt{C}\, \{\top\}, \neg\mathtt{b} \rightarrow Q \vdash \mathtt{loop\ b\ C}$. Because the inference technique we present is sound, the postconditions we infer are expectedly provable. However, pre- and postcondition alone are typically too weak to prove the program correct. They require loop invariants to strengthen the assertions and make the specification provable.

An *invariant* is an assertion of a program location that is true of any program state reaching the location. A loop invariant is always a weakened form of the postcondition [16]. An invariant is *inductive*, if it holds the first time the location is reached, and is preserved in every cycle returning to the location [40]. Verification of loops requires discovering *sufficiently strong* inductive invariants to prove the specification. For an invariant to be sufficient, it must be weak enough to be derived from the precondition, and strong enough to conclude the postcondition. Loop invariant inference is one of the most difficult problems in verification [14, 47].

Inductive loop invariants and postconditions are symbiotic: the discovery of one assists finding the other. However, it is important to recognize their difference. Every invariant is necessarily a (weak) postcondition, as it must hold at termination; but a postconditions must not be invariant. For example, the loop over natural numbers `i` and `n`, `for(i=0;i<n;i++) i++;` has an invariant $0 \le \mathtt{i} \le \mathtt{n}$ and a postcondition $\mathtt{i} = \mathtt{n}$. Invariant inference solutions frequently assume preconditions and postconditions are known, but this assumption is impractical. Manually annotating code fragments with specifications is nontrivial and laborious.

Given a candidate specification, an automated theorem prover can check if the inference rule premises hold and prove the conclusion. When negative, a theorem prover provides a counterexample witnessing the failure. Deductive verifiers simplify the task further by performing the proof checking internally through compilation. In deductive verification, assertions that imply correctness are defined at various program points [20]. The verifier does not synthesize a proof, but rather uses the available hints to check that a program adheres to its specification [7]. For example, the verification-aware programming languages Dafny [24] and Viper [31] support adding assertions next to program elements, then discharge the proof obligations to a theorem prover in the background. No further effort is required from a engineer to prove the correctness of a sufficiently strong specification.

2.2 Final values with the flow calculus of mwp-bounds

The *flow calculus of mwp-bounds* [22, 3] is a complexity-theoretic program analysis for reasoning about variable value growth in imperative programs. The analysis aims to discover a polynomially bounded data-flow relation between the *initial values* $x_1, \ldots, x_n$, for natural-number variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$, and the *final values* x_i' of $\mathtt{X}_i$ (for $i = 1, \ldots, n$). If it is possible to determine all variable values grow at most polynomially in inputs, the analysis assigns the program a bound to characterize the value growth. The conclusion is derived statically and syntactically by applying inference rules to the commands of the program. It is a purely mathematical analysis; no external solvers are needed.

As an internal bookkeeping procedure, the analysis tracks *coefficients* (or "flows") representing how data flows in variables between commands. The coefficient are, in order of lowest to highest degree of dependency: $\mathbf{0}$, indicating no dependency; $\boldsymbol{m}$ for *m*aximal of linear, $\boldsymbol{w}$ for *w*eak polynomial, or $\boldsymbol{p}$ for *p*olynomial. Finally, $\boldsymbol{\infty}$ stands for failure when no value growth bound can be established. For example, if an exponential dependency exists between two variables, the analysis assigns an ∞-coefficient to the target variable of the data flow.

The analysis result is binary. It states whether all final values can be bounded by polynomials in inputs. When affirmative, the input program is *derivable*. The analysis assigns an *mwp-bound* to every variable of a derivable program. An mwp-bound characterizes the value growth of a variable. If a derivable program terminates, the soundness theorem of the flow calculus guarantees the variable value growth is polynomially-bounded [22, p. 11]. The result is sound but not complete. Since the analysis omits termination, this is a partial correctness guarantee.

The flow calculus offers no guarantee for programs that are not derivable. Then, variable value growth is interpreted as unknown. A program always fails if a variable value grows "too fast", for example exponentially. A single variable can be the source of whole-program failure. Alternatively, failure may occur from inability to express satisfiable behavior. The latter is a built-in limitation of the analysis. To increase expressiveness and capture a larger class of derivable programs, the calculus includes nondeterminism in its inference rules. One program may admit multiple derivations, which in turn complicates the analysis. However, a program is derivable if there exists a derivation without ∞-coefficient.

2.3 From flow calculus to verification: opportunities and challenges

The preliminaries thus far have covered two rather disjoint topics, yet we aim to unify them. The goal relies crucially on two judgements. The first is the intuition, and claim (supported by Sect. 7), that the flow calculus is *practically useful* for postcondition inference. It fits a research gap as we are unaware of comparable alternatives. The second relies on recent advancements [3] in the flow calculus theory. It has matured sufficiently from its origin to now be ready for applications. We are scientifically compelled to explore the potential.

However, our goal goes beyond connecting dots. In general, advancing theoretical concepts to practice exposes limitations in the theory. Although many past inefficiencies of the flow calculus have been resolved, particularly through the ∞-coefficient; it has in turn introduced new challenges. The underlying nondeterminism gives rise to a state explosion problem. Clever strategies are necessary to handle derivation failure and determine mwp-bounds. The most recent advancement [4] describes a method of counting program bounds and producing arbitrary bound instances. It remains an open question to find approaches that enable mwp-bounds exploration. For example, determining existence of a mwp-bound satisfying criteria, identifying unique mwp-bounds, and reasoning about mwp-bounds at failure are beyond the current capabilities. We extend the state-of-the-art in these directions, which then gives a technique for postcondition inference.

3 Technical foundations

Sections 3–5 present the technical background and enhancements needed to obtain postcondition inference. These details are not necessary to fluently comprehend the rest of the paper. A more gratifying reading experience may follow from reading out of order.

3.1 Base Language and matrix construction

▶ **Definition 2** (Imperative language). *Letting natural number variables range over* `X` *and* `Y` *and boolean expressions over* `b`*, we define* expressions `e`*, and* commands `C` *as follows*

```
e ::= X || X - Y || X + Y || X * Y
C ::= skip || X = e || if b then C else C || while b C || loop X C || C;C
```

The command `loop X C` *means "do* `C` `X` *times" and the variable* `X` *is not allowed to occur in command* `C`*. Command* `C;C` *is for sequencing. We write "program" for a series of commands composed sequentially.* ◻

In examples, we implicitly convert between conventional **for** loops and `loop`s of the imperative language. The values of boolean expressions do not matter; for emphasis we substitute `b` with `*` in control expressions. Although the base language is rudimentary, it captures a core fragment of conventional mainstream programming languages, like C and Java. Our technique applies to all languages sharing the same core, including intermediate representations. Conversely, the approach is applicable even if a programming language provides little structure.

The flow calculus of mwp-bounds assigns *mwp-matrices* to expressions and commands of a program. The matrices track, with coefficients, how data flows in variables between commands. The procedure for assigning matrices is defined by the inference rules of the mwp-calculus [3, Sect 2.2]. The calculus omits evaluating loop iteration bounds and control expressions, trading precision for efficiency. At conclusion, the calculus assigns one matrix to the analyzed program that characterizes how variable values grow during computation. Construction of matrices is not relevant for the developments of this paper. Our enhancements concern using the matrices once they exist. A reader interested in an exposition of mwp-matrix construction should refer in order to [22, Sect 5–6] and [3, Sect 2–3].

3.2 Decoding mwp-matrices

An mwp-matrix is an abstraction that captures data flow facts about the analyzed program. Because an mwp-matrix contains an exponential number of derivations, it can appear complicated. The best strategy for understanding it is recursive, starting from the elements.

3.2.1 The mwp-matrix elements

Coefficients. The coefficients $\text{MWP}^{\infty} = \{0, m, w, p, \infty\}$ characterize variable dependencies in commands. They are the core building blocks of mwp-matrices (hence the name). A programs with simple data flows is assigned an mwp-matrix of plain coefficients. However, the coefficients are insufficient to represent nondeterminism. We will discuss the meaning of the coefficients more in Sect. 5.

Domain. In mwp-calculus, one command has up to 3 derivations. A domain $\mathcal{D} = \{0, 1, 2\}$ is a bidirectional map to encapsulate the derivation options. The domain members can be translated to mwp-calculus rules and vice versa.

Degree. The degree of choice, $k : \mathbb{N}$, is a counter of admissible derivations. In other words, it is the number of times a derivation choice has to be made. The degree is equal to the count of binary arithmetic operations in the analyzed program, as these operations are the source of nondeterminism.

A derivation choice δ is the second core building block of mwp-matrices. It contains the nondeterminism of the mwp-calculus.

▶ **Definition 3** (Derivation choice). *Letting $\mathcal{D}$ be a domain and $k \in \mathbb{N}$ be the degree of choice, we define a derivation choice as $\delta(i, j)$, where $i \subseteq \mathcal{D}$ and $j \leq k$.* ◻

We call i the *value* and j the *index* of the derivation choice. The value indicates the selected mwp-calculus inference rule and index the program point where the rule is applied. Since a program is a *series* of commands, correspondingly we have *sequences* of derivation choices, denoted by $\Delta = (\delta_1, \delta_2, \dots, \delta_k)$. There are two restrictions on Δ. An index is allowed to occur at most once and the sequence is sorted by index in ascending order. The restrictions support efficient computation during matrix construction. Since we are concerned with interpreting mwp-matrices, we assume every Δ satisfies the uniqueness and order properties by construction. In an mwp-calculus, different coefficients can be assigned to one program variable. The fact that Δ produces a specific coefficient is represented in a *monomial*.

▶ **Definition 4** (Monomial). *Letting $\alpha \in \textsc{mwp}^{\infty}$ be a coefficient and Δ be a sequence of derivation choices, we define a monomial as (α, Δ).* ◻

Simple data flow patterns do not require derivation choices, thus Δ is empty. For example, $m.(\delta(0,0), \delta(2,1))$ and 0 are both monomials by definition. Finally, the fact that different Δ produce different coefficients is represented by a *polynomial structure*. The elements of an mwp-matrix are polynomial structures.

▶ **Definition 5** (Polynomial structure). *We define a polynomial structure as a sequence of monomials $\big(0, (\alpha_1, \Delta_1), (\alpha_2, \Delta_2), \dots, (\alpha_m, \Delta_m)\big)$ where $m \geq 0$.* ◻

3.2.2 Reading an mwp-matrix by example

Conceptually, an mwp-matrix represents the derivations of an analyzed program. Concretely, it connects program variables with polynomial structures. The matrix size is determined by the number of program variables. Given n variables, the matrix size is $n \times n$. The matrix is labelled by the program variables and the elements are polynomial structures[1]. An mwp-matrix M is interpreted column-wise. The data-flow facts about a variable at column j are collected in rows $i = (1, \dots, n)$ in M_{ij}.

▶ **Example 6** (mwp-matrix). Consider the mwp-matrix assigned to *LucidLoop*:

```
for(i=0;i<X1;i++)
{ X3=X2*X2;
  X3=X3+X5;
  X4=X4+X5; }
```

$$:\quad \begin{array}{c} \\ \texttt{X1} \\ \texttt{X2} \\ \texttt{X3} \\ \texttt{X4} \\ \texttt{X5} \end{array} \begin{array}{c} \begin{array}{ccccc} \texttt{X1} & \texttt{X2} & \texttt{X3} & \texttt{X4} & \texttt{X5} \end{array} \\ \begin{pmatrix} m & 0 & p(0,1), p(1,1) & p(0,2), \infty(1,2), \infty(2,2) & 0 \\ 0 & m & w(0,1), p(1,1), w(2,1) & \infty(1,2), \infty(2,2) & 0 \\ 0 & 0 & m & \infty(1,2), \infty(2,2) & 0 \\ 0 & 0 & 0 & m, \infty(1,2), \infty(2,2) & 0 \\ 0 & 0 & p(0,1), m(1,1), w(2,1) & p(0,2), \infty(1,2), \infty(2,2) & m \end{pmatrix} \end{array}.$$

[1] For clarity and compactness, we omit needless 0's, δ-symbols, and delimiters in mwp-matrices.

Variables X1, X2, and X5 are assigned at most m-coefficients. An m at the diagonal means the variable depends on its (own) initial value. The absence of derivation choices implies exactly one mwp-bound describes each of the three variables. Variable X3 shows varying dependencies on X1, X2, and X5 in different derivations; however, the dependency is at most $p$olynomial. The absence of ∞ means the value growth of X3 is acceptable in all derivations. Variable X4 is assigned ∞ in some derivations. This signals that some derivation will fail and the source of failure is X4. ◻

Later in the paper we develop enhanced strategies to extract more precise information from mwp-matrices. Although we show only compact cases, an mwp-matrix captures 3^k derivation choices where k is the count of binary operations. Clever solutions are necessary to explore and interpret the accumulated data efficiently, but have been missed by previous analysis refinements [3, 4]. An efficient evaluation strategy to handle complex mwp-matrices is required for postcondition inference.

3.3 Interpreting mwp-bounds

The columns of an mwp-matrix encode variable value growth bounds. An mwp-bound is an expression of form $\max(\vec{x}, \text{poly}_1(\vec{y})) + \text{poly}_2(\vec{z})$. Variables characterized by m-flow are listed in $\vec{x}$; w-flows in $\vec{y}$, and p-flows in $\vec{z}$. Variables characterized by 0-flow do not occur in the expression and no bound exists if some variable is characterized by ∞. The poly_1 and poly_2 are *honest polynomials*, build up from constants and variables by applying $+$ and $\times$. Any of the three variable lists might be empty, and poly_1 and poly_2 may not be present. When characterizing value growth, an mwp-bound is approximative; it excludes precise constants and degrees of polynomials. A *program bound* is a conjunction of its variables' mwp-bounds.

▶ **Notation 1.** *Letting* X *be a variable and* W *an mwp-bound, we write* X' $\leq W$ *to denote the variable's mwp-bound. We use* X' *to denote the variable at the postcondition where* X *refers to its final value. The variables in* W *always refer to initial values.* ◻

▶ **Example 7.** The mwp-bound is interpreted as, "the *final value* of X is bounded…"

$$\begin{aligned}
\mathtt{X'} &\leq 0 && \text{...by a constant.}\\
\mathtt{X'} &\leq \mathtt{X} && \text{...linearly in X (its initial value).}\\
\mathtt{X'} &\leq \max(\mathtt{X}, \mathtt{X1}+\mathtt{X2}) && \text{...by a (weak) polynomial in X or X1 + X2.}\\
\mathtt{X'} &\leq \max(\mathtt{X}, \mathtt{Y}) + \mathtt{X1}\times\mathtt{X2} && \text{...by a polynomial in X or Y, and X1} \times \text{X2.}
\end{aligned}$$
◻

4 Variable-guided matrix exploration

We introduce two solutions to address current limitations of the flow calculus of mwp-bounds.

1. An mwp-matrix *evaluation strategy*, to efficiently determine if an individual variable admits an mwp-bound of specific form.
2. Improved *failure handling*, to identify variables that maintain acceptable value growth behavior in presence of whole-program derivation failure.

To capture more programs, the mwp-calculus internalizes nondeterminism that causes a potential state explosion problem. Effectively handling this feature becomes critical in the second program analysis phase, where an mwp-matrix is evaluated to find mwp-bounds for variables. The first challenge with evaluation is finding the derivation choices that avoid failure. The second is determining the flow-coefficients assigned to each variable. The coefficients are not obvious from the derivation choices; rather, they require an application to the mwp-matrix. An *application* reduces the polynomial structures of an mwp-matrix to simple coefficients, which in turn produces either derivation failure or a program bound.

A brute force solution iterates all derivations and applies all choices to observe the result. However, such naive solution is impractical due to latency and a potential yield of exponentially many program bounds. An ideal solution finds the optimal bound, if it exists, and without redundancy of iterating every derivation.

The evaluation strategy we present operates on *unwanted* derivation choices and negates them. The evaluation result captures the permissible choices. The term *permissible* is abstract; the meaning depends on what is unwanted. For example, if the derivation choices of ∞-coefficients are unwanted, the permissible choices are those that avoid failure (non-∞). The result of our evaluation strategy is a *disjunction of choice vectors* that compactly capture the permissible derivation choices.

▶ **Definition 8** (Choice vector). *Letting $\mathcal{D}$ be a domain and $k \in \mathbb{N}$ be a choice degree, we define a choice vector as $\vec{C} = (c_1, c_2, \ldots, c_k)$, where $c_i \subseteq \mathcal{D}$ and $c_i \neq \varnothing$ for all $i \in \{1, 2, \ldots, k\}$.* ◻

We show choice vectors in Examples 9, 10, and 11.

4.1 Evaluation for identifying derivable programs

The precise mwp-matrix evaluation strategy is presented in Algorithm 1 and we describe it here informally. The evaluation is parametric on three inputs: **1.** degree of choice, k, **2.** domain $\mathcal{D}$, and **3.** a set of unwanted derivation choice-sequences, $\mathcal{S} = \{\Delta_1, \ldots, \Delta_n\}$ where $n \geq 0$. Observe that all parameters are obtained from an mwp-matrix, but the matrix or its coefficients are not used. The evaluation returns a list of choice vectors. In the maximally permissive case, the result is a single choice vector permitting everything. If no result exists or no choice is necessary, the result is an empty list. These outcomes are handled as base cases (Line 3). All interesting evaluations fall between these extremes.

The evaluation operates in two steps. First it simplifies $\mathcal{S}$ to the minimal-length, nonempty sequences while preserving its effect. Then, it generates the choice vectors by negating the remaining unwanted choices. The procedure is practically efficient because simplification eliminates redundancy before the choice vector are generated. Internally, it benefits from the finiteness of the domain and from having complete knowledge of all derivation choices.

The simplification step (Line 10) is crucial. It must be sound and complete, to ensure all distinct derivation patterns are preserved in $\mathcal{S}$, but also reduce $\mathcal{S}$ to the smallest set possible, to make the subsequent step efficient. Different simplifications are applied on $\mathcal{S}$, iteratively, until convergence; for example the following.

- Removing super-sequences – if a sub-sequence leads to an unwanted outcome, a longer sequence producing the same outcome is redundant.
- Head-elimination – if many non-singleton sequences differ only by the head element value, and the values equal the domain $\mathcal{D}$, the head is redundant. The many sequences can be replaced by one sequence without the head element.
- Tail-elimination – by symmetry, the same as above, but on the tail element.

The generation step (starting at Line 13) produces the choice vectors. They are constructed by computing the cross product of $\mathcal{S}$. We take one derivation choice from each sequence, then eliminating those choices. This prevents choosing any unwanted sequence completely. If the choice vector elements are nonempty after elimination, we appended the choice vector to the result (Line 19). We have annotated the computational costs of the iterative steps in Algorithm 1. Since the generation step involves a product, it is the source of potential

Algorithm 1 mwp-matrix evaluation

Input: degree k ($\mathbb{N}$), domain $\mathcal{D}$ (set), derivation choice-sequences $\mathcal{S}$ (set)
Output: choice vectors $\mathcal{C}$ (list)

$\mathcal{C} \leftarrow \varepsilon$
// Handle base cases
if $k = 0$ **or** $|\mathcal{D}| \leq 1$ **or** $\mathcal{S} = \varnothing$ **then**
  if $\mathcal{S} = \varnothing$ **then**
    Create $\vec{c} \coloneqq \text{ChoiceVector}(k, \mathcal{D})$ ▷ k-length vector of elements in $\mathcal{D}$
    $\mathcal{C} \leftarrow \vec{c} :: \mathcal{C}$
  return $\mathcal{C}$
// Step 1: Simplify until $\mathcal{S}$ convergences
do Capture initial *size* $\coloneqq |\mathcal{S}|$ ▷ $\mathcal{O}(\mathcal{S}^3)$
  $\mathcal{S} \leftarrow \text{SIMPLIFY}(\mathcal{S})$
while *size* $\neq |\mathcal{S}|$
// Step 2: Generate choice vectors
Compute $P \coloneqq$ the product of sequences in $\mathcal{S}$
for all paths p in P **do** ▷ $\mathcal{O}(\prod_{s \in \mathcal{S}} |s|)$
  Create $\vec{c} \coloneqq \text{ChoiceVector}(k, \mathcal{D})$
  for all (i, j) in p **do** ▷ $\mathcal{O}(|\mathcal{S}|)$
    Remove i at $\vec{c}_j$
  if $\forall j, \vec{c}_j \neq \varnothing$ **then**
    $\mathcal{C} \leftarrow \vec{c} :: \mathcal{C}$
return $\mathcal{C}$

inefficiency. However, we have not encountered natural programs where the simplification does not reduce $\mathcal{S}$ sufficiently to make the generation step problematic. A full implementation of the algorithm is included in our artifact, and described in more detail in the documentation of pymwp[2].

▶ **Example 9** (*LucidLoop* is derivable)**.** In Example 6, the monomials causing failure are $\infty.\delta(1,2)$ and $\infty.\delta(2,2)$. The matrix degree is $k = 2$, the domain is $\mathcal{D} = \{0,1,2\}$, and the unwanted derivation choice-sequences are $\mathcal{S} = \{((1,2)),((2,2))\}$. The evaluation returns a choice vector $\big(\{0,1,2\},\{0,1,2\},\{0\}\big)$ witnessing successful derivation choices. ◻

A choice vector application requires making a selection at each vector index, then applying the selection to the polynomial structures of the mwp-matrix. A monomial evaluates to $\delta(i,j) = \alpha$ if the jth choice is i, and 0 otherwise. A polynomial structure evaluates to its maximal coefficient. Thus, the application produces the maximal coefficient among the monomials, cf. Example 11.

4.2 Querying variable mwp-bounds in derivable programs

Until now, the flow calculus of mwp-bounds has been restricted to deriving mwp-bounds for all variables concurrently. For postcondition inference, we want to obtain information about *individual* variables. Since Algorithm 1 does not require mwp-matrices or coefficients as input, it is directly reusable for variable-specific evaluations.

[2] See: `https://statycc.github.io/pymwp/choice`

Analyzing variable $\mathtt{X}_j$ requires taking derivation choices only from column j, instead of the entire mwp-matrix. To determine if a variable admits a particular form of mwp-bound, we issue "queries" against the evaluation procedure, altering the derivation choices provided as parameter $\mathcal{S}$. For example, to find derivations with at most m-coefficients, if they exist, we take the derivation choices from monomials whose coefficient are in $\{w, p, \infty\}$. If the evaluation returns a choice vector, it specifies the derivations where the variable is assigned an mwp-bound of at most m-coefficients. Bounds of w- and p-form can be queried similarly by adjusting inputs to $\mathcal{S}$.

▶ **Example 10** (Variable X3 is bounded by at most w-coefficients)**.** In Example 6, variable X3 does not have a derivation with at most m-coefficients. This is determined by evaluation on derivation choices $\mathcal{S} = \{((0,1)),((1,1)),((2,1))\}$—the choices in column X3 with $\{w, p, \infty\}$ coefficients—which does not return a choice vector. However, an mwp-bound of at most w-coefficients exists by the choice vector $\big(\{0,1,2\},\{2\},\{0,1,2\}\big)$. ◻

Variable query is safe for derivable programs because all variables are guaranteed to have an mwp-bound by the soundness theorem of the mwp-calculus [22, p. 11]. Additional caution is needed when a program is not derivable.

4.3 Variable mwp-bounds in presence of failure

Derivation failures arise from the mwp-calculus inference rules and affect the loop commands `while` and `for`. If a program is not derivable, the calculus assigns no bound to its variables. Effectively, no guarantee is offered and variable value growth is labelled unknown. This treatment limits the utility of the analysis. Since our inference focuses on loops, this restriction is critical and excludes many programs we wish to analyze.

We claim the mwp-matrices accumulate sufficient information to permit reasoning about *certain* variables in presence of failure. For motivation, consider a variant of *LucidLoop* where all derivations produce failing ∞-coefficients.

▶ **Example 11** (Always failing derivation)**.** The program is not derivable because variable X4 is assigned ∞-coefficients in every derivation in the mwp-matrix,

```
while(*) {
  X3=X2*X2;
  X3=X3+X5;
  X4=X4+X5; }
```

$$:\quad \begin{array}{c} \\ \mathtt{X2} \\ \mathtt{X3} \\ \mathtt{X4} \\ \mathtt{X5} \end{array} \begin{array}{c} \begin{array}{cccc} \mathtt{X2} & \mathtt{X3} & \mathtt{X4} & \mathtt{X5} \end{array} \\ \begin{pmatrix} m & \infty(0,1),\infty(1,1),w(2,1) & \infty(0,2),\infty(1,2),\infty(2,2) & 0 \\ 0 & m,\infty(0,1),\infty(1,1) & \infty(0,2),\infty(1,2),\infty(2,2) & 0 \\ 0 & \infty(1,1) & \infty(0,2),\infty(1,2),\infty(2,2) & 0 \\ 0 & \infty(0,1),m(1,1),\infty(1,1),w(2,1) & \infty(0,2),\infty(1,2),\infty(2,2) & m \end{pmatrix} \end{array}.$$

But not all variables are problematic. There is no dependency between variables X2 and X5, and the problematic X4, as indicated by the 0-flows at row X4. Reasoning about variable X3 is more complicated due to the ∞-flows in column X3. ◻

A baseline determination of program derivability is simple by Algorithm 1. In the negative case, at least one variable must be a source failure. It is critical to recognize when a variable causes derivation failure, it fails in every derivation. To identify failing variables, it suffices to evaluate variables individually for at most p-coefficients. A variable cannot be bounded if no choice vector exists. Among the remaining variables, if the mwp-matrix permits concluding disjointness from failing variables, we query the mwp-bound as described in Sect. 4.2.

▶ **Example 11** (cont.)**.** By its choice vector, $\big(\{0,1,2\},\{2\},\{0,1,2\}\big)$, at index 1 variable X3 permits only one derivation choice: 2. Applying the choice to the matrix column X3 produces the coefficients: $\mathtt{X2}(2,1) \mapsto w$ and $\mathtt{X3}(2,1) \mapsto m$ and $\mathtt{X4}(2,1) \mapsto 0$ and $\mathtt{X5}(2,1) \mapsto w$. By the

0-flow, variable X3 is independent of the failing variable X4. Variable X3 can be bounded by at most a polynomial in inputs. □

5 mwp-bounds as postconditions

The enhanced flow calculus now gives a postcondition inference for free – the mwp-bounds are postconditions. The only task left is to specialize the technique to the use case. We want to compute postconditions for individual variables, if they exist, and find the optimal (least) postconditions they admit.

Although we have developed a way to query mwp-bounds by form, two concerns remain. First, mwp-bounds cannot be totally ordered since certain bound-forms are incomparable. Second, mwp-bounds of different form can evaluate to the same numeric polynomial. For example, the three mwp-bounds $W_1 \equiv \max(0, \mathtt{X1} + \mathtt{X2}) + 0$ and $W_2 \equiv \max(\mathtt{X1}, 0) + \mathtt{X2}$ and $W_3 \equiv \max(\mathtt{X2}, 0) + \mathtt{X1}$ are all numerically equal to $\mathtt{X1} + \mathtt{X2}$. Therefore, we need an alternative approach to reason about optimality.

5.1 Optimal mwp-bounds by form

To establish ordering on mwp-bounds, we leverage two built-in features of the mwp-calculus. The individual coefficient are ordered $0 < m < w < p < \infty$. Furthermore, the mwp-bounds carry semantic meaning by *form*. The growth of a variable value is at most linear (resp.iteration-independent, iteration-dependent) if its mwp-bound contains at most m (resp. w, p) coefficients. This gives sufficient justification for our definition of optimality.

▶ **Definition 12** (Optimality). *We define an order on mwp-bounds by form, by the* maximal coefficient *it contains: 0-bound < m-bound < w-bound < p-bound < ∞ (none). A variable's mwp-bound is optimal if it is the minimal bound in the order.* □

For example, among W_1, W_2, and W_3, the w-bound $\max(0, \mathtt{X1} + \mathtt{X2}) + 0$ is optimal because it contains no p-coefficients. In turn, it provides evidence that the variable's value growth is independent of loop iteration. The other two candidates are too weak to establish the same conclusion. Like derivability, it suffices to categorize a variable as having a specific kind of bound if there exists a derivation admitting the bound.

5.2 Variable postcondition search

Assuming a program is derivable, or a variable is disjoint from failure, a general procedure for deriving a variable's postcondition is as follows.

1. Run the evaluation procedure of Algorithm 1, iteratively, on the derivation choices of monomials whose coefficient is greater than $(0, m, w, p)$. A solution exists for variables that are not associated with failure.
2. Stop once the evaluation procedure returns a choice vector; the first solution is optimal.

The first choice vector defines precisely the derivation choices by which the optimal variable bounds can be located. Since the procedure is decomposed into individual variables, a natural question is whether optimal variable bounds exist concurrently. The answer can be found by taking the intersection of choice vectors. If the intersection is nonempty, it defines the derivations where mwp-bounds of variables occur concurrently. Related questions about mwp-bounds can be formulated similarly as operations on choice vectors.

▶ **Definition 13** (Choice vector intersection). *Letting* $\vec{C_a} = (a_1, a_2, \ldots, a_k)$ *and* $\vec{C_b} = (b_1, b_2, \ldots, b_k)$ *be choice vectors of length* k*, we define the intersection of* $\vec{C_a}$ *and* $\vec{C_b}$ *as*

$$\vec{C_a} \cap \vec{C_b} = \begin{cases} (a_1 \cap b_1, a_2 \cap b_2, \ldots, a_k \cap b_k), & \text{if } \nexists\, i \text{ such that } a_i \cap b_i = \varnothing, \\ \varnothing, & \text{otherwise.} \end{cases}$$ ◻

5.3 Program analysis for postcondition inference

Obtaining a technique for postcondition inference requires adjusting the base language of Sect. 3.1 to a language of loops. A *loop program* is a command `while` or `loop`, whose body is a command `C` in the base language. We compute postconditions for all variables whose mwp-bounds are expressible in the mwp-calculus. Since the analysis is compositional, the preferred order of processing is by depth, from loop nests to parent. This permits composing matrices of the nests with the parent and omitting repeated analysis. Given a loop program P, the inference proceeds as follows.

1. Extract all loops from the program P.
2. For each loop l:
   i. Run the mwp analysis to derive the mwp-matrix, $l : M$.
   ii. Using Algorithm 1, evaluate M to determine if l is derivable.
      ▷ If yes: mark every variable as satisfactory.
      ▷ If no: mark the variables disjoint from failure as satisfactory, cf. Sect. 4.3.
   iii. Evaluate satisfactory variables for optimal postconditions, cf. Sect. 5.2.
   iv. Record the postconditions of satisfactory variables.
3. Return the analysis result for P.

5.4 Postconditions as descriptors of variable behavior

Focusing on language of loops restricts the encountered computations. It is worthwhile to clarify how this change affects variable postconditions.

- *Linear* mwp-bounds are assigned to variables that do no change, are targets of direct assignments (without arithmetic), or are modified by constant factors. Inside loops, other operations are "too strong" to retain linear behavior.
- *Iteration-independent* mwp-bounds are not loop invariant. Rather, iteration-independence is better understood as a quasi-invariance property. A variable is iteration-independent if its final value depends on a fixed number of iterations (e.g., the last one), or eventually reaches a fixpoint. Iteration independence implies that beyond the fixpoint the variable is unaffected by an increase in loop iteration count.
- *Iteration-dependent* mwp-bounds capture patterns of arithmetic computations involving multiple variables. They can occur in `loop` commands, but not in `while` commands; this is a built-in restriction of the mwp-calculus. It reflects the intuition that the calculus over-approximates `while` loops to iterate infinitely, whereby arithmetic operations involving polynomial p-coefficients cannot be bounded soundly.
- *Inconclusive* ∞-result marks the absence of an expressible postcondition. It characterizes variables that fall outside the previous three cases, and deductively informs that a variable's value growth is either beyond a polynomial or challenging to identify.

Since one variable can be assigned multiple incomparable bounds, the definition of optimality does not guarantee the numerical minimum. Consider, for example, variable `X3` in *LucidLoop*. It is assigned mwp-bounds $W_1 \equiv \max(\texttt{X3}, \texttt{X2}) + \texttt{X1} \times \texttt{X5}$ and $W_2 \equiv \max(\texttt{X3}, \texttt{X2} + \texttt{X5})$ and $W_3 \equiv \max(\texttt{X3}, \texttt{X5}) + \texttt{X1} \times \texttt{X2}$. Assume that $\texttt{X1} > 0$ for the loop to iterate. Excluding W_2 would leave two alternatives that can evaluate to distinct numeric values, yet both are equally optimal by definition. However, although possible in theory, it is uncertain if such scenario occurs in natural loops.

6 Implementation and evaluation

6.1 Postcondition inference with mwp_ℓ

We chose to implement our postcondition inference as an extension of the static analyzer pymwp. pymwp [4] is an open source Python implementation of the flow calculus of mwp-bounds for C language. By default, pymwp takes as input a C file and analyses the variable value growth in each function.

Our implementation is a new loop analysis mode, mwp_ℓ, integrated into the pymwp analyzer. The loop analysis is complementary to the default function analysis mode, which we name mwp_f for distinction. The primary differences are that mwp_ℓ looks for *optimal bounds by variable* in a loop, and mwp_f finds existence of *any bound for all variables* in a function. Though applicable to both, the theoretical advancements presented in this paper are implemented only in mwp_ℓ to permit experimentally evaluating the impact.

The postcondition inference required mapping the procedure described in Sect. 5.3 to C language. mwp_ℓ supports all C language loops. **`while`** and **`do`**...**`while`** follow immediately from the theoretical `while`. A `loop` corresponds to **`for`**, but the implementation was more challenging. A loop requires the form "do `C` `X` times", with a singular guard variable `X` that does not occur in the body command `C`. mwp_ℓ checks that a C language **`for`** matches this form and ignores it otherwise. The restriction is a minor inconvenience in practice since incompatible **`for`** loops can be refactored to the expected form by introducing fresh variables. Loop-related jump statements and verification macros are supported and treated as `skip`. For soundness, pymwp must be run with a `--strict` flag. It ensures analyzed C fragment is fully expressible in the base language before inferring postconditions.

If we want to insert the analyzer results as concrete verification assertions, two additional steps are required. These are recording the initial variable values, since the initial values are needed to express the postconditions; and defining the constants left implicit in mwp-bounds. We demonstrate these steps in the experiments and through our artifact.

6.2 Experiment design

Our experiments are guided by three research questions.

1. **How effective is our technique at discovering postconditions in general?** We run mwp_ℓ on two standard benchmark suites from loop invariant inference literature. These suites contain versatile challenges independent of the applied inference technique.
2. **How does our technique compare to leading automatic inference approaches?** Though approximative and lightweight, we hypothesize our technique is useful for assisting software engineers in specifying postconditions. To establish this, we run experiments on the state-of-the-art invariant detector, Daikon. We then compare the inferred postconditions, and assess analyzer differences and utility in supporting specification tasks.

Table 1 Benchmarks characteristics summarized. Column n is the min–max of each metric, and mean $\overline{x}$ is relative to benchmark count (#). Single benchmark can contain multiple nested loops. Variable count includes loop guards and variables occurring in loop bodies.

Suite (#)	**linear** (49)			**mwp** (30)			**nonlinear** (37)		
Description	n	total	$\overline{x}$	n	total	$\overline{x}$	n	total	$\overline{x}$
Lines of code	8–29	650	13.27	6–16	271	9.03	9–41	694	18.76
Loop count	1–1	49	1.00	1–1	30	1.00	1–3	48	1.30
Loop variables	1–6	117	2.39	2–6	105	3.50	2–21	208	5.62
Loop types	✓`while`	−`for`	−nests	✓`while`	✓`for`	−nests	✓`while`	−`for`	✓nests

3. **What is the impact of the introduced theoretical enhancements?** We compare mwp_ℓ and mwp_f in terms of quantity and optimality of the variable bounds they produce. We use a benchmark suite designed to pose challenges to mwp-based flow analyses.

6.3 Experiment setup

Benchmarks. Our experiments use the three micro-benchmark suites summarized in Table 1. **1.** The *linear* "Code2Inv" suite [42], and **2.** *nonlinear* suite [32, 47] contain pre-annotated loop programs with respective inductive invariants. The suites have appeared in multiple prior invariant inference evaluations [32, 39, 42, 46, 47]. The benchmarks range from single-variable loops to classic algorithms, e.g., geometric series and divisor computations. For the linear suite, we unified benchmarks that are identical for postcondition inference[3]. **3.** The *mwp benchmarks* suite [3, 4] contain unannotated programs with complex data flows, variable dependencies, and arithmetic operations. We modified the benchmarks to avoid nested loops because mwp_ℓ and mwp_f scope them differently, and omit loopless benchmarks. All suites contain branching statements and nondeterministic control-expression, simulating external function calls. We lifted local variables and declarations to inputs, to reduce contextual information per our problem formulation; and expanded n-ary expressions to binary form to accommodate pymwp.

Verification and ground truth. Motivated by the discovery that the original linear suite contained nine invalid benchmarks [39, Appendix G], we verify all benchmarks in Dafny [24]. This has multiple benefits: it ensures the assertions we make of benchmarks are provably correct; it provides a ground truth for our experiments, and resounds an alarm of the issue[4]. Our verification efforts of the linear suite confirmed exactly nine invalid instances—the proofs are included in our artifact—and uncovered one type-related integer overflow. In our experiments, we exclude the invalid benchmarks and fix the type issue. Although we could have used other verification tools, we selected Dafny to emphasize our postcondition inference is not specific to C, and generalizes to imperative paradigm.

Comparison target. We evaluate mwp_ℓ against Daikon [15], a mature, open-source dynamic invariant detector. It has front-ends to support numerous programming languages and data source inputs. Using execution traces and templates, Daikon predicts likely invariants, based on input samples and a confidence level, at various program points. Daikon requires a sufficiently large set of samples to produce valid and interesting results. These outcomes

[3] Programs with same precondition and loop, and ones that differ only by unused variables, are identical.
[4] The suite continues to be used without modification in experiments postdating the discovery, e.g., [45].

are partially manageable by configuration options. For our experiments, we provide four input samples of each benchmark, set the confidence level to 0.5, and defer to defaults otherwise. Daikon differs significantly from mwp_ℓ in behavior, software features, and scope of detectable invariants. However, we choose it as a comparison target because it supports local/offline postcondition inference, handles C language inputs, and is sufficiently actively maintained [1]. These three are minimally necessary for comparison experiments. Daikon is among the few analyzers we know of to meet the combination of criteria.

Metrics. The pymwp and Daikon analyzers differ in tracked information and results, therefore we define the experiment metrics by case. In experiments involving mwp_ℓ or mwp_f, we measure whether a postcondition (an mwp-bound) is discovered, and if yes, manually compare it to the ground truth. We record all other meta-data collected by the analyzer, including loop and variable counts and analysis time. The time reflects pure analysis time and excludes parsing. All experiments use the `--strict` flag, meaning benchmarks with unsupported syntax are not analyzed. For experiments involving Daikon, we print and record the inferred `EXIT`–invariants (postconditions) and calculate traces sizes. Daikon defines separate commands for inference and printing of invariants to accommodate multiple output formats. We measure Daikon time as latency to run the inference command only, without printing. In absence of built-in support, we use Linux `time` to obtain the metric. Besides time, the metrics are deterministic. Since we measure time by just one execution, it should be treated as relative and referential. In particular, times between the two analyzers are incomparable because they are measured differently.

We ran all experiments on commodity hardware, on a 2-core Ubuntu 20.04 Linux/amd64 virtual machine with 16 GB of random-access memory. Complete experiment resources, software version details, and replication instructions will be made available as a public artifact. The artifact supports completely all results presented in Sect. 7.

7 Analysis of experiment results

7.1 Research question findings

Inference generalizability. On the linear suite mwp_ℓ finds bounds for 87% of loop variables, cf. Table 2. Despite the suite name, some variables' growth rates exceed polynomial bounds. Such variables are not expressible in the mwp-calculus, which explains the missed cases. The linear suite contains simple data flows, therefore mwp_ℓ finds postconditions nearly instantly. The nonlinear suite of complex arithmetic is more challenging. Since all benchmarks are `while` loops, mwp_ℓ identifies m- and w-bounds, but no p-bounds, as expected. Six loops are not analyzable because they contain unsupported syntax, particularly division operators. Among the 89% of loops that are, mwp_ℓ finds postconditions for 69% of variables. The result is encouraging because the nonlinear benchmarks represent real algorithms. The missed variables either exceed polynomial growth bounds or are not expressible. Although the benchmarks are difficult, the analysis time remains modest compared to the mwp suite. It suggests the computations that are challenging to mwp-based analyses are not prevalent in these natural algorithms.

Comparison study with Daikon. We must first recognize that running static and dynamic analyzers is different. In the static case, we analyze C program *fragments* directly from syntax. Obtaining results with Daikon requires up to five additional steps: constructing ideally multiple syntactically complete programs, compilation to machine code, execution to obtain traces, running the invariant detector on the traces, and printing the postcondi-

Table 2 Analysis results for mwp_f and mwp_ℓ, with totals and (mean) of experiment metrics. *Loops* is the number of analyzed loops including loop nests. *Bounds* is the number of inferred postconditions with a mean relative to analyzed loop variables, *vars*. The *mwp*-columns show the breakdown of the postconditions by form. Column ∞ is the number of unbounded variables.

Analyzer	Suite	Loops	Vars.	Time, ms	Bounds	m, w, p	∞
$\mathbf{mwp}_\ell$ (ours)	linear	49 (1.0)	117	179 (3.65)	102 (.87)	101, 1, 0	15 (.13)
	nonlinear	42 (.89)	180	832 (17.33)	124 (.69)	116, 8, 0	56 (.31)
	mwp	30 (1.0)	105	3,741 (124.70)	77 (.73)	45, 28, 4	28 (.27)
$\mathbf{mwp}_f$	mwp	30 (1.0)	105	3,973 (132.43)	60 (.57)	32, 20, 8	45 (.43)

Table 3 Daikon analysis results, with totals and (mean) of experiment metrics. Daikon finds postconditions in procedures, which increases the variable counts from Table 1 for some benchmarks. We show the total procedure-scoped variables in *vars*. The total inferred postconditions are shown in *postcond*. Because multiple likely postconditions may be inferred for the return variable of a single procedure, the mean of postconditions is relative to benchmark count. *Missed* column is the least count of variables that do not occur in any postcondition.

Analyzer	Suite	Trace lines	Time, ms	Vars.	Postcond.	Missed
Daikon	linear	14,055 (286.84)	50,578 (1032.20)	131	96 (1.96)	22 (.17)
	nonlinear	10,878 (294.00)	38,192 (1032.22)	208	154 (4.16)	55 (.26)
	mwp	5,796 (193.20)	30,594 (1019.80)	105	64 (2.13)	22 (.21)

tions. One obvious benefit is trace-based analysis can be applied independent of source programming language. We expected Daikon to find precise invariants on the numerical benchmarks, but the results failed to confirm the hypothesis, cf. Table 3. Daikon found likely invariants for all linear and nonlinear benchmarks, but missed 3 benchmarks in the mwp suite. In comparison, mwp_ℓ found postconditions for these missed benchmarks, but missed 3 others in the same suite. However, a simple comparison of postcondition counts is insufficient to characterize the analyzers differences. Significant behavioral distinctions concern the analysis scope, postcondition correctness, and control over output. **1.** Daikon generates output at procedure exit and entry points, not within a procedure [1]. Therefore, postconditions of local variables are not analyzable, and it was necessary to add return statements to every benchmark. Daikon postconditions always involve the return variable. Variables that do not influence the return variable are excluded from postconditions; these are recorded as *missed* in Table 3. mwp_ℓ does not have the same restrictions, as it gives postconditions for all expressible variables at loop termination, independent of a return statement. **2.** Since Daikon finds likely invariants, additional effort is required to determine correctness of the invariants. For example, through our verification of the linear suite, we found 30 of the 96 postconditions are immediately correct, but 48 require introducing new assumptions to the context, and 18 are generally incorrect as they hold for isolated inputs only. Similar postprocessing is not required by mwp_ℓ, though it does require identifying omitted constants. **3.** The utility of automatic inference decreases if the results are spurious or excessive in quantity. Daikon has configuration options, e.g., confidence level, to reduce noise, but in principle it has no explicit upper bound on how many postconditions it generates. For example, Daikon generates 1–9 postconditions per each return variable of the nonlinear suite. mwp_ℓ has a strict limit on generated output, defined by the count of loop variables. In an ideal case, mwp_ℓ gives a 1:1 ratio of postconditions to variables.

Impact of theoretical enhancements. Our results on the mwp suite, in Table 2, confirm mwp_ℓ outperforms mwp_f at discovering variable postconditions; and by inspection of the individual bounds, finds equal or improved postconditions for every bounded variable. The analysis expressiveness has strictly improved. We confirmed postconditions correctness by verifying them in Dafny. mwp_ℓ and mwp_f operate similarly in two phases: first they construct a matrix, then evaluate the matrix for variable bounds. The primary source of latency is the construction phase, which remains unaffected by the proposed enhancements. Therefore, the difference in analysis time is expectedly insignificant. mwp_ℓ and mwp_f differ in how they perform matrix evaluation. mwp_f determines satisfactory derivations, and if at least one exists, returns an arbitrary bound for all variables. Only mwp_ℓ implements the enhancements of this paper: it locates the optimal variable bounds, which explains its superior results. The proposed evaluation strategy is practically efficient, as compared to mwp_f, it adds no notable performance overhead.

7.2 Precision of approximative postconditions

Beyond performance, we are curious about our technique's utility in assisting engineers in software verification tasks. Because postcondition quality aspects are observable in the summarized data, as a case study, we inspect an arbitrary benchmark from the nonlinear suite. We give additional examples for comparison in Table 4, with the corresponding benchmarks included in Appendix A. As part of our evaluation, we verified the benchmarks using mwp_ℓ in the process, which grounds our findings in experience.

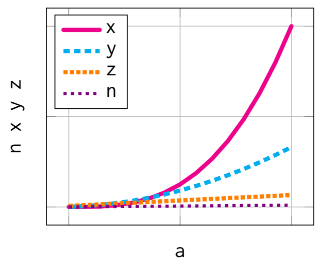
▶ **Example 14** (Cohen's cubes)**.** Listing 3 shows a nonlinear benchmark to generate consecutive cubic values. Its sample execution traces and variable values growth are depicted aside.

Listing 3 The cohencu benchmark

```
assume(n==0∧x==0∧y==1);
assume(z==6∧a≥n);
while (n<a) {
  n=n+1;
  x=x+y;
  y=y+z;
  z=z+6; }
assert(x==a*a*a);
```

Postcondition traces

a	n	x	y	z
0	0	0	1	6
1	1	1	7	12
2	2	8	19	18
3	3	27	37	24
10	10	1000	331	66



The predefined postcondition is $\mathsf{x} = \mathsf{a} \times \mathsf{a} \times \mathsf{a}$. The other variables have final values: $\mathsf{n} = \mathsf{a}$, $\mathsf{y} = 3\mathsf{a} \times \mathsf{a} + 3\mathsf{a} + 1$, $\mathsf{z} = 6\mathsf{a} + 6$, and variable a is unchanged from its initial value. The postconditions inferred by Daikon are $\mathsf{x} \ \mathsf{mod} \ \mathsf{a} = 0$ and $\mathsf{x} > v$, w.r.t. the initial value of v, where v in $\{\mathsf{a}, \mathsf{n}, \mathsf{y}, \mathsf{z}\}$. The Daikon results are correct if we introduce the precondition $\mathsf{a} \geq 2$. The mwp_ℓ result determines variables a, n, and z grow at most linearly in initial values, and variables x and y are assigned ∞. Although coarse, the simplicity hides layers of utility. We know the relations on linear variables are straightforward to define and prove; more attention is needed to reason about x and y. We obtain exactly one sound judgement for each variable, instead of multiple uncertain suggestions. The mwp-bounds are maximally optimal given the expressiveness of the mwp-calculus and obtained rapidly from syntax alone. To an engineer, this provides insight of the program behavior and guides allocation of extended verification efforts. ◻

Table 4 Postcondition comparison on select benchmarks, cf. Appendix A. We determined the precise expressions manually assuming loops iterate; otherwise no variables change. The expressions refer to variables at different program points. We use ' to denote a variable holding its final value, otherwise a variable holds its initial value. For the likely postconditions inferred by Daikon, we append symbol * to postconditions that require additional assumptions to prove, and a symbol † when a postcondition does not generalize.

Benchmark	mwp_ℓ	Daikon	Precise

example 3.4
LucidLoop
	linear: `X1`, `X2`, `X5`
	`X3'` $\leq \max($`X3`,`X2` + `X5`$)$
	`X4'` $\leq$ `X4` + `X1` $\times$ `X5`
		`X3'` > `X2`*
		`X3'` > `X4`†
		`X3'` > `X5`*
			`X1`, `X2`, `X5` – no change
			`X3'` = `X2` $\times$ `X2` + `X5`
			`X4'` = `X4` + `X1` $\times$ `X5`

not-infinite #4	linear: `X3`, `X5`
	`X1'` $\leq \max($`X1`,`X2` + `X3`$)$
	`X2'` $\leq \max($`X2`,`X3`$)$
	`X4'` $\leq \max($`X4`,`X5`$)$
		—	`X3`, `X5` – no change
			`X1'` = 2 $\times$ `X2` $\vee$ `X1'` = 4 $\times$ `X3`
			`X2'` = 2 $\times$ `X3`
			`X4'` = 2 $\times$ `X5`

linear #5	linear: `size`, `x`, `z`
	`y'` $\leq \max($`y`,`z`$)$
		`y'` $\leq$ `y`*
		`y'` $\leq$ `z`*
			`size`, `z` – no change
			`x'` = `size`
			`y'` = `y` $\vee$ `y'` = `z`

linear #97/98	linear: `i`, `x`, `y` `j'` : ∞	`j'` > `i`* `j'` − `x` − 2 = 0† `j' mod y` = 0 `j'` > `y`*	`x`, `y` – no change `i'` = `x` + 1 `j'` = `jx` + `j` = 2`x` + 2

Based on the observations and our experience, mwp_ℓ and Daikon target different problems with complementary applications. Daikon is by design better suited for analysis of complete running systems with heap-based manipulations. mwp_ℓ assists engineers during the programming phase, where complete details of programs are still unknown or under development. mwp_ℓ could, for example, be converted to a plugin in the Frama-C ecosystem [10].

8 Related works

Our work is primarily related to specification inference for software verification. Automatic specification inference, whether from program syntax or natural language, is a challenging problem [14, 47, 48]. A common first step is a conceptual "divide-and-conquer": breaking down the problem into parts—preconditions, postconditions, and inductive invariants—restricts the problem space and enables developing increasingly better partial solutions. The literature is rich in techniques for loop invariant detection. Seminal works and prominent techniques have been developed, for example, based on abstract interpretation [12, 11, 27, 23], constraint solving [9], interpolation [25, 21, 26], counterexample-guided abstraction refinement [8], logical abduction [14], symbolic execution [33], SMT and model-checking [17, 37, 44], and machine learning [15, 42, 39, 46, 47]. These techniques can be further distinguished by the form of invariants they generate: linear inequalities [14], polynomials [40, 34, 32], array properties[19, 34], etc.

In contrast, existing works on postcondition inference are rare. The one closest to ours is an abstract interpretation-based analysis by Popeea and Chin [35]; though, abstract interpretation differs considerably from the mwp analysis. Similar to ours, the Popeea and Chin analysis applies static reasoning to mathematical program abstractions of an imperative core language; but, it also has additional features like configurable analysis precision and recursion support. Measuring the impact of these theoretical differences would have been the ideal research question for our experiments. Unfortunately, the Popeea and Chin analysis appears at a different stage of progression from theory to practice, making such comparison impossible. Instead, we considered three alternatives from dynamic analysis. EvoSpex [28] is a postcondition analysis, but designed for Java methods (accessors, mutators, heap structures, etc.) and thus distant in aims from our loop analysis. The numeric invariant generator DIG [34] is a close relative of Daikon: it can perform inference at different program points, making it usable for postcondition inference. It would have been a suitable comparison target, though we chose Daikon [15] for its maturity. Compared to our static technique, dynamic analyses in general have operational differences similar to those observed in Daikon, cf. Sect. 7.1.

Recently, the strategy to treat specification conditions as separate problems has garnered criticism on the basis it limits abilities to achieve fully-automatic proof construction. This argument then grounds the motivation for using large language models to infer specifications for full proofs [45]. However, the approach suffers from the *assertion inference paradox* [16]. Briefly, since the goal of program verification is to prove correctness—that an implementation satisfies its specification—it requires having both elements and assessing one against the other. The core problem is, if we infer the specification from the implementation the fundamental property of independence is lost between the mathematical property to be achieved and the software that attempts to achieve it. To counter the paradox, not all specification conditions should be inferred automatically. The mwp-bounds are approximative by design and meant to *assist* the verification process, thus we intentionally maintain human oversight and avoid the paradox.

Beyond software verification, our postcondition inference strengthens the connection between complexity theory and verification. Whereas in [32] complexity results are obtained from loop invariants, we obtain specification conditions from complexity-theoretic origins. The bidirectionality suggests extended exploration of the connection is warranted.

9 Conclusion and future directions

Specifications are essential to formal methods and inference facilitates their automatic discovery. In this paper, we have proposed a complexity-theoretic analysis for inferring partial specifications, namely postconditions. This result required four new enhancements: projecting mwp analysis on individual variables, improving derivation failure-handling, an evaluation strategy to obtain optimal mwp-bounds, and applying the analysis to a new use-case. We believe our technique offers complementary strengths among the related approaches by the kind of postconditions it computes—sound, variable-specific, and approximative—and how it arrives to those results through a lightweight, static compositional syntactic analysis without external solvers. These claims are supported by our implementation, mwp_ℓ, and experimental results.

Our postcondition interference builds on the flow calculus of mwp-bounds. Although we have extended the capabilities of the analysis, multiple future improvements remain, some of which emerge from this work. The two main directions are enriching the expressiveness of the core language to cover more programs and improving the analysis precision. Concrete improvements include the following.

- For precision, the calculus should leverage assumptions when available, track immutability of variables, and account for the variables in control expressions of `while` loops. We expect the last two to be straightforward to achieve.

- Currently, p-bounds cannot occur in `while` loops; this is a categorical restriction of the mwp-calculus. Discovering ways to relax this restriction would improve expressiveness and permit discovery of more postconditions.
- The analysis purposely omits constants for efficiency, but constants are needed for precise assertions. Investigating how constant tracking could be introduced could take inspiration from [5]. In turn, it could uncover program optimization opportunities. For example, in the running example, the postcondition of variable `X4` is precise. A confirmation of this fact would allow lifting `X4` outside the loop.
- On the practical side, our analysis does not cover division operator and required expanding operations to binary form. Currently, these limitations can be resolved by refactoring the input program, but should be resolved at the theoretical level.

We are encouraged by the continued enhancements to the flow calculus of mwp-bounds and uncovering its extended utility. In its current state, the analysis could be implemented as a developer plug-in to assist writing formal specifications. In future research, we will consider extensions to the flow calculus capabilities. We are curious if similar solver-free syntactic analyses could be designed to infer other specification conditions or invariants more broadly.

References

1. The Daikon invariant detector, 2024. URL: https://plse.cs.washington.edu/daikon.
2. Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3Test: Assertion-Augmented Automated Test case generation. *Information and Software Technology*, 176:107565, December 2024. doi:10.1016/j.infsof.2024.107565.
3. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *LIPIcs*, pages 26:1–26:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.FSCD.2022.26.
4. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. pymwp: A Static Analyzer Determining Polynomial Growth Bounds. In *Automated Technology for Verification and Analysis*, , pages 263–275. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-45332-8_14.
5. Amir M Ben-Amram and Geoff Hamilton. Tight Polynomial Worst-Case Bounds for Loop Programs. *Logical Methods in Computer Science*, 16(2):4:1–4:39, May 2020. doi:10.23638/LMCS-16(2:4)2020.
6. Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 467–478. ACM, 2015. doi:10.1145/2737924.2737955.
7. Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. *Deductive Verification of Smart Contracts with Dafny*, page 50–66. Springer International Publishing, 2022. doi:10.1007/978-3-031-15008-1_5.
8. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003. doi:10.1145/876638.876643.
9. Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. Linear Invariant Generation Using Non-linear Constraint Solving. In *Computer Aided Verification*, pages 420–432. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-45069-6_39.
10. Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. Frama-C user manual, 2010. URL: https://www.frama-c.com/download/frama-c-user-manual.pdf.

11 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.

12 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96. ACM Press, 1978. doi:10.1145/512760.512770.

13 Ugo Dal Lago. *A Short Introduction to Implicit Computational Complexity*, pages 89–109. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-31485-8_3.

14 Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *ACM SIGPLAN Notices*, 48(10):443–456, October 2013. doi:10.1145/2544173.2509511.

15 Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, December 2007. doi:10.1016/j.scico.2007.01.015.

16 Carlo Alberto Furia and Bertrand Meyer. *Inferring Loop Invariants Using Postconditions*, pages 277–300. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-15025-8_15.

17 Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with BLAST. In *Model Checking Software*, pages 235–239. Springer Berlin Heidelberg, 2003. doi:10.1007/3-540-44829-2_17.

18 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.

19 Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant Generation in Vampire. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2011, pages 60–64. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-19835-9_7.

20 Reiner Hähnle and Marieke Huisman. *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools*, page 345–373. Springer International Publishing, 2019. doi:10.1007/978-3-319-91908-9_18.

21 Ranjit Jhala and Kenneth L McMillan. A practical and complete approach to predicate refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2006, pages 459–473. Springer Berlin Heidelberg, 2006. doi:10.1007/11691372_33.

22 Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Transactions on Computational Logic*, 10(4):28:1–28:41, August 2009. doi:10.1145/1555746.1555752.

23 Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976. doi:10.1007/BF00268497.

24 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-17511-4_20.

25 Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136. Springer Berlin Heidelberg, 2006. doi:10.1007/11817963_14.

26 Kenneth L. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification*, pages 104–118. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14295-6_10.

27 Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, March 2006. doi:10.1007/s10990-006-8609-1.

28 Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1223–1235. IEEE, 2021. doi:10.1109/ICSE43902.2021.00112.

29 Anders Møller and Michael I Schwartzbach. Static Program Analysis, 2024. URL: https://cs.au.dk/~amoeller/spa/.

30 Jean-Yves Moyen. Implicit Complexity in Theory and Practice, 2017. URL: https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf.

31 Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer Berlin Heidelberg, 2016. doi:10.1007/978-3-662-49122-5_2.

32 ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'17, pages 605–615. ACM, 2017. doi:10.1145/3106237.3106281.

33 ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. SymInfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, , pages 804–814. IEEE, 2017. doi:10.1109/ase.2017.8115691.

34 Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. DIG: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–30, September 2014. doi:10.1145/2556782.

35 Corneliu Popeea and Wei-Ngan Chin. Inferring Disjunctive Postconditions. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, pages 331–345. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-77505-8_26.

36 Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, March 1953. doi:10.1090/S0002-9947-1953-0053041-6.

37 Daniel Riley and Grigory Fedyukovich. Multi-phase invariant synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '22, pages 607–619. ACM, 2022. doi:10.1145/3540250.3549166.

38 D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995. doi:10.1109/32.341844.

39 Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *8th International Conference on Learning Representations, ICLR 2020*, . OpenReview.net, 2020. URL: https://openreview.net/forum?id=HJlfuTEtvB.

40 Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL04, pages 318–329. ACM, 2004. doi:10.1145/964001.964028.

41 Fabian Schiebel, Florian Sattler, Philipp Dominik Schubert, Sven Apel, and Eric Bodden. Scaling Interprocedural Static Data-Flow Analysis to Large C/C++ Applications: An Experience Report. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *LIPIcs*, pages 36:1–36:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ECOOP.2024.36.

42 Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31*, volume 31 of , pages 7762–7773. NeurIPS, 2018.

43 Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. Formal Methods in Industry. *Formal Aspects of Computing*, August 2024. doi:10.1145/3689374.

44 Hari Govind Vediramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. *Formal Methods in System Design*, 63:81–109, 2024. doi:10.1007/s10703-023-00412-3.
45 Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *Computer Aided Verification*, pages 302–328. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-65630-9_16.
46 Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, pages 106–120. ACM, 2020. doi:10.1145/3385412.3385986.
47 Shiwen Yu, Ting Wang, and Ji Wang. Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '23, pages 175–187. ACM, 2023. doi:10.1145/3597926.3598047.
48 Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2S: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20. ACM, November 2020. doi:10.1145/3368089.3409716.
49 Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'15. ACM, August 2015. URL: http://dx.doi.org/10.1145/2786805.2786858, doi:10.1145/2786805.2786858.

A Benchmarks

Benchmarks for postconditions in Table 4.

Listing 4 mwp/example 3.4

```
int foo(int X1, int X2, int X3,
        int X4, int X5) {
    for (int i = 0; i < X1; i++) {
        X3 = X2 * X2;
        X3 = X3 + X5;
        X4 = X4 + X5;
    }
    return X3;
}
```

Listing 5 mwp/not infinite #4

```
int foo(int X1, int X2, int X3,
        int X4, int X5) {
    while (rand()) {
        X1 = X2 + X2;
        X2 = X3 + X3;
        X4 = X5 + X5;
    }
    return X4;
}
```

Listing 6 Linear #5

```
int foo(int x, int y, int z,
        int size) {
    while (x < size) {
        x = x + 1;
        if (z <= y)
            y = z;
    }
    return y;
}
```

Listing 7 Linear #97/98

```
int foo(int i, int j, int x, int y) {
    assume(y == 2);
    while (i <= x) {
        i = i + 1;
        j = j + y;
    }
    return j;
}
```