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 rarely been the targeted focus of prior investigations. In this paper, we present a static program analysis for automatically inferring postconditions from program syntax. Our solution is a compositional and sound technique for bounding variable values in numerical loops. For each variable, it finds one approximative postcondition, of at most polynomial form, if a postcondition is expressible in the underlying theory. The technique is applicable in imprecise contexts, on program fragments, 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 or under development.

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.CVIT.2016.23

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 loop invariants, reviewed in Sect. 7—the strategies to intentionally infer postconditions have received less attention, with a few exceptions [36, 29]. The study is warranted because postconditions assist discovery of other verification conditions [18]. As generalized assertions, postconditions benefit other software development activities, like testing [4, 47] and maintenance [38].

Unfortunately, defining formal specifications is non-trivial and requires expertise. Specifications must be consistent, precise, and complete descriptions of a program's functional

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:27
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

behavior. When expressed in natural language, ambiguity and omissions arise. Existing techniques, for example in automated resource analysis [24, 9], already compute varied notions of postconditions as means to obtain their main result. Therefore, it seems intuitive that such techniques could support software engineers in specification tasks. This is precisely the key intuition we exploit in this paper.

Our technique is a syntactical flow calculus, inspired by complexity theory, that automatically infers approximative variable postconditions in numerical loops. Our technique is sound, such that if a postcondition can be inferred, we know it to hold. In this view, a program is assumed to implement its intended behavior, but misses a formal proof. Then, pairing our solution with complementary inference techniques and verification tools leads to full formal guarantees.

1.1 Problem formulation and overview

We analyze deterministic imperative numerical loops with the goal to automatically infer variable 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 control expressions, control flow, 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. A program may be just a *fragment* still under development; not necessarily an executable whole-program. 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 that 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 express the growth of variable values within the analyzed loop. Then, the postconditions assist discovery of complete and precise specifications and permit formal verification. □

Our analysis derives mwp-bounds, a kind of value growth bounds of program variables. An *mwp-bound* [24] represent a variable's *final value*, relative to the *initial variable values*, and *omits constants*. A critical idea is that the **mwp-bounds are postconditions**. An mwp-bound is a symbolic formula, of at most polynomial form, that approximates the upper bound of the final value. Since the problem formulation contains no preconditions, a function in terms of the inputs is the most precise achievable postcondition, in most cases. Variables whose value growth is beyond polynomial are not expressible as mwp-bounds. However, our experiments in Sect. 6.3 suggest that this is not a prohibitive limitation; the technique can bound variables at a rate 69% or above, across a diverse set of benchmarks.

Based on the mwp-bound form, we categorize variables' value growth as (in increasing order): linear, iteration-independent, iteration-dependent, or inconclusive. We discuss the semantics in Sect. 5.4; but briefly, they provide verbal descriptors. For example, a linear variable has only a light dependency on the loop; its value is updated in each iteration by no more than a constant factor. The iteration dependencies describe how a variable behaves if the loop iteration count changes. While our method is not complete for arbitrary programs, we can guarantee that the inferred mwp-bounds are *optimal* w.r.t. the expressiveness of the underlying theory. The exact definition of optimality is given in Sect. 5. Informally though, an mwp-bound is optimal if it is the least-bound form admitted in the categorization.

▶ **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 $\max(\mathtt{X3}, \mathtt{X2} + \mathtt{X5})$.
- Variable `X4` value is iteration-dependent, and bounded by $\mathtt{X4} + \mathtt{X1} \times \mathtt{X5}$.

We determine manually the precise postconditions for comparison. The linear variables never change from their initial values. If the loop iterates, the final value of `X3` is $\mathtt{X2}^2 + \mathtt{X5}$; otherwise, the value remains unchanged. Variable `X4` postcondition is precise as expressed. ◻

In the remainder of the introduction, we highlight the key insights and broader context of our work.

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 on just loop commands [8]. In verification, inferring inductive loop invariants—conditions implied by a loop's precondition and preserved in each iteration—is among the most challenging problems [15, 42, 46]. Although the communities differ in their motivations for studying loops, we focus on the intersection. Complexity-theoretic analyses are naturally suited for postcondition inference, yet we have found only a few publications explicitly 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* [14] 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. Thus it provides guarantees by construction. Implicit complexity is implicit in at least two ways: computational model and explicit program bounds need not be specified [31]. Our results demonstrate how the syntactic orientation makes implicit complexity a natural candidate for integrating complexity-theoretic solutions to practical applications.

Subprograms to reason about general programs. It is well-known by Rice's Theorem [37] that constructing a perfect static analyzer for arbitrary programs is impossible. However, this creates endless opportunity in designing increasingly useful approximative techniques. Typically, analyses sacrifice precision or termination, but rarely expressiveness [30, p. 4].

Our approach—like two comparative techniques [27, 13] in Sect. 6.2—challenge this norm. Starting with a subprogram analysis, we map our programming language to general (Turing-complete) programs. The justification is, if sufficient program fragments can be analyzed, the technique yields useful feedback. This is enabled by our bottom-up way to derive variable summaries over loops, which avoids common scalability issues of static analysis [41, 10].

1.2 Contributions

- Our main result is a technique for automatic postcondition inference, founded on the flow calculus of mwp-bounds. It supports software engineers in program verification by reducing required manual effort in specifications generation. The technique is applicable in imprecise contexts and in absence of exact initial 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 numerical 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 through comparisons with 3 state-of-the-art analyzers and performance experiments. The results validate the theoretical claims, provide evidence of practical efficiency and utility, and show that the technique generalizes to the targeted problem and natural algorithms.

2 Conceptual 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`, that performs command `C` until the expression `b` becomes false.
3. *Postcondition* Q, the final logic expressions asserted to hold after the loop terminates.

Using a Hoare triple [22], 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 [18].

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 inferred postconditions are expectedly provable. However, pre- and postcondition alone are typically too weak to prove the program correct. A specification typically requires loop invariants to strengthen the assertions and make the specification provable (refer to Appendix A for an example).

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 [18]. 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 [15, 46].

Inductive loop invariants and postconditions are symbiotic: the discovery of one assists finding the other [18]. 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 $0 \leq \mathtt{i} \leq \mathtt{n}$ 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 non-trivial and laborious.

2.2 Final values with the flow calculus of mwp-bounds

The *flow calculus of mwp-bounds* [24, 5] 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* $x_1, \dots, x_n$, for natural-number variables $\mathtt{X}_1, \dots, \mathtt{X}_n$, and the *final values* x_i' of $\mathtt{X}_i$ (for $i = 1, \dots, n$). 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 in a bottom-up manner. 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 and 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. The mwp-bounds are monotonic over-approximations; it does not decrease if variable value decreases over subtraction. If a derivable program terminates, the soundness theorem of the flow calculus [24, p. 11] guarantees the variable value growth is polynomially-bounded. 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; and more broadly a common limitation of implicit computational complexity systems [7]. 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.

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.

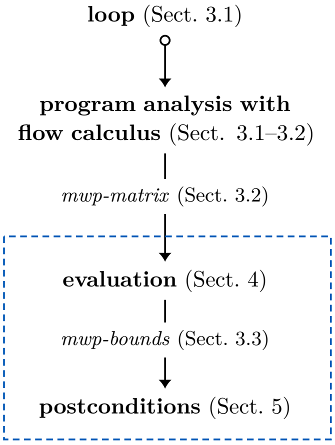

Figure 1 Analysis workflow.

As illustrated in Fig. 1, we specialize the flow calculus of mwp-bounds to loop postcondition inference. The boxed region identifies the enhancements introduced in this paper. The region outside the box shows the existing concepts and methods we build on.

The workflow assumes as input an imperative loop (Sect. 3.1) that we analyze using the flow calculus (Sect. 3.1–3.2). The analysis produces an *mwp-matrix* (Sect. 3.2). Our enhancements concern evaluating the mwp-matrices. We describe an evaluation procedure that enables extracting the optimal mwp-bounds from an mwp-matrix. These in turn provide the postconditions that address the problem formulation of Sect. 1.1.

▶ Note. Construction of mwp-matrices is not essential for the developments of this paper. We preview the construction procedure briefly in Sect. 3.1 and make no adjustments to it. Rather, mwp-matrices are the inputs of interest; our concern is the information they encode. A reader interested in how mwp-matrices are constructed should refer, in order, to [24, Sect. 5–6] and [5, Sect. 2–3]. Our work assumes the procedure described in [5].

3.1 Imperative 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 [5, Sect 2.2] and previewed in Figures 2 and 3. The calculus omits evaluating loop iteration bounds, termination, 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.

3.2 Decoding mwp-matrices

An mwp-matrix is an abstraction that captures data flow facts about the analyzed program. It contains up to an exponential number of derivations, thus it is a complex data structure. The best strategy for understanding it is recursive, starting from the elements.

3.2.1 The mwp-matrix elements

Coefficients. The coefficients $\textsc{mwp}^{\infty} = \{0, m, w, p, \infty\}$ characterize variable dependencies in commands. Recall, 0 stands for absence of dependency; the *mwp* are *m*aximal of linear, *w*eak polynomial, and *p*olynomial, respectively; and ∞ means failure. They are the core building blocks of mwp-matrices, hence the name of the calculus. A programs with simple data flows is assigned an mwp-matrix of plain coefficients. However, the coefficients alone are insufficient to capture nondeterminism in an mwp-matrix.

Nondeterminism. The nondeterminism arises from the inference rules of the mwp-calculus shown in Fig. 2[1]. There are the three ways to analyze an expression of binary addition[2]. A variable is denoted as $\mathtt{X}_i$ for the ith column of the mwp-matrix (alternatively, $\mathtt{X}_j$ for the jth column). Informally, E2 says "we can always assign a w to the variables of an expression".

$$\frac{}{\vdash \mathtt{e} : \{{}^{w}_{i} \mid \mathtt{X}_i \in \mathrm{var}(\mathtt{e})\}}\ \text{E2} \qquad \frac{\vdash \mathtt{X}_i : V_1 \quad \vdash \mathtt{X}_j : V_2}{\vdash \mathtt{X}_i + \mathtt{X}_j : pV_1 \oplus V_2}\ \text{E3} \qquad \frac{\vdash \mathtt{X}_i : V_1 \quad \vdash \mathtt{X}_j : V_2}{\vdash \mathtt{X}_i + \mathtt{X}_j : V_1 \oplus pV_2}\ \text{E4}$$

Figure 2 A selection of mwp-calculus rules for assigning vectors to expressions.

In rules E3 and E4, a vector of coefficients is denoted by V_n. The rule E3 says "the vector of the left operand is multiplied by a polynomial coefficient." The p coefficient tracks the fact that certain patterns of such data flow might not be polynomially bounded [24, p. 13]. Applying the rules to a vector propagates the effect to all variables that are transitively dependent on the left operand. Rule E4 is similar, except p coefficient is applied to the right operand. Thus, three different derivation can arise from an expression, and the command that contains the expression. The total number of derivations admitted by a program follows from this nondeterminism. Going forward, we omit the details of these rules and instead refer to them by a mapping $0 \mapsto \text{E4}, 1 \mapsto \text{E3}, 2 \mapsto \text{E2}$[3].

Domain. A domain $\mathcal{D} = \{0, 1, 2\}$ is a bidirectional map to encapsulate derivation options. The domain members can be translated to mwp-calculus inference 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 during program analysis. The degree is equal to the count of binary arithmetic operations (+ and -) in the program.

[1] To ease the presentation, we show rules of the original flow calculus [24].
[2] Corresponding rules exist for subtraction and multiplication is always handled by E2.
[3] Some formulations treat multiplication with a similar mapping, but then it is always onto rule E2.

Representing nondeterminism in mwp-matrices. A derivation choice δ is the second core building block. It enables representing 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. Informally, it captures that an inference rule i is applied at program point j. 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. First, an index is allowed to occur at most once, and second, the sequence is sorted by the index in ascending order. The restrictions support efficient computation during matrix construction. Since we are only concerned with *interpreting* mwp-matrices, we assume every Δ satisfies the uniqueness and order properties by construction.

By Fig. 2, it is evident that one variable can be assigned multiple coefficients. The fact that Δ reduces to a particular 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 making derivation choices, thus Δ can be empty. For example, both $m.(\delta(0,0), \delta(2,1))$ and 0 are monomials by definition. The former says "apply rule 0 at index 0, and rule 2 at index 1, to obtain an m-coefficient".

Finally, the fact that *different* Δ reduce to *different coefficients* is represented by a *polynomial structure*. The polynomial structure is prefixed by 0 to ensure it is non-empty.

▶ **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 $m \geq 0$.* ◻

The elements of an mwp-matrix are polynomial structures. Their positioning in an mwp-matrix indicates which variables they refer to.

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[4]. The matrix size is determined by the number of program variables, such that given n variables, the matrix size is $n \times n$. The matrix is labelled by the program variables. 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; ①
  X4=X4+X5; ② }
```

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

We mark the program points where a derivation choice must be made as ① and ②, giving the mwp-matrix a degree of 2. The program points correspond to indices 0 and 1, respectively. The addition expressions are analyzed by rules of Fig. 2. The mwp-matrix contains derivations choices at columns `X3` and `X4` because they are the data-flow targets of the additions.

[4] 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. Since the program has no commands that would introduce multiple derivations on these variables, their polynomial structures are simple coefficients. Exactly one mwp-bound describes each of the three variables.
- Variable X3 shows varying dependencies on X1, X2, and X5, at corresponding rows. The coefficients indicate that the dependencies are at most polynomial. The absence of ∞ means the value growth of X3 is acceptable in all derivations.
- Variable X4 is assigned ∞ in some derivations. Intuitively, this is because the value of X4 accumulates at each iteration, making the value growth potentially problematic. The ∞ signals that some derivations fail, and identifies X4 as the source of failure. However, this is the current extent of our capabilities to describe X4. □

This paper develops 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 degree. Clever solutions are necessary to explore and interpret the mwp-matrix data efficiently, but have been missed by previous refinements [5, 6].

3.3 Interpreting mwp-bounds

The columns of an mwp-matrix encode variable value growth bounds. An mwp-bound is an expression of form $\max(\vec{x}, \mathrm{poly}_1(\vec{y})) + \mathrm{poly}_2(\vec{z})$. 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 ×. The variables with w-flow occur in poly_1 and variables with p-flow in poly_2. 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. The differences between the bound forms of Example 7 are discussed in Sections 5.1 and 5.4.

▶ Note. 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...

X2' ≤ X2	...X2 is bounded linearly in its initial value.
X3' ≤ max(X3, X2 + X5)	...X3 is bounded by a weak polynomial in X3 or X2 + X5.
X4' ≤ max(X4) + X1 × X5	...X4 is bounded by a polynomial in X4 and X1 × X5.

□

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.

The nondeterminism of the mwp-calculus permits capturing more programs, but in turn 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 coefficients assigned to each variable. The coefficients are not obvious from the derivation choices of an mwp-matrix; rather, they require an application. An *application* reduces the polynomial structures to simple coefficients, which in turn leads to derivation failure or produces 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 an evaluation 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 $\vec{C} = (c_1, c_2, \dots, c_k)$, where $c_i \subseteq \mathcal{D}$ and $c_i \neq \emptyset$ for all $i \in \{1, 2, \dots, k\}$.* ◻

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: the degree of choice k, domain $\mathcal{D}$, and a set of unwanted derivation choice-sequences, $\mathcal{S} = \{\Delta_1, \dots, \Delta_n\}$ 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 (Lines 3–7). All interesting evaluations fall between these extremes.

The evaluation operates in two steps. First it simplifies $\mathcal{S}$ to the minimal length, non-empty 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 reduces $\mathcal{S}$ to a minimal set. Different simplifications are applied on $\mathcal{S}$ iteratively until convergence, including the following.

Remove super-sequences. If a sub-sequence leads to an unwanted outcome, a longer sequence producing the same outcome is redundant. For example, if we have in $\mathcal{S}$ two sequences, $((0,0),(0,1),(1,2))$ and $((0,1))$, we remove the longer sequence.

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. For example, the three sequences $((0,0)(1,1))$, $((1,0)(1,1))$, $((2,0)(1,1))$ can be replaced by $((1,1))$.

Tail-elimination. By symmetry, the same as above, but on the tail element.

The generation step (Lines 13–19) produces the choice vectors. They are constructed by computing the cross product of $\mathcal{S}$. We take one derivation choice from each sequence,

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)
 1 $\mathcal{C} \leftarrow \varepsilon$
 2 // Handle base cases
 3 if $k = 0$ or $|\mathcal{D}| \leq 1$ or $\mathcal{S} = \varnothing$ then
 4     if $\mathcal{S} = \varnothing$ then
 5         Create $\vec{c} \coloneqq \mathrm{ChoiceVector}(k, \mathcal{D})$        ▷ $k$-length vector of elements in $\mathcal{D}$
 6         $\mathcal{C} \leftarrow \vec{c} :: \mathcal{C}$
 7     return $\mathcal{C}$
 8 // Step 1: Simplify $\mathcal{S}$ until convergences
 9 do  Capture initial $size \coloneqq |\mathcal{S}|$                                 ▷ $\mathcal{O}(\mathcal{S}^3)$
10     $\mathcal{S} \leftarrow \textsc{Simplify}(\mathcal{S})$
11 while $size \neq |\mathcal{S}|$
12 // Step 2: Generate choice vectors
13 Compute $P \coloneqq$ the product of sequences in $\mathcal{S}$
14 for all paths $p$ in P do                                       ▷ $\mathcal{O}(\prod_{s \in \mathcal{S}} |s|)$
15     Create $\vec{c} \coloneqq \mathrm{ChoiceVector}(k, \mathcal{D})$
16     for all $(i, j)$ in $p$ do                                   ▷ $\mathcal{O}(|\mathcal{S}|)$
17         Remove $i$ at $\vec{c}_j$
18     if $\forall j, \vec{c}_j \neq \varnothing$ then
19         $\mathcal{C} \leftarrow \vec{c} :: \mathcal{C}$
20 return $\mathcal{C}$
```

then eliminating those choices. This prevents choosing any unwanted sequence completely. If the choice vector elements are non-empty 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. The algorithm obviously terminates. Since the generation step involves a product, it is the source of potential 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 detail online in the documentation of pymwp[5].

▶ **Example 9** (*LucidLoop* is derivable)**.** In Example 6, the distinct monomials causing failure are $\infty.\delta(1,1)$ and $\infty.\delta(2,1)$. The mwp-matrix degree is $k = 2$, the domain is $\mathcal{D} = \{0,1,2\}$, and the unwanted derivation choice-sequences are $\mathcal{S} = \{((1,1)),((2,1))\}$. The evaluation returns a choice vector $\big(\{0,1,2\},\{0\}\big)$ witnessing successful derivation choices. The choice vector communicates that we can select any derivation rule to analyze command ①, but must apply "inference rule 0" (E4) at command ②. ◻

The application of a choice vector requires making a selection at each vector index, then applying the selections 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).

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

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.

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 can be determined by evaluation on derivation choices $\mathcal{S} = \{((0,0)),((1,0)),((2,0))\}$—the distinct 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 $\big(\{2\},\{0,1,2\}\big)$. ◻

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 [24, p. 11]. Additional caution is needed when a program is not derivable.

4.3 Variable mwp-bounds in presence of failure

A derivation failure arises from the inference rules associated with commands `while` and `loop` (in Fig. 3). The side-condition of the loop (L) rule says "we are permitted to analyze a loop command, if the coefficients at the diagonal are at most m". The side-condition ensures data flows cannot exceed polynomial value growth. The rule application involves computing a fixed point over the loop body and accounting for the iteration guard variable, $\mathtt{X}_\ell$. The side condition of `while` loops (W) is more restrictive. It requires, in addition, that no p coefficient can occur anywhere in the mwp-matrix. The rule is more restrictive because a `while` loop may not terminate, yet still ensures at most polynomial values. Any `while` loop that satisfies the side-condition is analyzed by computing a fixed point of the loop body.

$$\forall i, M^*_{ii} = m \quad \frac{\vdash \mathtt{C} : M}{\vdash \mathtt{loop}\ \mathtt{X}_\ell\{\mathtt{C}\} : M^* \oplus \{{}^{p}_{\ell}\!\rightarrow j \mid \exists i, M^*_{ij} = p\}}\ \mathrm{L}$$

$$\forall i, M^*_{ii} = m \text{ and } \forall i,j, M^*_{ij} \neq p \quad \frac{\vdash \mathtt{C} : M}{\vdash \mathtt{while\ b\ do\ \{C\}} : M^*}\ \mathrm{W}$$

Figure 3 The mwp-calculus rules for commands `while` and `loop`.

Programs that do not satisfy the side-conditions fail the derivation. 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 inconclusive. In the mwp-matrix, such derivations translate to ∞ coefficients. This treatment of failure limits the utility of the analysis. Since our goal is to infer variable postconditions in loops, this restriction excludes many programs we want 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** (An always failing derivation). The program is not derivable because variable `X4` is assigned ∞ coefficients in every derivation,

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

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

But not all variables are problematic. By the 0 coefficients that are boxed for emphasis, there is no dependency between variables `X2` and `X5` and the problematic `X4`. Reasoning about variable `X3` is more complicated due to the many ∞ coefficients in column `X3`. ◻

A baseline determination of program derivability is straightforward by Algorithm 1. In the negative case, at least one variable must be a source failure. It is critical to recognize that when a variable causes derivation failure, it fails in every derivation. Therefore, 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 all failing variables, we query the mwp-bound as described in Sect. 4.2. The treatment recognizes that mwp-matrices simultaneously track both dependence and *independence* of how data flows in variables between commands.

▶ **Example 11** (cont.). By the choice vector $(\{2\},\{0,1,2\})$, at index 0 variable `X3` permits one derivation choice: 2. Applying it to the mwp-matrix column `X3` produces coefficients `X2`$(2,0)\mapsto w$, `X3`$(2,0)\mapsto m$, `X4`$(2,0)\mapsto 0$, and `X5`$(2,0)\mapsto w$. By the 0 coefficient, variable `X3` is independent of `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 $W_1 \equiv \max(0, \texttt{X1} + \texttt{X2}) + 0$ and $W_2 \equiv \max(\texttt{X1}, 0) + \texttt{X2}$ and $W_3 \equiv \max(\texttt{X2}, 0) + \texttt{X1}$ are all numerically equal to $\texttt{X1} + \texttt{X2}$. 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 $\max(0, \texttt{X1} + \texttt{X2}) + 0$ 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 non-empty, 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* $\vec{C_a} = (a_1, a_2, \dots, a_k)$ *and* $\vec{C_b} = (b_1,\ b_2,\ \dots,\ b_k)$ *be choice vectors of length* k*, we define the intersection of* $\vec{C_a}$ *and* $\vec{C_b}$ *as*

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

□

▶ **Example 14** (Successful and optimal mwp-bound of X3). The choice vector intersection enables checking if two properties hold in the same derivation. Example 9 reveals *LucidLoop* is derivable by choice vector $(\{0,1,2\},\{0\})$. Example 10 identifies that variable X3 is assigned its optimal mwp-bound in derivations $(\{2\},\{0,1,2\})$. To locate the derivations that make the whole program derivable *and* assign the optimal mwp-bound to X3, we take the intersection of the two choice vectors $(\{0,1,2\},\{0\}) \cap (\{2\},\{0,1,2\}) = (\{2\},\{0\})$. □

5.3 Program analysis for postcondition inference

Obtaining a technique for postcondition inference requires adjusting the imperative 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 imperative language. We compute postconditions for all variables whose mwp-bounds are expressible in the mwp-calculus with the the evaluation procedure of Sect. 5. Since the analysis is compositional, the preferred order of processing is bottom-up, analyzing loop nests before the parent. This permits composing the mwp-matrices of the loop 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 impacts variable postconditions.

Linear. A variable's mwp-bound is linear if its value does not change, it is a target of direct assignment (without arithmetic), or it is modified by constant factors. Inside loops, other operations are "too strong" to retain linear behavior. In *LucidLoop*, `X1`, `X2` and `X5` are in this category because their values do not change.

Iteration-independent. The notion of iteration independence differs from invariance; rather, iteration-independence is better understood to be 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 it eventually reaches a fixed point. Iteration independence implies that beyond the fixed point, the variable is unaffected by an increase in loop iteration count. In *LucidLoop*, variable `X3` is iteration-independent. Assuming the loop iterates at least once, the final value of `X3` is determined by `X2` and `X5`, and increasing the iteration count has no impact. The mwp calculus recognizes this fact through the w coefficient.

Iteration-dependent. Iteration-dependent mwp-bounds capture 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 bound soundly. In *LucidLoop*, variable `X4` is iteration-dependent because at each iteration `X4` increases in value. Changing the number of times the loop iterates impacts the final value of `X4` accordingly.

Inconclusive. The inconclusive ∞-result marks the absence of an expressible postcondition and characterizes variable value growth that falls outside the previous three cases. It deductively informs that a variable's value growth is either beyond a polynomial or too challenging to express with mwp-bounds. In Example 11, variable `X4` is marked inconclusive because, in absence of loop termination, the value of `X4` grows in perpetuity.

Note on numerical minimality. 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$_\ell$

We implemented our postcondition inference as an extension of the static analyzer pymwp. pymwp [6] is an open source Python implementation of the flow calculus of mwp-bounds for C language. It does not rely on any external theories or solvers; its only dependency is a C parser. Any program acceptable to the parser is forwarded to analysis. By default, pymwp takes as input a C file and analyzes the variable value growth in each function.

Our implementation is a new loop analysis mode, mwp$_\ell$, 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$_\ell$ 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 of this paper are implemented only in mwp$_\ell$ to permit experimentally evaluating the impact.

The postcondition inference required mapping the procedure described in Sect. 5.3 to C language. mwp$_\ell$ 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$_\ell$ checks that a C language `for` matches this form. Otherwise, a `for` loop is converted to a `while` to ensure all loop forms are analyzable. The translation is safe because the treatment of a `while` command is more restrictive than that of a `loop`. However, the translation comes at a cost since a `while` bounds strictly fewer variables than a `loop`. Within loops, jump statements (`break`, `continue`) and verification macros (`assume`, `assert`) are supported and treated as `skip`. For soundness, pymwp must be run with a *strict* flag. The flag ensures the C program is fully expressible in the imperative language of Sect. 3.1 before producing results.

If we want to insert the analyzer results as concrete verification assertions, two additional steps are required. First, we record the initial variable values in additional variables, since the initial values are needed to express the postconditions. Second, we define the constants left implicit in mwp-bounds. We demonstrate these steps in Appendix A.

Research questions. We examine three research question concerning the implementation of our technique.

1. **How does our technique compare to leading inference approaches?** We compare postcondition inference between mwp$_\ell$ and three state-of-the-art automatic analyzers, and assess their capabilities in supporting specification tasks in imprecise contexts (Sect. 6.2).
2. **How effective is our technique at discovering postconditions in general?** We execute mwp$_\ell$ on four benchmark suites containing versatile challenges from complexity theory and loop invariant inference literature (Sect. 6.3).
3. **What is the impact of the introduced theoretical enhancements?** We hypothesize the enhancements of this paper are critical for extending the utility of the flow calculus in formal verification. To quantify the improvements, we compare the results of mwp$_\ell$ and mwp$_f$ across the four benchmark suites (Sect. 6.3).

6.2 Comparisons of analyzer capabilities

To answer RQ1, we compare[6] mwp_ℓ to advanced alternatives in following categories: complexity, verification, and postcondition inference. KoAT [9, 28][7] is a static analyzer of complexity bounds; Duet [17, 2] is a static analyzer with capabilities to infer polynomial invariants, and Daikon [16, 1] is a dynamic invariant detector for postconditions. As comparison workloads we consider the loops in Fig. 4; they are extracted from the benchmark suites of Sect. 6.3. For Listings 3 and 4, mwp_ℓ returns postconditions for variables `X1`–`X5`. The results of Listing 3 are presented in Sect. 1.1. For Listing 4, variables `X3` and `X5` are labelled linear and the other mwp-bounds are iteration-independent: $\mathtt{X1'} \le \max(\mathtt{X1}, \mathtt{X2} + \mathtt{X3})$, $\mathtt{X2'} \le \max(\mathtt{X2}, \mathtt{X3})$, and $\mathtt{X4'} \le \max(\mathtt{X4}, \mathtt{X5})$. For Listing 5, mwp_ℓ labels `y` as linear and `x` as ∞. To give a preview of our findings, which we summarize in Table 1, the other techniques are orthogonal. The comparison shows mwp_ℓ is useful in cases the other tools ignore, and vice versa.

Listing 3 *LucidLoop*

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

Listing 4 Function condition

```
while(nondet())
{ X1=X2+X2;
  X2=X3+X3;
  X4=X5+X5; }
```

Listing 5 Iteration limit

```
assume(y==0);
while(y<1000)
{ x=x+y;
  y=y+1; }
```

Figure 4 Loop cases for comparison. The loop in Listing 3 is known to terminate and its guard variable `X1` does not occur in the body. In Listing 4, the loop iteration and termination are unknown since they are controlled by a nondeterministic function. The loop in Listing 5 has a fixed iteration space, 0–1000, but the postcondition of `x` is difficult to infer. Assuming $\mathtt{y} = 0$, the precise formula is $\mathtt{x'} = \mathtt{x} + (\mathtt{y'} \times \mathtt{y'} - \mathtt{y'}) \div 2$ where `y'` is the iteration count[8].

The complexity analyzer KoAT. As part of the automated termination and complexity prover AProVE [19], KoAT infers complexity bounds—including time, cost, and size bounds—of integer programs. The bound closest to our semantics of postconditions is *size bounds*, which indicate how large the absolute value of an integer variable may become [27]. Applying KoAT on C programs requires translating the C code, through LLVM bytecode, into integer programs; the transformation output is then analyzed by KoAT [20]. The translation step renames the program variables. Variables that do not contribute to the complexity result are discarded during pre-processing. This design has the following effects: (i) KoAT can distinguish between loops that differ only on iteration counts and (ii) size bounds are inferred only for variables that impact the loop iteration. The latter is complementary to the mwp `loop` form "do `C` `X` times" that requires a guard variable `X` does not occur in loop body `C`.

Case-wise comparison. For Listing 3, KoAT generates size bounds for `X1` and `i`, but ignores the body variables `X2`–`X5`. The size bounds are $\mathtt{X1'} : 2 \cdot \mathtt{X1}$ and $\mathtt{i'} : \mathtt{X1} + 2$. For Listing 4, KoAT produces no size bounds for variables `X1`–`X5`. For Listing 5, KoAT tightly bounds `y` by size bound $\mathtt{y} : 1000$, and it could distinguish if the limit changed. However, variable `x` is discarded because it does not impact the loop iteration.

[6] Refer to Appendix B for technical details of this comparison.
[7] By KoAT we mean the extended re-implementation, sometimes styled as KoAT2.
[8] Without assuming $\mathtt{y} = 0$, the postcondition is $\mathtt{x'} = \mathtt{x} + (\mathtt{y'} \times \mathtt{y'} - \mathtt{y'}) \div 2 - (\mathtt{y} \times \mathtt{y} - \mathtt{y}) \div 2$.

Feature	Daikon	Duet	KoAT	mwp_ℓ
Postcondition analysis scope	function entry/exit	invariants	loop control	loop internal
Numerical domain	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{N}$
Postcondition expressivity	$>+$	$>+$	$>+$	$+$
Program fragment analysis	○	◐	●	●
Handles program divergence	○	●	●	●
Distinguishes iteration count	●	○	●	○
Body variables coverage	◐	●	◐	●
Results soundness	○	●	●	●
Postconditions for Fig. 4	◐ ○ ◐	○ ○ ○	◐ ○ ◐	● ● ◐

Table 1 Summary of analyzer capabilities. The analyzers target different program scopes with varying behaviors and assumptions. The symbols mean ● = yes, ◐ = partial, and ○ = no. The features are phrased to make a positive response favorable w.r.t. the problem formulation of Sect. 1.1.

Duet analyzer of unbounded concurrency. Duet is a static analyzer for concurrent programs whose thread count cannot be statically bounded [2]. It contains a collection of analysis tools, including implementations of *transition ideals* from [13]. Transition ideals compute loop summaries that produce over-approximations of a formula that describes the loop body. These summaries can capture non-linear invariants and generalize over arbitrary control flow. The theory is monotone, such that a program with more precise specifications yields a more informative summary. The transition ideals theory is implemented in Duet as linear and quadratic simulation modes. Duet then aims to prove program correctness using the invariants generated by these modes.

Case-wise comparison. The programs have no assertions because the goal is to infer them. In absence of assertions, Duet produces a single response "no errors and no unsafe assertions", which is not meaningful for our use case. After adding postconditions, Duet verifies them successfully. For example, if we add to Listing 5 assertions $x' = x + (y' \times y' - y') \div 2$ and $y' = 1000$, it verifies with "0 errors, 2 safe assertions". When assertions are not available, mwp_ℓ could assist Duet by inferring initial assertions.

Postconditions with Daikon. Daikon [16, 1] is a dynamic invariant detector, with front-ends to support many programming languages. Daikon predicts *likely* invariants at function entry and exit points. The inference relies on execution traces, templates, and configuration options. Daikon infers postconditions for a `return` variable. A single function may generate multiple postconditions, and variables that do not impact the return variable do not occur in postconditions. Daikon requires executing a program to obtain traces, analyzing the traces, then confirming (possibly manually) that the postcondition is correct.

Case-wise comparison. Daikon does not produce results for the displayed fragments, but after a modification to a whole-program, it infers `X3' > X2`, `X3' > X4` and `X3' > X5` for Listing 3. These postconditions do not generalize or require additional assumptions to prove. Daikon finds no result to describe Listing 4. For Listing 5, Daikon infers the precise numeric value for `x` or `y`, depending on which variable is returned. Although it does not recover the arithmetic formula, it is the only technique that gives a postcondition for `x`.

Table 2 Benchmark suite characteristics summarized by count and (mean). A single benchmark can contain multiple functions, loops, and sequential and nested loops. Since the evaluated analyzers target different program scopes, variable counts are specified by scope. Loop-scoped variables include loop guards and variables in the loop body.

Suite	Linear	mwp	Complexity	Non-linear	Total
Benchmarks	49	30	504	37	620
Lines of code	652 (13.31)	270 (9.00)	6,066 (12.04)	710 (19.19)	7,698
Loops	49 (1.00)	30 (1.00)	740 (1.47)	48 (1.30)	867
Variables, loop	117 (2.39)	105 (3.50)	1,921 (3.81)	208 (5.62)	2,351
Variables, functions	131 (2.67)	105 (3.50)	1,519 (3.01)	208 (5.62)	1,963

6.3 Performance experiments

Setup – benchmarks, metrics, and environment. We consider four benchmark suites of numerical C loops, summarized in Table 2. The complexity suite is *Complexity C Integer* from the Termination Problem Database, v11.3 [3]. It contains mainly linear time complexity problems, with unknown termination behavior, and is used in the annual Termination and Complexity Competition [21]. The linear[9] [42] and non-linear (NLA) [32, 46] suites are from invariant inference literature. The benchmarks range from single-variable loops to classic algorithms that embed linear and non-linear loop invariants. The mwp suite [6] contains problems designed to pose challenges to mwp-based analyzes. To obtain strictly comparable results, we excluded from the mwp suite nested loops and loop-less benchmarks. All suites contain branching statements and nondeterministic control expressions. Complexity and non-linear suites have nested and sequential loops.

We measure the count and kind of discovered postconditions, i.e., variable bounds. We record all meta-data collected by pymwp, including input statistics and analysis time. The time reflects analysis only, excluding parsing and file writes, with a timeout of 10 seconds. We use the strict flag to enforce only fully supported syntax is analyzed.

We ran the experiments on commodity hardware, on a 10-core macOS M1 15.3.2 arm64 native host with 16 GB of RAM. Complete experiment resources and replication instructions will be made available as a public artifact.

RQ2 Inference generalizability. Across the four suites mwp_ℓ succeeds at analyzing 85%-100% of the loops and finds postconditions for 69%-87% of variables in those loops, in Table 3. The postconditions are different in form from most complexity analyzers (refer to [27, 6]). Most complexity analyzers are essentially restricted to linear arithmetic [27], but mwp_ℓ shows success also on the non-linear suite. The non-linear suite results are particularly encouraging, because it contains complex arithmetic and real algorithms. The verification suites analysis time remains modest compared to the complexity-based suites. It suggests the computations that are challenging to mwp-based analyses are not prevalent in the natural algorithms. On the linear suite mwp_ℓ finds postconditions nearly instantly.

We observe limitations mainly on expressivity. First, loops with unsupported syntax are not analyzable. For example, the complexity suite has loops that update variables by function calls to a random integer generator; such operations are inherently outside the aims

[9] We excluded the 9 invalid benchmarks [39, Appendix G], and unified benchmarks with same precondition and loop, as they are identical for postcondition inference; ending with 49 benchmarks.

Table 3 Analysis results for mwp_f and mwp_ℓ, with totals and (mean) of experiment metrics. Loops/functions shows the number of analyzed instances, including the suite coverage (%). Bounds is the number of inferred postconditions, with a mean relative to analyzed variables, *vars*. The *mwp* columns show postconditions breakdown by form, and ∞ is the number of unbounded variables.

Analyzer	Suite	Loops	Vars.	Bounds	m, w, p	∞	Time, ms
mwp_ℓ	Complexity	636 (.86)	1,576	1,367 (.87)	1,349, 18, 0	209 (.13)	22,561 (30.49)
(ours)	Linear	49 (1.0)	117	102 (.87)	101, 1, 0	15 (.13)	93 (1.90)
	Non-linear	42 (.88)	180	124 (.69)	116, 8, 0	56 (.31)	461 (9.60)
	mwp	30 (1.0)	105	77 (.73)	45, 28, 4	28 (.27)	2,255 (75.17)
	Suite	Functions	Vars.	Bounds	m, w, p	∞	Time, ms
mwp_f	Complexity	403 (.80)	1,152	792 (.69)	789, 0, 3	360 (.31)	18,674 (37.05)
	Linear	49 (1.0)	131	95 (.73)	94, 1, 0	36 (.27)	117 (2.39)
	Non-linear	28 (.76)	151	11 (.07)	10, 1, 0	140 (.93)	744 (20.11)
	mwp	30 (1.0)	105	60 (.57)	32, 20, 8	45 (.43)	1,878 (62.60)

of the flow calculus. Some variables' growth rates are truly beyond polynomial bounds and such variables are not expressible. Finally, in some cases—like Listing 5 from the linear suite—the value growth is too complicated to express, and thus missed by mwp_ℓ. Based on the results, increasing expressivity of the flow calculus should be the main direction for future enhancements. Overall, mwp_ℓ succeeds at analyzing most loops and generates informative, non-∞, postconditions rapidly.

RQ3 Impact of theoretical enhancements. Comparing mwp_ℓ to mwp_f allows quantifying the significance of the theoretical enhancements of this paper. We show the results in Table 3. mwp_f is the state-of-the-art analyzer among the implementations computing mwp-bounds. Although the results are not strictly comparable—since mwp_f targets function-scope and mwp_ℓ targets loop-scope—we preprocessed the mwp suite to be comparable. The other three suites can be compared over the means. The results on the mwp suite show that the enhancements of the paper increase the abilities to obtain postconditions. On the mwp suite, mwp_ℓ bounds 73% of variables compared to mwp_f at 57%. Further, mwp_ℓ always finds the optimal bounds, captured in the distribution of the bound forms. Across the other suites, particularly on the non-linear suite, mwp_ℓ shows significant improvement in producing useful results. mwp_f fails on many non-linear benchmarks because there exists a failing variable, which yields ∞ on all variables. The results further suggest there is a performance benefit to scoping the analysis to loops, instead of functions, unlike originally designed.

7 Related works

Our work is primarily related to specification inference for software verification, but with deep complexity-theoretic roots. Automatic specification inference, whether from program syntax or natural language, is a challenging problem [15, 46]. A common first step is a conceptual "divide-and-conquer": breaking down the problem into parts: preconditions, postconditions, and inductive invariants. Restricting the problem space in this way enables developing increasingly better partial solutions. Inductive invariants and postconditions are the most relevant to our technique.

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.

Loop invariant inference. The existing literature is rich in techniques for loop invariant inference. One way to categorize them is based their style of execution [18]. The static approaches [12, 25, 15, 11, 40] infer invariants over program syntax using e.g., search-based techniques and program logics. The static approaches are naturally limited in precision and scalability, but can offer sound guarantees. Dynamic approaches [16, 42, 39, 45, 46] are data-driven and rely on execution traces. One benefit is that they can process any traceable program without having to worry about language-level details. Unfortunately, the quality and validly of the generated invariants is strongly influenced by the available inputs, and it is not possible to guarantee the invariants hold for all executions. For best of both worlds, hybrid techniques combine static and dynamic methods [34, 33, 32]. For example, the "guess-and-check" strategy involves an analyzer iteratively guessing a likely invariant based on program traces, then using a verifier (an SMT solver) to check its validity [32, 42]. The techniques can be further be distinguished by the form of invariants they generate: linear inequalities [15], polynomials [40, 34, 32], array properties [23, 34], etc. While loop invariant inference is necessary to obtain formal guarantees for loops, the problem is complementary to the one addressed in this paper. This is enforced by our comparison with Duet in Sect. 6.2.

Postcondition inference. Existing works that *explicitly* target postcondition inference are rare. Among the few instances, Popeea and Chin [36] introduce a static abstract interpretation-based analysis. It is conceptually closest to our goals, though abstract interpretation differs considerably from the flow calculus of mwp bounds. While the Popeea and Chin analysis could have been a useful comparison target, unfortunately we could not locate an implementation, making a comparison impossible. Among dynamic postcondition analyzers, EvoSpex [29] is 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, but stronger emphasis on numerical invariants. It can perform invariant inference at different program points, making it usable for postcondition inference. Although DIG would have been a suitable comparison target, we selected Daikon [16] for our comparison on the basis of maturity. Compared to our static technique, the dynamic analyses in general have operational differences similar to those of Daikon. Finally, if we expand the scope further, and consider technques that compute postconditions as a component of their main result, we can expand to other static analyzers. Our comparison with the complexity analyzer KoAT [20] demonstrates this concept and how our technique compares in that regard.

Inferring whole program specifications. 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 (LLMs) to infer specifications. For example [35, 44, 26] are a few recent or ongoing projects exploring LLM-assisted specification generation. However, such fully-automated inference strategies may suffer from the *assertion inference paradox* [18]. 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 that if we infer the specification from the implementation, the fundamental property of independence is lost, between the mathematical property and the software that attempts to achieve it. Therefore, not all specification conditions should be inferred automatically. The mwp-bounds *assist* in the verification process by providing approximative postconditions to add to programs. Thus we intentionally maintaining human oversight and avoid the paradox.

8 Conclusion, open problems and future directions

Specifications are essential to formal methods and automatic inference facilitates their discovery. We have proposed a complexity-theoretic analysis for inferring partial specification conditions, namely postconditions. This result required four new enhancements: projecting the flow calculus of mwp bounds on individual variables, improving derivation failure-handling, introducing an evaluation strategy to obtain optimal mwp-bounds, and applying the analysis to a new use case in verification. A comparison study shows our technique offers complementary strengths among the related approaches. The distinction comes from the kind of postconditions it computes—approximative and sound loop variable summaries—and how it arrives to those results. The lightweight, static and compositional analysis requires only program fragments, and does not rely on external solvers, number libraries, etc. The theory is materialized in our implementation, mwp_ℓ. Our experiments show that the enhancements of the paper extend the utility of the flow calculus, and make it applicable toward use in formal verification.

Although we have extended the capabilities of the flow calculus, multiple directions for future work remain; some of which emerge from this paper. The two main directions are enriching the expressiveness of the imperative language and improving the analysis precision.

- 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 [8]. In turn, it could uncover program optimization opportunities. For example, in *LucidLoop*, 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 of the flow calculus, and motivated to uncover its potential utilities. The analysis could already be implemented as a developer plug-in to assist writing formal specifications. In future research, we will consider extensions to the capabilities of the flow calculus. We are curious if similar solver-free syntactic analyses could be designed to infer other specification conditions or invariants. Another ongoing project is to formally verify the theory. The results of this paper provide justification for the formalization effort.

References

1 Daikon, 2024. URL: https://plse.cs.washington.edu/daikon.

2 Duet static analyzer, 2024. URL: https://github.com/zkincaid/duet.

3 The Termination Problem Database, 2024. URL: https://github.com/TermCOMP/TPDB.

4 Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3Test: Assertion-Augmented Automated Test case generation. *Information and Software Technology*, 176:107565, 12 2024. doi:10.1016/j.infsof.2024.107565.

5 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.

6 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.

7 Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. *Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs*, pages 203–218. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48899-7_15.

8 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, 5 2020. doi:10.23638/LMCS-16(2:4)2020.

9 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*, 38(4):1–50, 8 2016. doi:10.1145/2866575.

10 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.

11 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.

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 John Cyphert and Zachary Kincaid. Solvable Polynomial Ideals: The Ideal Reflection for Program Analysis. *Proc. ACM Program. Lang.*, 8(POPL), 1 2024. doi:10.1145/3632867.

14 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.

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

16 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, 12 2007. doi:10.1016/j.scico.2007.01.015.

17 Azadeh Farzan and Zachary Kincaid. *Duet: Static Analysis for Unbounded Parallelism*, pages 191–196. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39799-8_12.

18 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.

19 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing Program Termination and

Complexity Automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 10 2016. doi:10.1007/s10817-016-9388-y.

20 Jürgen Giesl, Nils Lommen, Marcel Hark, and Fabian Meyer. *Improving Automatic Complexity Analysis of Integer Programs*, pages 193–228. Springer International Publishing, 2022. doi:10.1007/978-3-031-08166-8_10.

21 Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. *The Termination and Complexity Competition*, pages 156–166. Springer International Publishing, 2019. doi:10.1007/978-3-030-17502-3_10.

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

23 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.

24 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, 8 2009. doi:10.1145/1555746.1555752.

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

26 Yue Chen Li, Stefan Zetzsche, and Siva Somayyajula. Dafny as verification-aware intermediate language for code generation, 2025. doi:10.48550/arXiv.2501.06283.

27 Nils Lommen and Jürgen Giesl. *Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs*, pages 3–22. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-43369-6_1.

28 Eleanore Meyer and Nils Lommen. KoAT2: Complexity Analysis Tool for Integer Programs, 2024. URL: https://github.com/aprove-developers/KoAT2-Releases.

29 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.

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

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

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. Symlnfer: 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, 9 2014. doi:10.1145/2556782.

35 Gabriel Poesia, Chloe Loughridge, and Nada Amin. dafny-annotator: AI-Assisted Verification of Dafny Programs, 2024. doi:10.48550/arXiv.2411.15143.

36 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.

37 Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 3 1953. doi:10.1090/S0002-9947-1953-0053041-6.

38 David 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. URL: https://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification.

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. *Form. Asp. Comput.*, 37(1), 12 2024. doi:10.1145/3689374.

44 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.

45 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.

46 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.

47 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, 8 2015. doi:10.1145/2786805.2786858.

A Application to concrete program verification

Program verification requires recording the initial variable values (L3) and defining constants left implicit in mwp-bounds (L14). After adding invariants, the Dafny verifier immediately constructs a proof, which confirms that the assertions always hold. Manufacturing suitable invariants is a challenge for the related works that specialize in inductive invariant inference.

Listing 6 *LucidLoop* verified in Dafny.

```
 0  method LucidLoop(X1: nat, X2: nat, X3: nat, X4: nat, X5: nat){
 1
 2    // capture initial values
 3    var X1', X2', X3', X4', X5' := X1, X2, X3, X4, X5;
 4
 5    for i := 0 to X1'
 6      // invariants
 7    {
 8      X3' := X2' * X2' + X5';
 9      X4' := X4' + X5';
10    }
11
12    // postconditions
13    assert X1'≤X1 ∧ X2'≤X2 ∧ X5'≤X5; // linear
14    assert X3'≤Max(X3, X2*X2+X5);     // weak polynomial
15    assert X4'≤X4+X1*X5;              // polynomial
16  }
```

B Technical details of analyzer comparisons

We analyzed the programs in Listings 7–9 as follows.

mwp$_\ell$. We run pymwp v0.5.4 in the loop analysis mode.

```
pymwp [program].c
--mode L              # activate loop analysis
--strict              # require compliant C syntax
```

KoAT. The KoAT web interface is sufficient to confirm our findings. We applied the following options: ✓ control-flow refinement ✓ size bounds ✓ unsolvable loops, and default timeout. The web interface address is:

`https://aprove.informatik.rwth-aachen.de/interface/v-koat/c`

Duet. We ran Duet from source, git revision `1d36b05`, with following options. The compositional recurrence analysis generates invariants for sequential programs.

```
duet.exe [program].c
-cra                  # compositional recurrence analysis
-cra-refine           # enable loop refinement
-monotone             # disable non-monotone analysis features
-[theory]             # lirr-usp or lirr-sp-quad
```

Daikon. Daikon requires multiple version of a program, then compiling and tracing them with Kvasir. The * means we trace multiple versions of input.

Supply options to Kvasir before the program name argument.

```
kvasir-dtrace
--dtrace-file=[program].dtrace    # trace destination
--decls-file=[program].decls      # declarations file
[program]_*.o                     # compiled program
```

After tracing, we run Daikon with following options to infer postconditions.

```
java -cp $DAIKONDIR/daikon.jar
daikon.Daikon
--conf_limit=.50                  # confidence
-o [program].inv.gz               # output
[program]_*.dtrace                # path to traces
[program].decls                   # path to declarations
```

After inference, the invariants can be printed to a suitable display format.

```
java -cp $DAIKONDIR/daikon.jar
daikon.PrintInvariants
--wrap_xml --output_num_samples   # format options
[program].inv.gz                  # source
[program].txt                     # output path
```

Comparison programs. These are the programs from Fig. 4 expanded with headers and return statements. The programs are from the benchmark suites used in the performance evaluation of Sect. 6.3. For Duet analysis, `loop` must be renamed to `main`. However, `main` raises an error with KoAT. Daikon expects adding a separate `main` method with calls to `loop`. In Listing 9, the unsupported `assume` should be omitted before analysis with for KoAT.

Listing 7 mwp/example 3.4

```
int loop(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 8 mwp/not infinite #4

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

Listing 9 Linear #02

```
int loop(int x, int y) {
    assume(y==0);
    while(y<1000) {
        x = x + y;
        y = y + 1;
    }
    return x;
}
```