Polynomial Postconditions via mwp-Bounds

Abstract. Formal specifications are necessary to guarantee software meets critical safety properties, but retrofitting specifications to existing software requires considerable resources and expertise. Although inference eases discovery of specifications, existing inference techniques are limited in targeting different specification conditions. Particularly, inference of postconditions has rarely been the targeted focus of prior investigations. This paper presents a static program analysis that automatically synthesizes postconditions from pure program syntax. These postconditions represent final values of variables in numerical loops. For each variable, the analysis finds one approximative postcondition, of at most polynomial form, if it is expressible in the underlying theory. This theory is a sound and compositional flow calculus with complexity-theoretic origins. However, applying the flow calculus to postcondition inference required several enhancements. As technical contributions, we resolve multiple open problems about the flow calculus and increase its expressive power. Experiments and comparison study show the postcondition analysis is orthogonal to alternative techniques and generalizes to classic algorithms. These findings suggest the postcondition analysis can offer support in specification tasks even if full implementation details are still in flux.

Keywords: Postcondition Inference · Static Analysis · Verification

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 [9]. Continued research effort must be dedicated to reducing the entry barriers.

This paper aims to improve accessibility of formal methods. We do this by introducing a program analysis that synthesizes partial specification conditions. We develop an automatic inference of approximative postconditions in numerical loops. Postconditions are specialized assertions that must hold after the loop terminates. They are complementary to inductive loop invariants, i.e., conditions that must hold pre-iteration and be preserved by the loop [31]. Although postconditions may be derivable from invariants, loop invariant inference is one of the hardest problems in verification [14,32]. Therefore, we suggest the opposite approach: starting with postconditions. It is well-known that having postconditions supports discovery of inductive invariants [16]. Moreover, as generalized

assertions, postconditions assist in various software development activities [27], like testing [5,35] and code maintenance [29]. However, dedicated studies of postcondition inference are a novelty in research literature [28,23].

Specifications must be precise, consistent, and complete descriptions of program behavior. Making formal specifications mechanically verifiable requires they are expressed in a structured language. However, defining formal specifications is non-trivial and requires expertise. In the meantime, various resource analyses [20,10] already compute varied notions of postconditions as means to arrive to their primary result. Thus, it seems natural that resource analyses could be used to infer postconditions for formal specifications. This is precisely the key intuition we exploit in the paper. We present a complexity-theoretic flow calculus [20,6] that we use to automatically infer variable postconditions. In this view, the program implements the intended behavior, but misses a formal proof. Our goal is to assist software engineers in completing this missing step.

1.1 Problem formulation and solution overview

We analyze deterministic imperative numerical loops. The loop may be bounded or unbounded, with unknown termination behavior. The loop manipulates a fixed number of natural number variables. Beyond type, we make no assumptions about initial variable values. Our goal is to automatically infer postconditions of the variables occurring in the loop. A postcondition is an assertion about a variable's value that holds after iteration, if the loop terminates. The loop may represent a program fragment that is still under development. Reducing contextual information this way is practically motivated because such information is commonly absent in realistic unverified programs.

Listing 1.1 shows 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. The program *LucidLoop*, in Listing 1.2, shows the problem variant we address in the paper. When precise variable values, precondition, and iteration count are unknown, what postconditions (?) can we infer from the syntax?

Listing 1.1: Precise context (ideal)

```
assume(X2≤10 ∧ X4==2 ∧ X5>4);
for(i=0;i<10;i++) {
  X3=X2*X2;
  X3=X3+X5;
  X4=X4+X5; }
assert(X3≤100+X5 ∧ X4≥42);
```

Listing 1.2: Imprecise context (ours), LucidLoop

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

Our analysis infers postconditions that express the growth of variable values in the loop. Adding postpositions supports discovery of other specification conditions, like invariants, that then permit formal verification (the Duet analyzer in Sect. 6.2 is an example). The analysis is sound, guaranteeing that if a postcondition can be inferred, it is known to hold.

As result, our analysis generates mwp-bounds. These are symbolic expressions describing the value growth of the program variables. An *mwp-bound* [20] represent a variable's *final value* in terms of the *initial values* and omitting constants. An mwp-bound is an approximation of the upper bound of the final value. A critical idea is that the **mwp-bounds are postconditions**. Since the problem formulation assumes no pre-conditions, a function in terms of the inputs is the most precise achievable postcondition in most cases. An mwp-bound is maximally polynomial in form. A variable value growth that is beyond a polynomial is not expressible as an mwp-bound. However, our experiments (Sect. 7) suggest that this is not a prohibitive limitation. The analysis successfully assigns postconditions to most variables in a set of diverse benchmarks.

Based on the mwp-bound form, we categorize the variables as linear, iteration-independent, iteration-dependent, or inconclusive (in increasing order). We discuss the semantics of the categories in Sect. 5.4; but briefly, they are behavioral descriptors. For example, a linear variable has only a light dependency on the containing loop because its value is updated by at most a constant. While our analysis is not complete for arbitrary programs, we can guarantee that the inferred mwp-bounds are optimal (Sect. 5.1) w.r.t. in the expressiveness of the flow calculus. In other words, we find the least upper bound that describes variable value growth. At program point ⍰, our analysis gives the following result.

- Variables $\texttt{x1}$, $\texttt{x2}$, and $\texttt{x5}$ values have grown at most linearly from the initials
- Variable $\texttt{x3}$ value is iteration-independent and bounded by $\max(\texttt{x3}, \texttt{x2} + \texttt{x5})$
- Variable $\texttt{x4}$ value is iteration-dependent and bounded by $\texttt{x4} + \texttt{x1} \times \texttt{x5}$

We can determine manually the precise postconditions for comparison. The linear variables never change from their initial values. The final value of $\texttt{x3}$ is its initial value $\texttt{x3}$ or $\texttt{x2}^2 + \texttt{x5}$, depending on if the loop iterates. For variable $\texttt{x4}$, the precise postcondition is $\texttt{x4} + \texttt{x1} \times \texttt{x5}$.

1.2 Contributions

1. Our main result is an automatic analysis for postcondition inference in numerical loops (Sect. 5).The analysis is applicable in imprecise contexts and in absence of initial variable values and program annotations.
2. To obtain the analysis, we extend the flow calculus of mwp-bounds with two new capabilities: locating optimal bounds and bounding variables in presence of whole-program derivation failure (Sect. 4). This provides a strictly more expressive system than prior formulations of the flow calculus.
3. To materialize our theory, we implement mwp_ℓ for analyzing numerical loops in C (Sect. 5.5). mwp_ℓ is already integrated into a public static analyzer pymwp, extending the utility of our results beyond the paper presentation.
4. We demonstrate the relevance and effectiveness of the technique through analyzer comparisons (Sect. 6) and experiments (Sect. 7). The findings show our technique is orthogonal to alternatives, generalizes to natural algorithms, and infers postconditions for most variables in the evaluated benchmarks.

More broadly, our contributions are refreshing in three ways. They strengthen the existing connections [25,26] between complexity theory and formal verification. We show how to transform a technique from complexity theory to a practical application, which is non-trivial and rarely done [24,6]. Finally, we show that constructing static analyses around small core languages can be beneficial, if we assume a compositional analysis that covers a sufficiently large class of programs.

2 A High-Level Picture

2.1 Conceptual primer of the flow calculus

Our postcondition inference is based on the *flow calculus of mwp-bounds* [20,6]. It is a data flow analysis that tracks *variable value growth* between commands in imperative programs. The analysis aims to discover a polynomially bounded data-flow relation between the *initial values* $x_1, \ldots, x_n$, for natural-number variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$, and the *final values* x_i' of $\mathtt{X}_i$ (for $i = 1, \ldots, n$). More precisely, the flow calculus aims to guarantee variables are always polynomially bounded, including in every intermediate program state. For example, the program `X1=X2+X3;X1=X1+X1` is satisfactory, because the final value of each variable grows at most polynomially w.r.t. inputs. Variables `X2` and `X3` do not change and final value of `X1` is $2 \times (\mathtt{X2} + \mathtt{X3})$. In contrast, the program `X1=1;while(X2){X1=X1+X1}` is unsatisfactory, because variable `X1` grows exponentially in $2^{\mathtt{X2}}$. When satisfactory value growth can be confirmed for all variables, the analysis assigns the program a *bound* that characterizes the value growth. The analysis result is derived statically, by applying inference rules to the commands of the program, in a compositional bottom-up manner.

Value growth as data flow graphs. As an internal bookkeeping procedure, the flow calculus collects information in matrices (Sect. 3). For enhance intuition, we represent these matrices also as data flow graphs (Fig. 1). A *data flow graph* (DFG) of program $\mathcal{C}$ is a multigraph of n vertices where n is the number of variables involved in $\mathcal{C}$. The vertices are the variables of $\mathcal{C}$ and the edges denote dependence between variables.

Flow coefficients. The edges of a DFG are quantified by flow coefficients. The coefficients characterize the weight of the dependence between variables. When no dependence exists, the coefficient is 0 (we omit 0-edges in DFGs). An m denotes a maximally *linear* dependence. Inside loops, a linear dependence permits direct data flows between variables, but prohibits arithmetic operations. The next stronger dependence, w, stands for *weak polynomial*. The w coefficient permits some arithmetic operations, but with restriction. Inside loops, the value growth must be eventually unaffected by changes in the loop iteration count. The strongest permissible dependence is *polynomial*. A p coefficient signals that a variable value grows at most polynomially throughout computation. Finally,

other forms of dependence are either beyond polynomial or not expressible. The ∞ coefficient marks these cases as *failure*. The flow calculus is nondeterministic, therefore multiple coefficients may characterize one pair of variables.

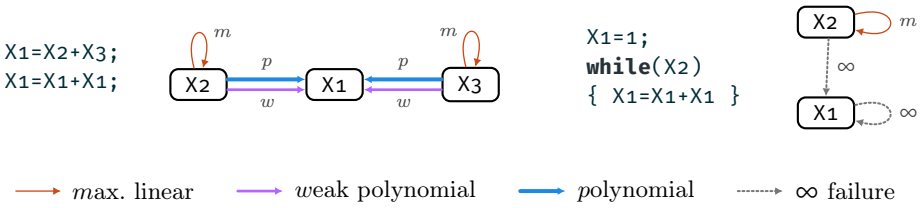


Fig. 1: Simple imperative programs and their corresponding data flow graphs. Edges below ∞ represent acceptable value growth. In the left program, all variable values grow at most polynomially, i.e., satisfactorily. In the right program, variable `X1` grows exponentially. This is "too fast" for a polynomial bound.

Analysis result and derivability. If all final values can be bounded by polynomial, then the input program is *derivable*. The flow calculus assigns an *mwp-bound* to every variable of a derivable program. An mwp-bound is an expression, in terms of the inputs, that characterizes the value growth of a single variable. If a derivable program terminates, the soundness theorem of the flow calculus [20, p. 11] guarantees that the variable value growth is polynomially-bounded. The result is sound but not complete— not all satisfactory programs can be captured, but on success we can prove the result with a derivation. Since the analysis omits termination, this is a partial correctness guarantee.

The flow calculus offers no guarantee to programs that are not derivable; their variable value growth is *unknown*. A program always fails if some variable value grows "too fast", e.g., exponentially. Failure may also arise from inability to express satisfactory behavior. This is a built-in limitation of the flow calculus; and more broadly, a common limitation among related systems [8, p. 2]. The flow calculus alleviates the issues with nondeterministic inference rules. The nondeterminism increases expressiveness, to capture a larger class of programs. But one program may admit multiple derivations, which in turn complicates the analysis procedure. A program is derivable if there exists a derivation without an ∞ coefficient.

2.2 Loop specifications for formal guarantees

Formally verifiable programs are defined as precise mathematical models through specifications. Verifying the program requires giving a proof that the program satisfies its specification. The motivation for using formal methods is the strength of guarantees it provides. In contrast to tests and inspection, a proof conclusively ensures behavioral correctness at the modeled level of abstraction.

Specifications. A specification is a formal contract with three components. 1. *Precondition* P, is the initial logic expressions assumed to hold before entering the loop. 2. Program, `loop b C`, performs command `C` until the expression `b` becomes false. 3. *Postcondition* Q, is the final logic expressions asserted to hold after the loop terminates. Using a Hoare triple [19], we can express the specification as $\{ P \}$ `loop b C` $\{ Q \}$. The implementation of the program is correct if we can construct a proof that shows the program satisfies its specification in all states. A correctness proof requires verifying that every computation terminates, every call to another procedure satisfies its preconditions, and the postcondition holds at program termination [16]. For an example of specifications, refer to Appendix A.

Relating postconditions and loop invariants. A postcondition is a higher-level view describing the program goal. In case of loops, a *postcondition* is a conditions that must hold after the loop terminates. The postcondition inference problem can be expressed informally 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 $P = \top, \{ \texttt{b} \}\ \texttt{C}\ \{ \top \}, \neg \texttt{b} \rightarrow Q \vdash$ `loop b C`. Postconditions inferred using the flow calculus are provable by the soundness theorem. However, pre- and postcondition alone are typically too weak to prove a program correct. Loop invariants are often necessary to make the specification provable.

An *invariant* describes an assertion about a program location that must be true for every program state reaching the location [16,27].[1] A loop invariant is a weakened form of the postcondition [16]. An invariant is inductive if it holds the first time the location is reached and is preserved in every cycle returning to the location [31]. Verifying loops requires discovering sufficiently strong inductive loop 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.

Inductive loop invariants and postconditions are symbiotic: the discovery of one assists finding the other [16]. Every invariant is necessarily a (weak) postcondition, as it must hold at termination, but a postcondition must not be invariant. For example, given natural numbers `i` and `n`, the loop `for(i=0;i<n;i++) i++;` has an invariant $0 \leq$ `i` $\leq$ `n` and a postcondition `i = n`. Invariant inference techniques may assume preconditions and postconditions are known. However, this assumption is impractical, since manually adding specifications to code fragments is non-trivial.

[1] A postcondition is also an invariant; for clarity, we always refer to a postcondition as such and use ‘invariant‘ to refer to an inductive loop invariant.

3 Technical Preliminaries of the Flow Calculus

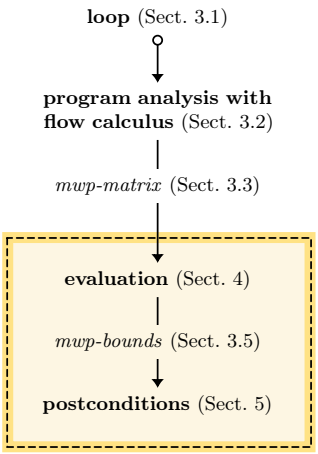


Fig. 2: Analysis workflow

Fig. 2 illustrates the complete analysis workflow. The boxed region highlights the steps and enhancements introduced in this paper. The region outside the box refers to the existing techniques we build on.

The analysis takes as input an imperative loop (Sect. 3.1). We analyze the loop using the inference rules of the flow calculus (Sect. 3.2). As output, the analysis produces an *mwp-matrix* (Sect. 3.3). Our enhancements involve evaluating these mwp-matrices. We define first an evaluation procedure (Sect. 4.2) that enables extracting optimal mwp-bounds from mwp-matrices, and in more cases than previously (Sect. 4.4). This provides the postconditions that address the problem formulation of Sect. 1.1.

3.1 Imperative language of input programs

Definition 1 (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.*

We implicitly convert between conventional **`for`** loops and `loops` of the imperative language in examples. 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. Similarly, the approach is applicable even if a programming language provides little structure.

3.2 About construction of mwp-matrices

How mwp-matrices are constructed is not crucial for the developments of this paper. The construction procedure is well-established in literature [20, Sect. 5–6] and [6, Sect. 2–3].[2] We give a brief overview, but make no adjustment to the

[2] Our work assumes the variant described in [6].

construction procedure. Rather, mwp-matrices are our inputs of interest. We focus on *interpreting* the information they encode.

During matrix construction, the flow calculus assigns matrices to commands of the input program. The procedure resembled typing, except the "types" are complicated matrices. An *mwp-matrix* M captures data flow facts about the analyzed program $\mathcal{C}$. The matrices track, through flow coefficients, how data flows in variables between commands. What coefficients are assigned is determined by the *inference rules* of the flow calculus. The relevant inference rules are included in Figures 3 and 5.[3] To trade precision for efficiency, the flow calculus omits evaluating loop iteration bounds and reasoning about termination and control expressions, overestimating their effect. At conclusion, the flow calculus assigns one mwp-matrix to the analyzed program, $\vdash \mathcal{C} : M$. It contains all derivations of the input program in one (complex) data structure.

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

Fig. 3: A selection of flow calculus rules for assigning vectors to expressions. A variable is denoted as $\mathtt{X}_i$ where the i refers to an index of a matrix column (resp. $\mathtt{X}_j$ for the jth column). In rules E3 and E4, a vector of coefficients is denoted by V_n. In other words, the rules assign vectors of coefficients to variables. Three rules can be used to analyze a single expression. The nondeterminism of the flow calculus comes from these inference rules.

Nondeterminism is necessary to improve the expressive power of the flow calculus, but complicates the analysis procedure. There are the three rules (Fig. 3) to analyze an expression of binary addition[4], like `X1 + X2`. Informally, the rule E2 says "we can always assign a w coefficient to all variables in the expression". Although such constant treatment seems potentially harmful, the inference rules for analyzing commands (Fig. 5) ensure only acceptable data flow patterns are accepted as derivable. The rule E3 says "the vector of the left operand is multiplied by a polynomial coefficient." The p-coefficient tracks the fact that certain data flow patterns might not be polynomially bounded [20, 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 to E3, except the p coefficient is applied to the right operand. Thus, three different derivation can arise from one expression, and from every command that contains the expression. 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] To ease the presentation, we show rules of the original flow calculus [20]. These rules are consistent with the enhanced variant [6] we build on.

[4] Similar rules exist for subtraction, and multiplication is always handled by E2.

3.3 Our starting point of interest: interpreting mwp-matrices

An mwp-matrix represents the derivations of an analyzed program. Before diving into full mwp-matrices, this section describes its *elements*.

Basic terms. Flow coefficients—described in Sect. 2.1—are the core building blocks of mwp-matrices. The flow coefficients are elements of the finite set $\text{MWP}^{\infty} \triangleq \{0, m, w, p, \infty\}$. A *domain* $\mathcal{D} \triangleq \{0, 1, 2\}$ is a bidirectional map of the flow calculus inference rules. In other words, the domain members can be translated to flow calculus inference rules (E2–E4) and vice versa. The *degree* of choice, $k : \mathbb{N}$, is a counter of derivations. The degree represents the number of times a derivation choice must be made during program analysis. The degree is precisely equal to the count of binary arithmetic operations whose operator is either + or −.

Definition 2 (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 $j \leq k$.*

A derivation choice, denoted by δ, represents the nondeterminism of the flow calculus. 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.

Definition 3 (Derivation choice sequence). *Letting k be the degree of choice, we define a derivation choice sequence as $\Delta = (\delta_1, \delta_2, \ldots, \delta_{k'})$ where $0 \leq k' \leq k$.*

Since a program is a *series* of commands, correspondingly we must have *sequences* of derivation choices, denoted as Δ. Simple data flow patterns do not require making derivation choices; therefore Δ can be empty. There are two restrictions on Δ. First, an index j is allowed to occur at most once in a sequence, i.e., the indices in a sequence must be unique. Second, the sequence is sorted by the index in ascending order. These 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. 3, 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 $\alpha \in \text{MWP}^{\infty}$ be a coefficient and Δ be a sequence of derivation choices, we define a monomial as (α, Δ).*

A monomial in the flow calculus is disjoint from the classic notion of monomials. For example, both $m.(\delta(0, 0), \delta(2, 1))$ and 0 are valid monomials by definition. The former says "if we apply rule 0 then rule 2 (at indices 0 and 1, resp.) then we obtain an m coefficient". The latter says "no matter the choice, we always obtain a 0 coefficient". Finally, the fact that different Δ reduce to different coefficients is represented by a *polynomial structure*.

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

The polynomial structure is prefixed by 0-monomial to ensure it is always non-empty. **The elements of an mwp-matrix are polynomial structures.**

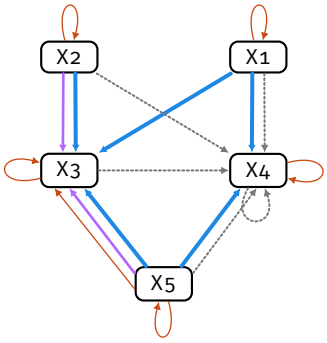
3.4 Decoding an mwp-matrix by example

An mwp-matrix M associates variables with polynomial structures.[5] This association is defined by position. The matrix size is determined by the number of program variables, such that for n variables, the matrix size is $n \times n$. The matrix is labelled by the variables. An mwp-matrix is interpreted column-wise. The data-flow facts about a variable at column j are collected in the polynomial structures at rows $i = (1, \ldots, n)$ in M_{ij}.

Example 1 (mwp-matrix of LucidLoop). Consider the mwp-matrix of LucidLoop.

```
for(i=0;i<X1;i++)
{ X3=X2*X2;
  X3=X3+X5; ①
  X4=X4+X5; ② }
```

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



We mark the program points where a derivation choice must be made as ① and ②. The program points correspond to indices 0 and 1, respectively. These points give the mwp-matrix a degree of 2. The addition expressions are analyzed by the inference rules of Fig. 3. The mwp-matrix contains derivations choices at columns `X3` and `X4` because they are the data-flow targets of the additions. Correspondingly, in Fig. 4, the variables are targets of dependence edges.

Fig. 4: A DFG of LucidLoop.

Variables `X1`, `X2`, and `X5` are assigned at most m coefficients. An m-flow at the mwp-matrix 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. Variable `X3` shows varying dependencies on `X1`, `X2`, and `X5` at corresponding rows. The coefficients indicate that the dependencies are at most polynomial (similarly observable in the DFG). 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 loop iteration and its value growth is potentially problematic. The ∞-flow signals that some derivations fail and flags `X4` as the source of failure. However, this is the current extent of our capabilities to describe `X4` [7].

This paper develops enhanced strategies to extract more precise information from mwp-matrices. poly$_2$Although we show only compact cases, an mwp-matrix

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

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 flow calculus refinements [6,7].

3.5 Interpreting mwp-bounds

The mwp-matrix columns encode the variable value growth bounds. An mwp-bound is an expression of form $\max(\vec{x}, \text{poly}_1(\vec{y})) + \text{poly}_2(\vec{z})$. Variables characterized by m-flow are listed in $\vec{x}$; w-flows in $\vec{y}$, and p-flows in $\vec{z}$. Variables characterized by 0-flow do not occur in the expression. No bound exists if some variable is characterized by ∞. The poly_1 and are *honest polynomials*, build up from constants and variables by applying $+$ and $\times$. 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. The mwp-bound is a monotonic over-approximation, i.e., it does not decrease if variable value decrease over subtraction. A program *bound* is a conjunction of its variables' mwp-bounds. Ex. 2 shows illustrative cases of mwp-bounds. The differences between the bound forms are discussed in Sections 5.1 and 5.4.

Note 1. For variable `X` and an mwp-bound W, we write $\texttt{X'} \leq W$ to denote the mwp-bound of variable `X`. We use the overloaded notation `X'` to refer to the variable's final value. Variables in W always refer to initial values.

Example 2. The mwp-bound (on left) is interpreted as, the *final value* of...

$\texttt{X2'} \leq \texttt{X2}$ ...`X2` is bounded linearly by its initial value.

$\texttt{X3'} \leq \max(\texttt{X3}, \texttt{X2} + \texttt{X5})$ ...`X3` is bounded by a weak polynomial in `X3` or `X2` + `X5`.

$\texttt{X4'} \leq \max(\texttt{X4}) + \texttt{X1} \times \texttt{X5}$ ...`X4` is bounded by a polynomial in `X4` and `X1` × `X5`.

4 Variable-Guided Matrix Exploration

4.1 Addressed limitations

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

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

The nondeterminism of the flow calculus permits capturing more programs, but creates a potential state explosion problem. Effectively handling this aspect becomes critical in the second analysis phase, where an mwp-matrix is *evaluated* to find mwp-bounds of variables. The first challenge is finding the derivation choices that do not produce failure. The second challenge 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 then leads to derivation failure or yields 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. The ideal solution should find the optimal bound (if one exists) and without the redundancy of iterating every derivation.

The evaluation strategy we present operates as follows. It takes the *unwanted* derivation choices then negates them. The evaluation result the 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, i.e., non-∞. The result of an evaluation is a *disjunction of choice vectors* that compactly capture the permissible derivation choices.

Definition 6 (Choice vector). *Letting $\mathcal{D}$ be a domain and $k \in \mathbb{N}$ be a choice degree, we define a choice vector as $\vec{C} = (c_1, c_2, \ldots, c_k)$, where $c_i \subseteq \mathcal{D}$ and $c_i \neq \varnothing$ for all $i \in \{1, 2, \ldots, k\}$.*

We show choice vectors in Examples 3, 4, and 5.

4.2 Evaluation for identifying derivable programs

The precise mwp-matrix evaluation strategy is presented in Algo. 1 and we describe it here informally. The evaluation is parametric on three inputs: (i) the degree of choice k, (ii) domain $\mathcal{D}$, and (iii) a set of unwanted derivation choice-sequences, $\mathcal{S} = \{\Delta_1, \ldots, \Delta_n\}$ where $n \geq 0$. Observe that all parameters are obtained from an mwp-matrix, but the mwp-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 advance 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. When a sequence leads to an unwanted outcome, every longer sequence producing the same outcome is redundant. For example, if $\mathcal{S}$ contains $((0,0),(0,1),(1,2))$ and $((0,1))$ we remove the longer 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 $\vec{c} \coloneqq \text{ChoiceVector}(k, \mathcal{D})$ ▷ 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 $\vec{c} \coloneqq \text{ChoiceVector}(k, \mathcal{D})$
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** $\mathcal{C}$

Head-elimination. If many non-singleton sequences differ only on their head element value, and the values cover the domain $\mathcal{D}$, then the head is redundant. The sequences can be replaced by one sequence without the head element. For example, the 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, 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 Algo. 1. The algorithm obviously terminates. Since the generation step involves a product, it is a 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 the documentation of pymwp.[6]

Example 3 (LucidLoop is derivable). In Ex. 1, the distinct monomials that cause failure are $\infty.\delta(1,1)$ and $\infty.\delta(2,1)$. The mwp-matrix degree is $k = 2$,

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

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. Ex. 5).

4.3 Variable projection and querying mwp-bounds

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 Algo. 1 does not require mwp-matrices or coefficients as input, it can be directly reused 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 (and whether 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 with at most w or p can be queried similarly, by adjusting $\mathcal{S}$.

Example 4 (Variable $\mathtt{x3}$ *is bounded by at most* w *coefficients).* In Ex. 1, variable $\mathtt{x3}$ does not have a derivation with at most m coefficients. This can be determined by evaluation of the distinct choices in column $\mathtt{x3}$ with $\{w, p, \infty\}$ coefficients, i.e., $\mathcal{S} = \{((0, 0)), ((1, 0)), ((2, 0))\}$. This evaluation does not return a choice vector. However, an mwp-bound of at most w 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 flow calculus [20, p. 11]. Additional caution is needed when a program is not derivable.

4.4 Variable mwp-bounds in presence of failure

Derivation failure arises from the inference rules of commands **`while`** and **`loop`**, in Fig. 5. The interesting parts about these rules are the side conditions. For the **`loop`** rule (L), the side-condition says a loop can be analyzed if the coefficients at the mwp-matrix diagonal are at most m. The **`while`** (W) rule is more restrictive. In addition to diagonal m, no p coefficients are allowed to occur anywhere in

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

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

Fig. 5: The flow calculus inference rules for commands **while** and **loop**.

the mwp-matrix. These side-conditions ensure variable values grow at most polynomially over repeated executions of the command body.

A program that does not satisfy the side-conditions is not derivable. Then, variable value growth is inconclusive and no bound is assigned to its variables. The flow calculus offers no guarantees to programs that are not derivable. A single rogue variable can cause a whole-program derivation failure. This treatment of failure limits the utility of the analysis. Since our goal is to infer postconditions of loops, this restriction excludes many programs we want to analyze.

Improving the failure handling. Our intuition is that the mwp-matrices accumulate information that permits reasoning about certain variables in presence of failure. The main new idea is to *isolate the failing variables* from those with acceptable behavior. If such isolation is possible, then the remaining variables can be analyzed. The idea is conservative because it does not change the rules of the flow calculus. Rather, the improvement is based on leveraging fully the information already collected in mwp-matrices. We make only the analytical adjustment illustrated in Fig. 6. Our treatment of failure is sound by the guarantees of the original flow calculus – explained shortly after the motivating example.

Example 5 (Evaluating an always failing derivation). The program in Fig. 6a is not derivable because variable X4 fails in every derivation. The area of failure, that we want to isolate, is shaded in the mwp-matrix of Fig. 6b. Not all variables are problematic. There is no dependency from X4 to variables X2 and X5 (row X4). Variable X3 is more complicated due to the ∞ coefficients in column X3. Since we want derivations where X3 is derivable, we compute its choice vector: $\big(\{2\}, \{0, 1, 2\}\big)$. At index 0, variable X3 permits one derivation choice: 2. Applying $(2, 0)$ to the mwp-matrix, at column X3, yields coefficients: X2 $\mapsto w$, X3 $\mapsto m$, X4 $\mapsto 0$, and X5 $\mapsto w$. The 0 reveals that in every derivable case variable X3 is independent of X4. Therefore, variables X3, X2 and X5 can be assigned mwp-bounds.

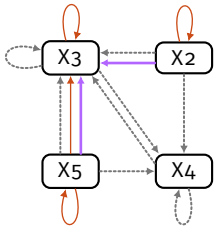
Variable grouping (Fig. 6d). The whole-program derivability is first determined by Sect. 4.2. In the negative case, at least one variable must be failing. Crucially, a failing variable fails in every derivation. Therefore, it suffices to evaluate variables individually for at most p coefficients, by Algo. 1. We group variables into two sets. Set A with variables that satisfy polynomial value growth, and set B with variables that fail. Which set each variable belongs to is determined by the existence of a choice vector.

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

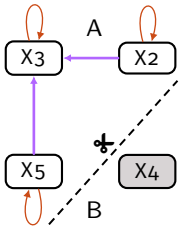
(a) Input program

	X2	X3	X4	X5
X2	m	$\infty(0,0), \infty(1,0), w(2,0)$	$\infty(0,1), \infty(1,1), \infty(2,1)$	0
X3	0	$m, \infty(0,0), \infty(1,0)$	$\infty(0,1), \infty(1,1), \infty(2,1)$	0
X4	0	$\infty(1,0)$	$\infty(0,1), \infty(1,1), \infty(2,1)$	0
X5	0	$\infty(0,0), m(1,0), \infty(1,0), w(2,0)$	$\infty(0,1), \infty(1,1), \infty(2,1)$	m

(b) mwp-matrix with failure



(c) DFG of the mwp-matrix



(d) Grouping and independence

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

while(*)
  X4=X4+X5;
```

(e) Sub-programs

Fig. 6: Improved failure handling demonstrated. The program (6a) is a modified version of LucidLoop. The variable X4 causes failure in every derivation (6b). Starting with 6c, we restrict the graph to derivations where non-failing variables can be bounded. This reduces the edges to those shown in 6d. The sub-program that corresponds to variables in set A (6e, above) is derivable.

Establishing independence from failure (Fig. 6d). Recall, the 0 coefficients track absence of data flows between variables. Thus, we can determine if sets A and B are data-flow independent from the 0s. We must ensure that there is no data flow *from* the failure set B to set A in the cases where variables in A are derivable. The condition can be determined by direct inspection of the mwp-matrix. The independence condition corresponds to a cut of the data flow graph.

Partitioning to sub-programs (Fig. 6e). When failure independence is affirmative, we partition the whole-program into two sub-programs.[7] These sub-programs correspond to the data flows in sets A and B. No bound can be derived for the failing variables in set B. Variables in set A are analyzed by the standard rules of the flow calculus, which ensures soundness. *Derivability of this sub-program is the source of increase in the flow calculus expressivity.*

5 mwp-bounds as Postconditions

5.1 Optimal mwp-bounds by form

The enhanced flow calculus now provides a technique for postcondition inference. We aim to compute the optimal postconditions for each variable, if they exist.

[7] This is a conceptual step; no concrete program transformation is necessary.

There are two unresolved concerns. First, mwp-bounds cannot be totally ordered since some 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, reasoning about optimality requires an alternative approach.

To establish an ordering, we leverage two built-in features of the flow calculus. The individual coefficient are ordered $0 < m < w < p < \infty$. Moreover, 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 7 (Optimality). *We define an order on mwp-bounds by form, by the* maximal *coefficient it contains:* $0 <$ *m-bound* $<$ *w-bound* $<$ *p-bound* $< \infty$*(none). A variable's mwp-bound is optimal if it is the least bound in this 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. It supplies the evidence that the variable's value growth is eventually loop iteration independent (discussed in Sect. 5.4). The other candidates W_2 and W_3 are too weak to reach the same conclusion. Finding a single derivation that admits the optimal form is sufficient.

5.2 Variable postcondition search

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

1. Run the mwp-matrix evaluation (Algo. 1) iteratively. Construct the set $\mathcal{S}$ from monomials whose coefficient is greater than m (next w, then p).
2. Stop at the first choice vector. This solution is optimal.

Since the search focuses on individual variables, we may want to ask whether multiple postconditions can occur concurrently. To determine the answer, we take the intersection of their choice vectors (Def. 8). A non-empty intersection specifies the derivations in which both postconditions hold. Related questions about postconditions can be formulated similarly, as operations on choice vectors.

Definition 8 (Choice vector intersection). *Letting* $\vec{C}_a = (a_1, a_2, \ldots, a_k)$ *and* $\vec{C}_b = (b_1, b_2, \ldots, b_k)$ *be choice vectors of length* k*, we define the intersection of* $\vec{C}_a$ *and* $\vec{C}_b$ *as*

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

Example 6 (Successful and optimal postcondition of X3*).* By Ex. 3, LucidLoop is derivable by choice vector $(\{0,1,2\}, \{0\})$. By Ex. 4, variable X3 is assigned its optimal postcondition in derivations $(\{2\}, \{0,1,2\})$. The whole-program is derivable *and* variable X3 is assigned its optimal postcondition in derivations defined by choices $(\{0,1,2\}, \{0\}) \cap (\{2\}, \{0,1,2\}) = (\{2\}, \{0\})$.

5.3 Program analysis for postcondition inference

Postcondition inference requires adjusting the imperative language (Sect. 3.1) to a language of loops. A *loop program* starts with a looping command (**`while`** or **`loop`**) whose body is any command in the imperative language. Using the mwp-matrix evaluation procedure, we compute postconditions for all variables that have expressible mwp-bounds. To avoid repeated analysis, we proceed from loop nests to parent and compose mwp-matrices in a bottom-up manner. Postcondition inference of loop P is defined as follows.

1. Extract all (possibly nested) loops from the program P.
2. (bottom-up) For each loop l:
   (i) Run the mwp analysis to derive the mwp-matrix, $l : M$.
   (ii) Using Algo. 1, evaluate M to determine if l is derivable.
      ▷ If yes: mark every variable as satisfactory.
      ▷ If no: mark the non-failing variables as satisfactory (Sect. 4.4).
   (iii) Evaluate satisfactory variables for optimal postconditions (Sect. 5.2).
   (iv) Record the postconditions of satisfactory variables.
3. Return the analysis result for P.

5.4 Postcondition categories as descriptors of variable value behavior

A language of loops restricts the kind of computations we may encounter. The inferred postconditions can be described by the following categories.

Linear. A variable's mwp-bound is linear if its value does not change or it is a target of direct assignment (without arithmetic). Inside loops, other operations are "too strong" to retain linear behavior. In LucidLoop, variables `X1`, `X2` and `X5` are linear because their values never change.

Iteration-independent. A variable is iteration-independent if its final value depends on a fixed number of iterations (e.g., the first or last one) or eventually reaches a fixed point. Iteration independence means that, beyond the fixed point, the variable is unaffected by an increase in loop iteration count. Thus, iteration-independence is a quasi-invariance property. In LucidLoop, if the loop iterates at least once, the final value of `X3` is determined by `X2` and `X5`.

Iteration-dependent. Arithmetic computations involving multiple changing variables lead to iteration-dependent value growth. It may occur in bounded loops, but not otherwise. This is because a continuously increasing value may be boundable under finite iteration, but cannot be bound soundly if the loop is unbounded. In LucidLoop, changing the number of times the loop iterates changes the final value of `X4` respectively. This makes `X4` iteration-dependent.

Inconclusive. If no expressible postcondition exists, the variable is inconclusive. It characterizes cases where a variable's value growth is outside the previous three categories. In Ex. 5, since the loop does not terminate, the value of `X4` grows in perpetuity, which marks the variable inconclusive.

A note on numerical minimality. One variable may be assigned multiple incomparable mwp-bounds. The definition of optimality does not guarantee the selected postposition is the numerical minimum by evaluation. For example, variable `X3` in LucidLoop is assigned three 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 $W_3 \equiv \max(\texttt{X3}, \texttt{X5}) + \texttt{X1} \times \texttt{X2}$. For purpose of this example, assume $\texttt{X1} > 0$ to consider the iterative case and ignore W_2. This leaves two alternatives. These can evaluate to distinct numeric values, yet both are equally optimal by definition.

5.5 Implementing postcondition inference with mwp_ℓ

We implemented the analysis of Sect. 5.3 as an extension of the static analyzer pymwp. pymwp [7] is an open source implementation of the flow calculus of mwp-bounds on a subset of `C`. pymwp takes as input a `C` file and analyzes variable value growth in each of its functions. Our implementation adds to pymwp a new loop analysis mode, mwp_ℓ. The loop analysis is complementary to the default function analysis mode, which we name mwp_f for distinction. The primary differences are that mwp_ℓ looks for optimal bounds by variable in a loop, and mwp_f finds existence of any bound for all variables in a function. Though applicable to both, the paper enhancements are implemented only in mwp_ℓ to enable evaluation.

During program analysis, the input file is mapped to the imperative language of Def. 1. Only loops that are fully expressible in the imperative language are analyzed. Analyzing program fragments is possible due to compositionality of the flow calculus. Stated differently, we can apply the analysis early, even if some program parts are missing. pymwp supports all loop constructs of `C`. The flow calculus treats the iteration space conservatively as an over-approximation. Keywords **`break`** and **`continue`** have no observable impact. Similarly, verification macros like **`assert`** may be present, but do not impact the analysis.

When a `C` loop construct is obviously bounded[8], it is treated as a bounded loop in the flow calculus. Otherwise, the construct is treated as an unbounded loop. Detection of loop boundedness impacts the number of derivable variables, since a bounded loop permits iteration-dependency. The current detection is based on loop form and it is still rudimentary. In the future, improving the analyzer's handling of richer loop forms would yield more bounded variables.

To use the analysis results as concrete verification assertions, two more steps are required. First, we must record the initial variable values because they are needed to express postconditions. Next, we must complete the parts that are left implicit in mwp-bounds, like constants. Sect. A demonstrates how we perform both steps. In general, the postpositions provide bounding expressions w.r.t. input variables, with some possible omissions. However, filling in the expression is conceptually easier than starting with no expression at all.

[8] For example, in **`for(int`** `i=0;i<N;i++)` the iterator `i` must not occur in the body and the body operations are arithmetic. Without nesting, it is clearly a finite loop.

6 Comparison with Related Techniques

6.1 Automatic inference of specification conditions

Our work relates primarily to approaches that aim to unify software verification and complexity. Whereas loop invariants are used in [25] to obtain complexity results, we synthesize specification conditions starting from complexity analysis. The bidirectionality suggests that further investigations, connecting the two topics, is warranted. Narrowing down to specifications, automatic specification inference in general is a challenging problem [14,34]. It is common to break down the problem into smaller parts: preconditions, postconditions, and (inductive) invariants. Inference of postconditions is the most relevant w.r.t. our analysis. Although invariants inference is studied extensively in literature [21,12,11,31,14,32,30,33,34,26,25], existing works that intentionally target postconditions are rare [28,23].

There are a few instances that infer postconditions statically. In [28], show how abstract interpretation can be used to obtain a static technique for postposition analysis. Conceptually it is close to our goals, but abstract interpretation differs considerably from the flow calculus. Moreover, no implementation is available for comparison. The complexity analyzer KoAT [18] is a static analyzer with an open source implementation. However, it targets complexity-theoretic program properties, like time and size bounds, and is not specialized for verification.

Dynamic analyzers are complementary to static techniques. They inspect program traces to infer *likely* invariants at the traced program points. EvoSpex [23] is a dynamic postcondition analyzer. It is designed for Java methods for reasoning about postconditions in classes (accessors, mutators, heap structures, etc.); and thus distant from numerical loop analysis. The invariant detector Daikon [15] handles many program constructs, including numerical loops. Since Daikon is compatible with our problem formulation, we discuss it in Sect. 6.2). DIG [26] is another numeric invariant generator. DIG can perform invariant inference at arbitrary program points, which makes it usable as a postcondition detector. Although it is similar to Daikon, DIG mixes static and dynamic techniques to obtain more informative results. In general, dynamic analyses differ notably from static analyses. Since their results are based on traces they are necessarily incomplete. The quality of inferred results is directly related to the available analysis inputs. Importantly, a dynamic analyzer requires executing the input program, which implies that the program must be runnable. The same is not necessary for our analysis that can work with program fragments.

6.2 A Technical Comparison of Alternative Approaches

Since we aim to support concrete verification tasks, we must compare our analysis to available implementations that can address the same problem. In this section, we compare[9] mwp_ℓ to three mature[10] analyzers: KoAT, Duet, and Daikon.

[9] Refer to Sect. B for the technical details of this comparison.

[10] Each analyzer is developmentally stable with 10+ years of history.

Listing 1.3:
Case: LucidLoop

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

Listing 1.4:
Case: Function condition

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

Listing 1.5:
Case: Finite iteration

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

Fig. 7: Loop cases for comparison. The loop in 1.3 is known to terminate and its guard variable `X1` does not occur in the body. In 1.4, the loop iteration and termination are unknown since they are controlled by a nondeterministic function. The loop in 1.5 has a fixed iteration space, but the postcondition of `x` is difficult to infer. Assuming `y` = 0, the precise formula is `x'` = `x` + (`y'` × `y'` − `y'`) ÷ 2 where `y'` is iteration count.

These analyzers are designed for complexity analysis, verification, and invariant inference, resp. Each analyzer analyzes different program scopes (cf. Table 1 and Sect. B.2) and has a different specification of output format. Therefore, a statistical comparison is insufficient for useful comparison. A better way is to present canonical cases that the analyzers handle differently. As the comparison workload we consider the loops of Fig. 7. They are instances from the benchmarks we will re-encounter in Sect. 7. To give a preview of our findings—which we summarize in Table 1—the alternative analyzers are orthogonal. The comparison shows mwp_ℓ is useful in cases the other tools ignore, and vice versa.

Inference with mwp_ℓ. The results of mwp_ℓ give a baseline for comparison.

- Listing 1.3: As explained in Sect. 1.1.
- Listing 1.4: `X3` and `X5` are linear. The other are `X2'` ≤ max(`X2`, `X3`), `X4'` ≤ max(`X4`, `X5`), and `X1'` ≤ max(`X1`, `X2` + `X3`).
- Listing 1.5: The program converts to a bounded loop. The constants `1` and `1000` are lifted to inputs c_1 and c_2, resp. The postconditions are `x'` ≤ `x` + c_2 × (c_1 + `y`) and `y'` ≤ `y` + (c_1 × c_2).

Complexity analyzer KoAT. KoAT [3] is a part of the automated termination and complexity prover AProVE [17]. KoAT infers complexity bounds: time, cost, size bounds, etc. The bound most relevant to our problem is the *size bound*, which indicate how large the absolute value of an integer variable may become [22]. Applying KoAT on C programs requires compiling the C code (through LLVM bytecode) into an integer program. The transformed program is then analyzed by KoAT [18]. The translation step renames the program variables. Variables that do not contribute to the complexity result are discarded during pre-processing. This treatment 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 the complement of the flow calculus, where the iterator of bounded loops is not allowed to occur in the body.

- Listing 1.3: We obtain `X1'` : $2 \cdot$ `X1` and `i'` : `X1` + 2; and `X2`–`X5` are discarded.
- Listing 1.4: No size bounds are generated for variables `X1`–`X5`.
- Listing 1.5: Variable `y` has precise size bound `y` : 1000. Variable `x` is discarded.

Duet – the analyzer of unbounded concurrency. Duet is a static verifier for concurrent programs whose thread count cannot be statically bounded [2]. We include it in this comparison because it contains analysis techniques that relate to our problem. In particular, the implementation of transition ideals [13] computes loop summaries that produce over-approximations of a formula that describes the loop body. The summaries can capture non-linear invariants and generalize over arbitrary control flow. The theory of transition ideals is monotone. TIn other words, a program with more precise specifications yields a more informative loop summary. Then, Duet aims to prove program correctness using the invariants generated from transition ideals.

- Listings 1.3 – 1.5: In absence of assertions, Duet produces a single response, `no errors and no unsafe assertions`. This response is not meaningful for our use case. After adding postconditions (assertions) manually, Duet verifies them successfully. For example, if we add to Listing 1.5 the assertions `x' = x + (y' × y' − y') ÷ 2` and `y' = 1000`, Duet verifies the program with `0 errors, 2 safe assertions`. When assertions are not available, mwp_ℓ could assist Duet toward obtaining the initial assertions.

Postconditions with Daikon. Daikon [15,1] is a dynamic invariant detector with front-ends to support many programming languages. Daikon predicts likely invariants at function entry and exit points. Critically, the function internals are opaque during invariant detection. The inference relies on execution traces, templates, and configuration options. Daikon infers postconditions for a **`return`** variable and a single function may generate multiple postconditions. Since the Daikon invariants are likely, they must be checked for correctness, possibly manually. Daikon does not produce results for the displayed fragments, but after a modification to a whole-program, it infers the following.

- Listing 1.3: `X3' > X2`, `X3' > X4` and `X3' > X5`. These either do not generalize or require additional assumptions to prove.
- Listing 1.4: No result.
- Listing 1.5: Precise numeric values for `x` or `y`, depending on which one is returned. Although the arithmetic formula is not recovered, Daikon is the only technique that can give a precise value for `x`.

Table 1: Summary of analyzer capabilities, behaviors, and assumptions. For input format, we write *program* to mean syntactical analysis written in a subset of the C language, though the subsets differ. A positive response is phrased as favorable: ● = yes, ◐ = partial, ○ = no.

Feature	**Daikon**	**Duet**	**KoAT**	**mwp$_\ell$**
Analysis scope (Sect. B.2)	function entry/exit	invariants	loop control	loop body
Input format	execution traces	program	program	program
Output format	likely invariants	SAT/error	size bounds	mwp-bounds
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. 7	◐ ○ ◐	○ ○ ○	◐ ○ ◐	● ● ●

7 Experimental Evaluation

We examine two questions about the implementation of our technique.

1. **How effective is mwp$_\ell$ at discovering postconditions in general?** We execute mwp$_\ell$ against four benchmark suites that include a rich set of benchmarks from complexity theory to loop invariant inference. Three of the suites are design-independent of the applied analysis technique.
2. **What is the concrete impact of the new theoretical enhancements?** We hypothesize that the introduced enhancements are critical for extending the utility of the flow calculus. To quantify this improvement, we compare performance of mwp$_\ell$ and mwp$_f$ across the four benchmark suites.

Setup: benchmarks, metrics, and environment. We consider four benchmark suites, of numerical C loops, and 620 total problem (detailed in Sect. C). The complexity suite is mainly linear time complexity problems whose termination behavior is unknown. The linear [32] and non-linear [25,34] suites come from loop invariant inference literature. The non-linear suite includes classic numerical algorithms like geometric series and divisor computations. The mwp suite [7] problems are designed pose challenges to analyses based on the flow calculus of mwp-bounds. All suites include branching statements and nondeterministic control expressions that simulate external function calls. The complexity and non-linear suites have problems with sequential and nested loops. As evaluation metrics, we recorded the count and kind of obtained postconditions (i.e., mwp-bounds). We also recorded all meta-data collected by pymwp like program statistics. We set the timeout to 10 seconds. Without timeouts the metrics are deterministic. We ran all experiments on commodity hardware; on a native 10-core macOS 15.3.2 M1 arm64 host with 16 GB of RAM. All experiment results are in Table 2.

Table 2: Experiment results for mwp_f and mwp_ℓ with totals and (mean). *Loops*/*functions* shows the number of analyzed instances and suite coverage (%). *Bounds* is the number of inferred postconditions, with the mean is relative to analyzed *variables*. The *mwp*-columns show a breakdown of postconditions by bound form. The number of unbounded variables is shown in column ∞.

Analyzer	**Suite**	**Loops**	**Variables**	**Bounds**	**m**	**w**	**p**	**∞**
mwp_ℓ	Complexity	623 (.84)	1407	988 (.70)	567	49	372	419 (.30)
(ours)	Linear	49 (1.0)	103	80 (.78)	22	7	51	23 (.22)
	Non-linear	43 (.90)	172	107 (.62)	48	8	51	65 (.38)
	mwp	30 (1.0)	105	77 (.73)	38	30	9	28 (.27)
	Suite	**Functions**	**Variables**	**Bounds**	**m**	**w**	**p**	**∞**
mwp_f	Complexity	399 (.79)	1153	616 (.53)	330	6	280	537 (.47)
(prior)	Linear	49 (1.0)	131	95 (.73)	38	1	57	36 (.27)
	Non-linear	29 (.78)	159	60 (.38)	14	3	43	99 (.62)
	mwp	30 (1.0)	105	66 (.63)	28	22	16	39 (.37)

7.1 Inference generalizability

Across the four suites, mwp_ℓ succeeds at analyzing 84%-100% of the loops. It finds postconditions for 62%-78% of variables occurring in those loops. The postconditions are different in form from most complexity analyzers (refer to [22,7]). Most complexity analyzers are essentially restricted to linear arithmetic [22]. Encouragingly, mwp_ℓ shows success also on the non-linear suite. The results are positive because the suite contains complex arithmetic and natural algorithms. The linear suite expectedly provides more postconditions, since the suite is easier. On the complexity problems, that dominate in problem quantity, mwp_ℓ already bounds 70% of variables. The number depends largely on how loop boundedness is decided. We expect that, after improving the current mechanism, mwp_ℓ could produce even more postconditions.

We observe some limitations w.r.t. expressivity. First, loops with unsupported syntax are not analyzed. For example, some variables in the complexity suite are updated by a random integer generator. Such operations are inherently outside the guarantees that can be provided by the flow calculus. When the growth rate of some variables is truly beyond polynomial, it is not expressible as mwp-bounds. Finally, sometimes the polynomial describing the variable value growth may be too complicated to express. Based on the findings, further increase in expressivity should be one of the main directions of future research. In summary, mwp_ℓ succeeds at analyzing most loops, and it generates postconditions for most variables in those loops.

7.2 Impact of theoretical enhancements

We compare mwp_ℓ and mwp_f to quantify the impact of the enhancements introduced in this paper. As practical implementations of the flow calculus,

mwp_f represents the state-of-the-art in the analysis capabilities. Since the two analysis modes differ on program scopes, not all results are comparable. However, we pre-processed the mwp-suite such that it is fully comparable. The mwp-suite results confirm that the paper enhancements improve abilities to infer postconditions. On the mwp suite, mwp_ℓ bounds 73% of variables compared to 63% by mwp_f. Further, mwp_ℓ finds optimal bounds, which is reflected in the distribution of bound forms. The other suites are comparable through the statistical means. Particularly on the non-linear suite, mwp_f bounds only 38% of variables. This is because one failing variable causes failure of all variables. On the same suite, mwp_ℓ bounds 62% variables due to its enhanced failure handling.

8 Conclusions and Future Directions

Assistance in specification inference is paramount to increase adoption of formal methods in practice. The paper has presented how to repurpose a complexity-theoretic analysis to postcondition inference. This required four new enhancements: (i) projecting the analysis on individual variables, (ii) introducing an evaluation strategy to obtain optimal mwp-bounds, (iii) improving derivation failure-handling to increase analysis expressiveness, (iv) and adopting the analysis to a new use case, i.e., formal verification. A comparison study shows our technique offers complementary strengths among the related inference approaches. The theory is materialized in an implementation, mwp_ℓ. Our experiments show that the paper enhancements improve the flow calculus and make it applicable toward uses in formal verification.

Future directions. Although we have extended the flow calculus capabilities, multiple directions for future work remain, and many emerge from this work. The two main directions concern enriching the analysis expressiveness and precision. E.g.,, leveraging assumptions (if available), tracking variable immutability, and accounting for control expression would generate more precise specification conditions. Currently, polynomial p-flows are not allowed inside unbounded loops. Discovering ways to relax this restriction would improve expressiveness and yield more postconditions. Due to the complexity-theoretic origins, the flow calculus targets polynomial *upper* bounds. For verification, it would be useful to extend the technique to also cover lower bounds, and assign bounds on exponential growth. On the practical side, our analysis does not cover division operator and requires expanding operations to binary form. 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 discovering its potential uses. The analysis could already be implemented as a developer plug-in to assist in writing specifications. In future research, we will consider extending the capabilities of the flow calculus. We are generally curious about whether similar solver-free syntactic analyses could be designed to infer *other* specification conditions. Another ongoing project is to formally verify the flow calculus theory. The results of this paper provide important justification to the formalization effort.

References

1. Daikon (2024), https://plse.cs.washington.edu/daikon
2. Duet static analyzer (2024), https://github.com/zkincaid/duet
3. KoAT2 (2024), https://github.com/aprove-developers/KoAT2-Releases
4. Termination Problem Database (2024), https://github.com/TermCOMP/TPDB
5. Alagarsamy, S., Tantithamthavorn, C., Aleti, A.: A3Test: Assertion-Augmented Automated Test case generation. Information and Software Technology **176**, 107565 (12 2024). https://doi.org/10.1016/j.infsof.2024.107565
6. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity. In: 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). LIPIcs, vol. 228, pp. 26:1–26:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.26
7. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: pymwp: A Static Analyzer Determining Polynomial Growth Bounds. In: Automated Technology for Verification and Analysis. pp. 263–275. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-45332-8_14
8. Baillot, P., Barthe, G., Lago, U.D.: Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs, pp. 203–218. Springer Berlin Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_15
9. ter Beek, M.H., Chapman, R., Cleaveland, R., Garavel, H., Gu, R., ter Horst, I., Keiren, J.J.A., Lecomte, T., Leuschel, M., Rozier, K.Y., Sampaio, A., Seceleanu, C., Thomas, M., Willemse, T.A.C., Zhang, L.: Formal methods in industry. Form. Asp. Comput. **37**(1) (12 2024). https://doi.org/10.1145/3689374
10. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing Runtime and Size Complexity of Integer Programs. ACM Transactions on Programming Languages and Systems **38**(4), 1–50 (8 2016). https://doi.org/10.1145/2866575
11. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear Invariant Generation Using Non-linear Constraint Solving. In: Computer Aided Verification. pp. 420–432. Springer Berlin Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 84–96. POPL '78, ACM Press (1978). https://doi.org/10.1145/512760.512770
13. Cyphert, J., Kincaid, Z.: Solvable Polynomial Ideals: The Ideal Reflection for Program Analysis. Proc. ACM Program. Lang. **8**(POPL) (1 2024). https://doi.org/10.1145/3632867
14. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. ACM SIGPLAN Notices **48**(10), 443–456 (10 2013). https://doi.org/10.1145/2544173.2509511
15. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1), 35–45 (12 2007). https://doi.org/10.1016/j.scico.2007.01.015
16. Furia, C.A., Meyer, B.: Inferring Loop Invariants Using Postconditions, pp. 277–300. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-15025-8_15
17. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing Program Termination and Complexity Automatically

with AProVE. Journal of Automated Reasoning **58**(1), 3–31 (10 2016). https://doi.org/10.1007/s10817-016-9388-y
18. Giesl, J., Lommen, N., Hark, M., Meyer, F.: Improving Automatic Complexity Analysis of Integer Programs, pp. 193–228. Springer International Publishing (2022). https://doi.org/10.1007/978-3-031-08166-8_10
19. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (10 1969). https://doi.org/10.1145/363235.363259
20. Jones, N.D., Kristiansen, L.: A flow calculus of *mwp*-bounds for complexity analysis. ACM Transactions on Computational Logic **10**(4), 28:1–28:41 (8 2009). https://doi.org/10.1145/1555746.1555752
21. Karr, M.: Affine relationships among variables of a program. Acta Informatica **6**(2), 133–151 (1976). https://doi.org/10.1007/BF00268497
22. Lommen, N., Giesl, J.: Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs, pp. 3–22. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-43369-6_1
23. Molina, F., Ponzio, P., Aguirre, N., Frias, M.: EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1223–1235. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00112
24. Moyen, J.Y.: Implicit Complexity in Theory and Practice (2017), https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf
25. Nguyen, T., Antonopoulos, T., Ruef, A., Hicks, M.: Counterexample-guided approach to finding numerical invariants. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 605–615. ESEC/FSE'17, ACM (2017). https://doi.org/10.1145/3106237.3106281
26. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: DIG: A dynamic invariant generator for polynomial and array invariants. ACM Transactions on Software Engineering and Methodology (TOSEM) **23**(4), 1–30 (9 2014). https://doi.org/10.1145/2556782
27. Nguyen, T., Nguyen, K., Duong, H.: Syminfer: inferring numerical invariants using symbolic states. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. p. 197–201. ICSE '22, ACM (5 2022). https://doi.org/10.1145/3510454.3516833
28. Popeea, C., Chin, W.N.: Inferring Disjunctive Postconditions. In: Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues. pp. 331–345. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-77505-8_26
29. Rosenblum, D.S.: A practical approach to programming with assertions. IEEE Transactions on Software Engineering **21**(1), 19–31 (1995). https://doi.org/10.1109/32.341844
30. Ryan, G., Wong, J., Yao, J., Gu, R., Jana, S.: CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In: 8th International Conference on Learning Representations, ICLR 2020. OpenReview.net (2020), https://openreview.net/forum?id=HJlfuTEtvB
31. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 318–329. POPL04, ACM (2004). https://doi.org/10.1145/964001.964028
32. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning Loop Invariants for Program Verification. In: Advances in Neural Information Processing Systems 31. , vol. 31, pp. 7762–7773. NeurIPS (2018), https://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification

33. Yao, J., Ryan, G., Wong, J., Jana, S., Gu, R.: Learning nonlinear loop invariants with gated continuous logic networks. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 106–120. PLDI '20, ACM (2020). https://doi.org/10.1145/3385412.3385986
34. Yu, S., Wang, T., Wang, J.: Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 175–187. ISSTA '23, ACM (2023). https://doi.org/10.1145/3597926.3598047
35. Zhang, Y., Mesbah, A.: 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). https://doi.org/10.1145/2786805.2786858

A Application to Program Verification

The following example shows how to complete an implementation with specification conditions, in the verification-aware Dafny programming language. The postconditions are those inferred by our analysis.

Concrete program verification requires two additional steps. First, recording the initial variable values (L4). This is simply a matter of creating copies of variables. Recording the initial values enables referring to them in the postconditions. Second, we add the constants (L14–16) omitted in the postconditions of our analysis. Although this step requires manual effort, is it significantly easier to "fill in" the constants, than to infer full assertion clauses. Maintaining human oversight in this step has an additional benefit. It forces to check that the assertions are sensible. If the implementation has a bug—e.g., variable grows exponentially when it should not—the bug becomes detectable during this step. A fully automatic technique would not alert to the issue.

After adding invariants, the Dafny verifier immediately constructs a proof, which confirms that the assertions always hold. Generating suitable inductive loop invariants is a challenge for the related works that specialize in inductive invariant inference.

Listing 1.6: LucidLoop verified in Dafny.

```
0  method LucidLoop(X1: nat, X2: nat, X3: nat, X4: nat, X5: nat){
1    // precondition: all variables are nat-type
2
3    // for bookkeeping -- record initial values
4    var X1', X2', X3', X4', X5' := X1, X2, X3, X4, X5;
5
6    for i := 0 to X1'
7      // invariants (omitted)
8    {
9      X3' := X2' * X2' + X5';
10     X4' := X4' + X5';
11   }
12
13   // postconditions
14   assert X1'≤ X1;                      // linear
15   assert X2'≤ X2;                      // linear
16   assert X5'≤ X5;                      // linear
17   assert X3'≤ Max(X3, X2*X2+X5);       // weak polynomial
18   assert X4'≤ X4+X1*X5;                // polynomial
19 }
```

B Technical Details of Analyzer Comparison

B.1 Executing the analyzers

We analyzed the programs in Listings 1.7–1.9 as follows.

mwp$_\ell$. We run pymwp v0.6.0 in the loop analysis mode.

```
pymwp [program].c --mode L --strict
```

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. Transition ideals is implemented as linear and quadratic simulation modes.

```
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
```

B.2 Analyzer scopes

```
int main(int X, int Y, int Z) {
  for(int i=0; i<Y; i++)        KoAT
    X = X + Z;                  mwp_ℓ
  assert(...);                  Duet
  return X;                     Daikon
}
```

Fig. 8: Postcondition analysis scopes. The analyzers focus on complementary program regions identified by the different colors. KoAT focuses on the variables that control loop iteration. mwp_ℓ analyzes variables inside the loop body. Duet infers loop summaries based on available assertions. However, the problem assumes no assertions exist yet. Daikon infers likely invariants of the **return** variable, while treating the function internals as opaque.

B.3 Comparison programs

Listing 1.7: 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 1.8: 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 1.9: Linear #02

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

These are programs of Fig. 7 expanded with headers and return statements. For Duet analysis, **loop** must be renamed to `main`. Having a `main` function raises an

error with KoAT. Daikon expects adding a separate `main` method with calls to `loop`. In Listing 1.9, the unsupported `assume` should be omitted before analyzing it with KoAT.

C Details of Experimental Evaluation

The complexity suite is the "Complexity C Integer" suite from the Termination Problem Database version 11.3 [4], This suite is used in the annual Termination and Complexity Competition.

The linear suite [32] contains inference problems for linear loop invariants. The problems are pre-annotated with assertions (these have no impact on our analysis). We excluded 9 benchmarks that are known to be invalid [30, Appendix G] as they violate the specified assertions. We also unified benchmarks that have the same precondition and loop, as they are identical for the purpose of *postcondition* inference, ending with 49 benchmarks in total.

The nonlinear suite [25] (also referred to as NLA-suite in literature) is an extended formulation of the suite, with the additional problems coming from [34].

The mwp suite [7] is designed specifically to be challenging for the flow calculus of mwp-bounds, with complex data flows and arithmetic operations. To obtain strictly comparable results between mwp_ℓ and mwp_f, as they scope variables differently, we exclude nested loops and loopless benchmarks.

The statistics of the suites are summarized in Table 3. A single benchmark can contain multiple functions, loops, and sequential and/or nested loops. Due to differences in the targeted program scopes between mwp_ℓ and mwp_f, the variable counts are specified by scope. Loop-scoped variables include loop guards and variables in the loop body. Function-scoped variables contain all parameters and variables in the function body. We modified the benchmarks by expanding n-ary expressions to binary form to match the input language of pymwp. Since the boundedness check is still rudimentary, some loop conditions were rewritten in detectable form. A more robust approach would use a specialized compiler.

Table 3: Benchmark suite characteristics summarized by count and (mean).

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