
# Exact Loop Bound Analysis

DANIEL RILEY, Florida State University, USA
GRIGORY FEDYUKOVICH, Florida State University, USA

There are many state-of-the-art techniques for loop bound analysis. Most of them target an upper bound for a given program, and others find a lower bound. *Exact* bound analysis still remains largely unexplored, but it offers new applications. To compute an *exact* bound for a program it is necessary to reason about the possible values the program's inputs can take and how they relate to each other. Since inputs can vary on any given execution of the program, it makes the problem of computing an exact bound challenging. In this work, we present a new approach to find an exact bound by way of *precondition synthesis* which iteratively considers under-approximations of a program under which the bound can be precomputed over initial values of program variables. For each precondition, our approach synthesizes a function over program variables such that when the function is applied to the initial values of the program variables, its output is an exact bound for the program. We reduce the precondition synthesis problem to that of *safety verification* which lends its correctness guarantees to the exact bounds we compute. Our technique has been implemented in a tool called ELBA, and we show that it is effective on a set of challenging single loop benchmarks under Linear Integer Arithmetic.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Logic and verification**; **Automated reasoning**;

Additional Key Words and Phrases: satisfiability modulo theories, automated bound analysis, automated safety verification, inductive invariant synthesis; functional synthesis

**ACM Reference Format:**
Daniel Riley and Grigory Fedyukovich. 2025. Exact Loop Bound Analysis. *Proc. ACM Program. Lang.* 9, PLDI, Article 220 (June 2025), 24 pages. https://doi.org/10.1145/3729323

## 1 Introduction

Modern computing landscapes are increasingly dominated by large-scale data centers and cloud computing infrastructures, where service efficiency is closely linked to energy consumption and workload distribution. Consequently, the design of next-generation warehouse-scale computing services presents a significant challenge for the continued advancement and economic viability of these systems. Design considerations extend beyond hardware and architectural advancements to encompass the crucial role of dependable software infrastructure. A common paradigm in warehouse-scale services involves distributing tasks across many systems to leverage parallel processing. However, a notable source of inefficiency arises from the idle time incurred when processes are forced to wait for the completion of dependent tasks. Improved estimations of the computational resources (e.g., cycles, iterations, and execution time) required by distributed processes can mitigate this wasted resource by enabling more efficient scheduling and resource allocation [26].

Current approaches to resource analysis focus primarily on determining upper bounds to estimate the energy, memory, or execution time a program requires. For instance, the symbolic resource

Authors' Contact Information: Daniel Riley, Florida State University, Tallahassee, USA, driley@cs.fsu.edu; Grigory Fedyukovich, Florida State University, Tallahassee, USA, grigory@cs.fsu.edu.


ACM 2475-1421/2025/6-ART220
https://doi.org/10.1145/3729323

bound analyses in [20] computes an upper bound from "cost equations" for code fragments of the program under analysis. The work of [32] generates intervals of input data sizes to determine where resource usage assertions hold or fail. Other analyses [6, 18, 22, 47] target an upper bound on a program's cost, which is most useful in proving termination, rather than in the context of optimizing energy consumption or memory allocation. In an attempt to find tighter upper bounds, the approach of [2] computes a bound on a *worst case* execution of a program.

An exact bound, compared to an upper bound, provides a precise understanding of the execution cost of a program, in terms of the possible values of its input variables. This precision can be helpful in the ability to more tightly constrain a program that must run on limited hardware, and it can provide guarantees on the execution of the program. We are *the first to formulate the problem* of *Exact Loop Bound Analysis* (ELBA) that is concerned with synthesizing a function to compute the number of iterations of a loop before its execution. Then, we present a first-of-its-kind method for the ELBA problem. As far as we have seen, no other attempts to discover exact bounds in this way have been attempted.

Exact bounds can be applied in a similar way as upper bounds. Furthermore, they unlock novel applications such as relational verification and equivalence checking [8, 46]. For instance, verifying that two programs produce identical outputs for the same inputs can be significantly enhanced by computing and utilizing the exact symbolic bounds of both programs [24] to facilitate the construction and analysis of perfectly aligned lockstep compositions. Moreover, knowing such bounds can reveal the precise input conditions under which a program demonstrably achieves optimal performance in terms of execution speed.

Another application for an exact bound of a program is analysis of a program for *existential non-termination*, e.g., when searching for a set of initial conditions that allow the loop to enter a trap and diverge. Those initial states can be difficult to find however. The key property about a program that would help in identifying these conditions is that the computation of a program's exact bound turns negative in some cases. This would enable the determination of the boundary between a subset of inputs, delineating when the loop will execute a finite number of iterations or run forever. Knowing potential non-terminating conditions in a program would further aid in debugging and significantly narrow the scope for potential fixes.

We formulate the ELBA problem as an instance of a *precondition synthesis* problem which is concerned with discovering a suitably weak initial condition over a program's inputs under which the program never reaches an error state. To represent the total number of remaining loop iterations (called transitions throughout the paper) at an arbitrary program state, we introduce a fresh variable called a *transition counter* that is decremented in the loop body. Our precondition synthesis aims to find an initialization of the transition counter such that it ultimately reaches a value of zero any time when the loop finishes. That is, this initialization requires a function to compute the precise number of transitions from concrete input values only, i.e., before the loop has started. Analysis of this kind is inspired by techniques to prove termination used in previous works, e.g., [11, 23, 47], namely to find the preconditions under which the instrumented transition counter's value is above zero at the end of the program. In contrast to those preconditions that can be imprecise, our analysis targets the most conservative precondition.

We present an approach to solving the ELBA problem that proceeds iteratively, by creating symbolic unrollings of the program with the instrumented transition counter, finding concrete values of every variable, including the transition counter, learning relationships among variables based on these data, and synthesizing preconditions under which these relationships hold. Possible program executions are considered one at a time. To begin the bound search, an unrolling is encoded into a logic formula that is then sent to a Satisfiability Modulo Theories (SMT) solver. The solver provides a model of the formula, which is used to gather concrete data of each step in

the unrolling. The data is then used to discover relationships among program variables, similar to [16, 36, 37, 40]. We specifically target relationships between the transition counter and the input variables to synthesize a function over the input variables for computing the exact number of transitions the program performs under certain preconditions. The ELBA approach is currently limited to linear integer arithmetic (LIA), due to the data analysis we use, and to single loop programs.

We seek a precondition that defines the program states for which the bound discovered from the data applies. Our goal is to make the precondition as weak as possible, encompassing the broadest set of valid input conditions. We present a novel *abduction*-based technique to synthesize a precondition under which the previously computed loop bound is correct. Abduction queries are posed for a finite number of unrollings of increasing lengths. The result of each query acts as a *bounded precondition*. These preconditions are gathered together and generalized to cover a possibly infinite number of program executions. The algorithm terminates when all the discovered preconditions describe all possible initial configurations. We validate the synthesized bound under each precondition using safety verification via the discovery of a safe inductive invariant. The existence of an inductive invariant guarantees that the value of the counter is zero any time when the loop finishes.

Our next contribution enables handling a challenging scenario of loops with multiple phases. In these cases, abduction results in a disjunctive formula, complicating the generalization. This disjunction arises from the necessity to encode each potential execution of the loop, with each disjunct representing a unique sequence of iterations and branch selections [40, 44]. The increased difficulty for precondition synthesis arises from the need to describe not just the input conditions but also how initial values affect the possible paths through the loop. To address this complexity and derive a general precondition, we systematically consider all possible combinations of disjuncts derived from each abductive query. After accumulating the combination of disjuncts, the algorithm generalizes each set into a candidate precondition that encompasses the execution states that terminate when the bound holds zero. This exhaustive exploration of combinations allows us to weaken the precondition and to cover more states applicable to the bound.

Our *Exact Loop Bound Analysis* technique has been implemented in the FreqHorn framework [16], yielding a new tool called ELBA. It leverage FreqHorn's safety verification application for the purposes of exact bound synthesis. The underlying SMT solver used by ELBA is Z3 [13]. We compare against the tools Loopus, KoAT, and FreqTerm [6, 18, 47]. Each is a bound analysis tool that finds upper bounds on program loops. These bounds are often loose upper bounds that do not capture the number of loop iterations with precision. Our benchmark set is taken from the *Termination Problem Data Base*[1], and includes a subset of the benchmarks from both Loopus, FreqTerm, and FreqHorn. These benchmarks consist of single loop programs, under linear integer arithmetic, and are deterministic within their loop. Input variables, however, may have non-determinism in their initial values, and this adds to the challenge of synthesizing an exact bound. Non-determinism in the input values means that each possible "configuration" of values must be considered. Our experimental results show that ELBA outperforms these tools by not only finding exact bounds, but also by producing a result on programs that other analysis tools cannot. ELBA found exact bounds for 62 out of the 75 benchmark programs, a result that is on the same order as the other bound analysis tools, Loopus and KoAT, which found 59 and 64 upper bounds on the benchmark set, respectively. Although ELBA is on average slower to reach its results than Loopus and KoAT, it comes with the exactness guarantees while the other tools do not.

To sum up, our contributions in this work are:

[1] https://github.com/TermCOMP/TPDB/tree/master/C_Integer.

- We have formulated the novel problem of synthesizing a function to compute the number of loop iterations precisely based on the input variables.
- We propose several use cases for the exact loop bound analysis, where only exact bounds would be applicable since upper bounds are often imprecise.
- We have developed a *first-of-its-kind* solution to compute exact loop bounds. Our algorithm leverages symbolic-reasoning techniques including precondition synthesis, abduction, and inductive-invariant generation to compute, classify, and prove the correctness of an exact bound *fully automatically*.
- Our novel technique computes bounds and their precondition for loops with non-deterministic initial values on input variables and branching control flow. This is challenging because it requires reasoning about all possible initial conditions a program can have.
- We have implemented our technique in a tool called ELBA, and we have evaluated it on a set of deterministic, single-loop programs under linear integer arithmetic. We show that the capability of ELBA is superior to other state-of-the-art bound analysis tools.

The paper continues with Sect. 2 which outlines the technique and introduces our motivating examples. We offer Sect. 3 for a brief background. Sect. 4 introduces our approach in the context of single loop programs. We introduce the core algorithms and explain them in the context of several examples. We discuss our experimental evaluation in Sect. 5 and related work in Sect. 6. The work is summarized in Sect. 7.

## 2 Motivation and Examples

In this section, we introduce our analysis on a set of examples that gradually become more challenging and interesting. Whenever applicable, we also give an intuitive comparison to existing related approaches and introduce new applications. First, we show an example to exhibit the use of inductive invariants to prove the correctness of the exact bound. Next, the example highlights ELBA's ability to reason about each of a program's possible execution configurations. It is possible that a loop can be dependent on input variables whose values are not know until runtime. ELBA can figure out the exact bound (i.e., which variable a loop primarily depends on) for each of the possible scenarios in these examples. Similarly, ELBA can find exact bounds for programs with branching control flow. These examples also require analysis over many possible execution scenarios, each of which must have an exact bound computed.

*Inductive invariants to prove correctness.* The program in Fig. 1 (a) has one variable with an unknown initial value (let $y_{in}$) and a loop that decrements the variable by one. We present this small example to give a high level view of how our exact loop bound analysis is performed. At first glance one could mistakenly think that the loop iterates $y_{in}$ times, but upon closer examination it becomes clear that the number of iterations is $max(y_{in}, 0)$. We trust the reader of this paper is able to *prove* that this function is indeed correct. One way to do so is illustrated in Fig. 1 (b): the program is instrumented with a fresh variable `tc` that is decremented in each iteration and must be equal to zero at the end of the loop. Then, we can use any program verifier that works by inferring an *inductive invariant* to prove that the assertion holds. In our case, invariant `tc = max(y, 0)` ensures that when the loop has terminated (and thus condition `y > 0` no longer holds), `tc` evaluates to zero.

*Analysis over input relationships.* The program in Fig. 1 (c) (which is already shown with the counter *tc*) has two decrementing variables and a conjunctive loop guard. In this example, we show that finding an exact bound, even on a small program, is more challenging than it might seem in Sect 4.2. An approach based on static analysis can infer *an upper bound* on the number of iterations

```
int y = nondet();
while (y > 0){
  y--;
}
```
(a)

```
int y = nondet();
int tc = max(y, 0);
while (y > 0){
  y--;
  tc--;
}
assert(tc == 0);
```
(b)

```
int y = nondet();
int z = nondet();
int tc = f(y, z);
while (y > 0 && z > 0){
  y--;
  z--;
  tc--;
}
assert(tc == 0);
```
(c)

Fig. 1. Examples to begin with: (a) straightforward loop with a single decrementing variable, (b) same one instrumented with a fresh counter `tc`, (c) slightly more interesting loop with two original variables and `tc`.

```
int i = nondet();
int tc = f(i);
while (i == 7 ||
       i == 5040){
  i = i!;
  tc--;
}
assert(tc == 0);
```
(a)

```
int x = nondet();
int n = nondet();
int tc = f(x, n);
while (x != n){
  x--;
  tc--;
}
assert(tc == 0);
```
(b)

```
int x = 0;
int y = 0;
int m = nondet();
int n = nondet();
int tc = f(x, y, m, n);
while (x < n){
  if(y < m) y++;
  else x++;
  tc--;
}
assert(tc == 0);
```
(c)

Fig. 2. (a) A loop that iterates only one or two times but must reason about a factorial, (b) a non-terminating example, (c) a loop with two phases.

(and thus an initial value of *tc*) to be $max(y, z, 0)$. Upper bounds are useful when we want to prove termination, and thus an upper bound like $2 \times max(y, z, 0) + 1000$ would work too[2], albeit it is less precise. The question we address in this paper is whether we can go further and infer the *most precise* (i.e., exact) bound, which enables an a priori calculation of the number of loop iterations under each concrete input. It turns out that such a bound for this example, expressed by function $f(y, z)$ is $\texttt{if } (y > 0 \texttt{ and } z > 0) \texttt{ then } min(y, z) \texttt{ else } 0$. We prove it by finding an inductive invariant using ELBA's internal verification algorithm.

*Applications.* The advantage of knowing this exact function is that we can apply it to any concrete input and compute a concrete number of iterations *without executing the program*. For instance, if initially $y = 5$ and $z = 9$, then the loop iterates five times – we would not be able to infer this knowing only an upper bound. Another interesting problem that our analysis can be applied to is inferring *conditional equivalence* of execution costs for two programs: what is the condition under which two programs iterate exactly the same number of times? For the programs in Fig. 1 (b-c), we can infer such a condition by 1) equating the exact bounds, and 2) abducing the produced equality.

[2]Here and later we assume algebraic integers.

Thus, the problem is to find the weakest $\psi$ such that the following implication is valid:

$$\psi(\texttt{y},\texttt{z}) \implies \mathit{max}(\texttt{y},\ \texttt{0}) = \texttt{if (y > 0 and z > 0) then } \mathit{min}(\texttt{y},\ \texttt{z}) \texttt{ else 0}$$

The solution to the problem is $\psi = \texttt{z} \geq \texttt{y}$ that can be obtained by an abduction solver, i.e., any SMT solver that supports quantifier elimination in linear integer arithmetic. Intuitively, this solution can be interpreted as if the program in Fig. 1 (c) was given z greater or equal than y, then its loop would iterate exactly the same number of times as the loop in the program in Fig. 1 (b). Again, we are able to infer this without conducting any specialized *relational verification* [24], but rather by operating on the inferred functions.

*Discrete iterations.* The program in Fig. 2 (a) that has a complex nonlinear computation in the loop body (for simplicity, we still count it as a single transition, but this of course can be made more precise): `if (i == 7) then 2 else if (i == 5040) then 1 else 0`. The challenge of finding this bound is attributed to non-existence of a simple linear relation between `i` and `tc`. However, our analysis can figure out that the loop may never iterate three times or longer, and the solution captures the boundary between the other possible two initial states precisely. At a high level, it is done by checking which conditions the loop is entered. The loop guard allows only two cases where the loop executes, and ELBA can determine precise conditions for them. With this information ELBA can reason about the number of iterations the loop performs by looking at the models produced by the SMT solver from a loop unrolling. The unrolling includes the transition counter *tc* and the analysis concludes that the loop can only iterate one or two times, and it matches those cases together with the preconditions $i = 5040$ and $i = 7$ respectively.

*Observing non-termination.* It could be the case that a loop may not terminate under certain initial values of its input variables. Program in Fig. 2 (b) is one such example. If the initial values of `x` and `n` are such that `x > n`, then the program enters the loop and eventually terminates. In the case when `x` and `n` are initially equal, the loop is not executed. However, the case when `x < n` is important to consider. In this case, the program enters the loop, but it does not terminate since `x` moves further away from `n`. What is interesting here is that our analysis can find the terminating configuration, and the bound, for the loop. While non-termination is not explicitly checked, a function can be checked for the scenario under which it produces a negative value. A negative value for the transition counter is clearly nonsense, but it can be interpreted as a configuration of the program's initial values that leads to the non-termination of the loop. In the case of this program, the exact bound is: `if (x > n) then (x - n) else if (x = n) then 0 else -1`. The $-1$ captures the case when the loop does not terminate.

*Handling branching control flow.* The loop in Fig. 2 (c) adds a level of complexity to the analysis for an exact bound. In this loop, there are two branches, and which branches are visited depends on the value of `m`. If the initial value of variable `y` is less than the initial value of `m`, the loop enters the first branch and iterates until `y` equals `m`. When that condition is fulfilled, the second branch executes until `x = n`.

Another execution of the program could have initial values such that the first branch is not visited at all. In this case, the loop only visits the second branch and terminates when `x = n`. There is a third case to consider in which the value of `n` is less than or equals zero initially. In this case, the loop does not iterate at all. The challenge is figuring out exactly how many iterations the loop can do in the three cases described above. ELBA considers each case one at a time and finds an exact bound for each possible execution of the program separately.

In the case where the initial values of the input variables are such that n is greater than x but y is greater than or equal to m then only the second branch is visited. Then ELBA reasons that under this condition the loop must iterate n times, and it also finds the precondition described above. When the input is such that both branches are visited then ELBA finds that the exact bound is m + n. Then the exact bound for this program, after the final case is examined, looks like:

```
if (n > x ∧ m > y) then m + n
else if (n > x ∧ m ≤ y) then n else 0
```

## 3 Background

The *Satisfiability Modulo Theories* (SMT) problem is concerned with determining the existence of a satisfying assignment to the variables in a given formula $\varphi$. The assignment is referred to as a *model*, and we write $\mathcal{M} \models \varphi$ to state that $\varphi$ is satisfiable (i.e., model $\mathcal{M}$ exists). If every model of $\varphi$ also satisfies a formula $\psi$ then $\varphi$ is logically stronger than $\psi$ (written $\varphi \implies \psi$). An unsatisfiable formula $\varphi$ is denoted $\varphi \implies \bot$. By $\varphi(x)$ we denote a formula over free variables $x$. In term/formula $\varphi$, a replacement of $x$ with $y$, we write $\varphi[y/x]$. Similarly, for a set of variables $X$ and a mapping $\mathcal{M}$, $\varphi[\mathcal{M}/X]$ denotes the replacement of $x_1, x_2, \ldots \in X$ respectively by $\mathcal{M}(x_1)$, $\mathcal{M}(x_2)$, etc. Throughout the paper, by $\Lambda$ we denote a set of all anonymous functions, e.g., $\lambda x \,.\, x + 1$ that returns an incremented argument.

We represent programs as *transition systems* and may refer to a program with either term.

*Definition 3.1.* A *transition system* is a tuple $\langle V, init, tr \rangle$. $V$ is a sequence of state variables, $init \in \Lambda$ is a one-state predicate that represents a set of *initial states*, and $tr \in \Lambda$ is a two-state predicate that represents a *transition relation* for $V$ and $V'$, i.e., state variables and next-state variables respectively.

*Example 3.2.* The transition system for the program in Fig. 2 (c), without the counter, is as follows.

$$V \stackrel{\text{def}}{=} \{x, y, m, n\} \qquad init \stackrel{\text{def}}{=} x = 0 \land y = 0 \qquad tr \stackrel{\text{def}}{=} x < n \ \land \ x' = ite(y < m, x, x + 1) \land y' = ite(y < m, y + 1, y) \land n' = n \land m' = m$$

*Definition 3.3.* Given a *transition system* $\langle V, init, tr \rangle$ and a set of error states *bad*, a *safe inductive invariant* is a formula *inv* over free variables $V$ that meets the following conditions (initiation, consecution, and safety, respectively):

$$init(V) \implies inv(V) \tag{1}$$

$$inv(V) \land tr(V, V') \implies inv(V') \tag{2}$$

$$inv(V) \land bad(V) \implies \bot \tag{3}$$

The *safety verification problem* for a given transition system and a set of error states is concerned with determining an existence of interpretation of *inv* that makes the implications in Def. 3.3 valid. In other words, *inv* represents a program property that covers every initial state (implication 1), is closed under the transition relation (implication 2), and does not cover any of the *bad* states (implication 3). We often refer to *safe inductive invariants* as simply *invariants* throughout the paper.

The *precondition synthesis problem* for some $T = \langle V, init, tr \rangle$ and some formula *bad* is concerned with finding a formula $p$ such that the safety verification problem $\langle V, init \land p, tr, bad \rangle$ has a solution. Because every precondition synthesis problem can be trivially solved by letting $p$ be $\bot$, we naturally target an adequately weak (but it may not be the weakest) $p$. With these restrictions in the precondition a safe inductive invariant can be found using standard methods to invariant synthesis; however it is much more challenging to find a good precondition itself and to prove it is the weakest possible.

**First possible execution**

| x | y | m | n | tc |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 2 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Second possible execution**

| x | y | m | n | tc |
|---|---|---|---|---|
| 0 | 0 | 0 | 2 | 2 |
| 1 | 0 | 0 | 2 | 1 |
| 2 | 0 | 0 | 2 | 0 |

Fig. 3. An example of two data matrices that correspond to two different possible executions of the program in Fig 2 (c).

*Example 3.4.* An interpretation of $p$ must restrict the initial value of the variables to make the program from Fig. 2 (c) safe.

$$\begin{aligned} p(x,y,m,n,tc) \wedge x = 0 \wedge y = 0 &\implies inv(x,y,m,n,tc) \\ inv(x,y,m,n,tc) \wedge (x < n) \wedge x' = ite(y < m, x, x+1) \wedge & \\ m' = m \wedge n' = n \wedge y' = ite(y < m, y+1, y) \wedge tc' = tc - 1 &\implies inv(x',y',m',n',tc') \\ inv(x,y,m,n,tc) \wedge (x \geq n) \wedge \neg(tc = 0) &\implies \bot \end{aligned} \tag{4}$$

Example 3.4 gives some motivation to find a precondition under which a property would hold (and an invariant would exist). Specifically, there must be a precondition that restricts x, y, and z so that it is not possible to violate the safety specification. A possible such precondition has x and y equal when x and either y or z is less than zero, or when all three are non-negative and y is less than z. With these restrictions in the precondition a safe inductive invariant can be found using standard methods to invariant synthesis; however it is much more challenging to find a good precondition itself and to prove it is the weakest possible. A valid interpretation of $p$ restricts the initial values of x and y to both be greater than zero.

We use symbolic *data learning* to discover relationships among program variables $V$. It gathers traces of data as finite unrollings of the transition system as shown below.

*Definition 3.5.* An *unrolling* of length $m$ of a transition system $\langle V, init, tr\rangle$ is a conjunction:

$$unrl(V,\ldots,V^{(m)}) \stackrel{\text{def}}{=} init(V) \wedge tr(V,V') \wedge \ldots \wedge tr(V^{(m-1)},V^{(m)})$$

over multiple copies of variables $V$, i.e., for all $i, j, k, n$, if $i \neq j$ and $k \neq n$, then $V^{(i)}[k] \neq V^{(j)}[n]$.

A model of an unrolling (if determined by an SMT solver) makes up the rows of a *data matrix*, which is used to infer relationships between the program variables.

*Definition 3.6.* For a given program, a given unrolling, and a given model $\mathcal{M} \models unrl(V,\ldots,V^{(m)})$, a *data matrix* is an $m \times n$ matrix denoted $DM$ which is composed of $m$ rows representing the transitions of a program and of $n$ columns representing the value of a variable $V[j]$. An element $DM[i,j]$ contains the value in the $i^{\text{th}}$ transition of the $j^{\text{th}}$ variable. That is, $DM[i,j] \stackrel{\text{def}}{=} \mathcal{M}(V^{(i)}[j])$.

*Example 3.7.* Recall the program in Fig. 2 (c). If the value of y is less than the value of m, then the first branch is taken until y sufficiently increments. Then x increments until the loop terminates. In another case, if y is greater than or equal to m, then only the second branch is visited. These two cases require two different data matrices shown in Fig. 3 to allow for the proper reasoning about how the variables are related.

There are many algorithms to infer formulas over $V$ from a data matrix. One such algorithm takes two rows $k$ and $l$ of a data matrix $DM$ and two variables $V[i]$ and $V[j]$ on a *canonical equation of a line* [15]. This pairing yields an equality of the form $(DM[l,j] - DM[k,j]) \cdot (V[i] - DM[k,i]) =$

$(DM[l,i]-DM[k,i])\cdot(V[j]-DM[k,j])$. Performing addition/subtraction of two equalities produced this way for any $V[i], V[j], k$, and $l$ leads to the discovery of new formulas relating multiple variables.

*Example 3.8.* Consider the data matrix in Fig. 3 for the second possible execution. Take the variables x, n, and tc. The equality $(1-0)\cdot(\mathtt{n}-2)=(2-2)\cdot(\mathtt{x}-1)$ which simplifies to $\mathtt{n}=2$ comes from relating the top two rows in the matrix and x with n. Another equality comes from relating tc and x, namely $(1-2)\cdot(\mathtt{x}-2)=(1-0)\cdot(\mathtt{tc}-0)$ which simplifies to $\mathtt{tc}=2-\mathtt{x}$. Taking the difference of these two equalities yields the equality, $\mathtt{tc}=\mathtt{n}-\mathtt{x}$.

Finally, we rely on the concept of logic *abduction* that allows us to infer preconditions. The goal of abduction is to find the weakest formula $\psi$ over variables $W$ that implies some $\phi$. This is done by universally quantifying the variables in $\phi$ except those in $W$. The resultant formula is a precondition that covers the states constrained by $\phi$ and is non-trivial. We write $\psi \leftarrow \textsc{abduce}(W,\phi)$ to learn the formula $\psi(W)$, such that:

$$\psi \leftarrow \textsc{abduce}(W,\phi) \stackrel{\text{def}}{=} \mathrm{QE}(\forall \mathit{Vars}(\phi) \setminus W \,.\, \psi \implies \phi)$$

## 4 ELBA Solver for Transition Systems

The problem of Exact Loop Bound Analysis (ELBA) is technically an instance of a precondition synthesis problem, defined in Sect. 3, and an instance of a second-order functional synthesis problem.

### 4.1 Problem Statement

We aim to find a suitable predicate that strengthens initial states for which the given safety property holds. In the context of loop bound analysis, a fresh variable is needed to compute the number of transitions, which we call an *transition counter* (tc), that decrements in each loop iteration. Intuitively, we wish to synthesize a function $f$ to precompute the value of tc over the initial values of program variables, such that tc holds zero after the loop has terminated.

The desired function $f$, when found, can be interpreted as a "transition budget" known a priori. The assertion indicates that this budget is *fully exhausted* at the end of the loop. Many termination analysis approaches, e.g., [11, 18, 47], operate by adding a fresh counter, and we extend its use in this work. Compared to the problem of precondition synthesis, instead of finding a precondition of arbitrary shape, our preconditions are restricted to an equality $\mathtt{tc}=f(V)$, where an integer function $f(V)$ places some restriction on tc in terms of the variables $V$. Note that since $\mathtt{tc}\notin V$, this constraint is not trivially vacuous. Formally, the instrumentation of $\langle V, \mathit{init}, \mathit{tr}\rangle$ with tc and with the equality in the desired precondition needs to meet the following properties for some safe inductive invariant *inv*:

*Definition 4.1 (ELBA).* Given a transition system $T=\langle V, \mathit{init}, \mathit{tr}\rangle$, by $T^f_{\mathtt{tc}}=\langle V\cup\{\mathtt{tc}\}, \mathit{init}\wedge \mathtt{tc}=f(V), \mathit{tr}\wedge \mathtt{tc}'=\mathtt{tc}-1\rangle$ we denote an instrumented transition system with a fresh integer variable $\mathtt{tc}\notin V$, and function $f\in\Lambda$. Given $T$ and a function $g\in\Lambda$, the ELBA problem is concerned with finding one particular $f$ such that $T^f_{\mathtt{tc}}$ is safe w.r.t. $\neg(\exists V' \,.\, \mathit{tr})(V)\wedge\neg(\mathtt{tc}=g(V))$.

Intuitively, in order to define a set of *error states*, as in Def. 3.3, we exploit a *loop guard* computed by eliminating the next-state variables from the transition relation in Def. 4.1 (i.e., $\neg(\exists V' \,.\, \mathit{tr})$). Function $g$ from the definition enables us to parametrize the problem with various *leftover* transition budget, and for single-loop programs, it makes sense to let $g \stackrel{\text{def}}{=} \lambda V \,.\, 0$. Thus, throughout this section, we let the *bad* predicate to be $(\neg\exists V' \,.\, \mathit{tr})(V)\wedge\neg(\mathtt{tc}=0)$. In other words, for some $T$ we are looking

for some $f$ and $inv$, such that the following three implications are valid:

$$init(V) \wedge \mathtt{tc} = f(V) \implies inv(V, \mathtt{tc}) \tag{5}$$

$$inv(V, \mathtt{tc}) \wedge tr(V, V') \wedge \mathtt{tc}' = \mathtt{tc} - 1 \implies inv(V', \mathtt{tc}') \tag{6}$$

$$inv(V, \mathtt{tc}) \wedge (\neg \exists V' \,.\, tr)(V) \wedge \neg(\mathtt{tc} = 0) \implies \bot \tag{7}$$

Strictly speaking, this is the problem of *functional synthesis* for a precondition in *second-order* logic:

$$\exists f \,.\, \exists inv \,.\, \forall V, V', \mathtt{tc}, \mathtt{tc}' \,.\ (5) \wedge (6) \wedge (7).$$

Although the user might be interested only in an interpretation of $f$ and not in the actual invariant; the latter is useful as a *proof that the former is correct*. In our approach, we synthesize both a bound and its proof.

### 4.2 Solver

Defined in the previous subsection, the ELBA problem can be approached by analyzing each of a program's possible initial conditions. In this section, we illustrate the main ingredients of our approach: the iterative synthesis of an exact bound, using program unrollings, and the synthesis of its precondition by *bounded abduction*. That is, for a given transition system $T$, we consider a subproblem to find a finite set of disjoint formulas $p_1, \ldots, p_n$, such that $p_1 \vee \ldots \vee p_n = \top$, for each $i \neq j$, $p_i \wedge p_j = \bot$, and each $p_i$ defines a restriction on the initial states *init* that allows us to analyze a set of *under-approximations*.

*Definition 4.2.* Given a transition system $T = \langle V, init, tr \rangle$ and a formula $p$ over free variables $V$, we say that a transition system $T_p = \langle V, init \wedge p, tr \rangle$ is an *under-approximation* of $T$.

Some under-approximations (incl. the trivial for $p \stackrel{\text{def}}{=} \bot$) may make the transition system contain no transitions. We say that $T = \langle V, init, tr \rangle$ is *empty* if $\forall V' \,.\, init(V) \implies \neg tr(V, V')$. A loop bound for an empty transition system is obviously zero since the loop never executes.

Lemma 4.3. *If $T$ is* empty *then $f \stackrel{\text{def}}{=} \lambda V \,.\, 0$ is a solution for the ELBA problem for $T^f_{\mathtt{tc}}$.*

Our ELBA solver is defined in Alg. 1. It works by enumerating under-approximations and finding bounds for each one separately. The key construction for data gathering is loop unrolling, recall Def. 3.5. Given a satisfiable unrolling, values for each variable in each loop iteration in an unrolling are taken from a model generated by an SMT solver. Thus, there is no need to actually run the program on different datasets, and the entire computation can be done at compile time. The models are then analyzed to synthesize a function over the program variables.

The ELBA solver begins by searching for a satisfiable unrolling[3] and then synthesizes a function $f$ such that the equality $\mathtt{tc} = f(V)$ holds for each iteration (modulo indexed variables for $V$) of that unrolling. Next, it assumes that the discovered $f$ is a possible bound for some under-approximation and tries to describe it (by finding its precondition), but as weak as possible. The algorithm poses an abduction query to find this precondition under which the function from data is evaluated to zero when computed with the variables' initial values. When a bound is synthesized, the associated precondition is blocked from the next search, so that redundant steps are avoided. Again, the bound search continues until the space covered by the initial states of the program are also covered by the preconditions. The core algorithm can be described in four steps:

[3]In practice, the algorithm could take considerable time to find a satisfiable unrolling when it depends on large concrete numbers (e.g., for a loop with a counter explicitly bounded by 1000). As an optimization, our implementation preprocesses the program by soundly abstracting large constants away. When a bound is synthesized, the constants are added back to the solution which is correct by construction.

**Algorithm 1:** ELBA Solver$(T, g)$

**Input:** $T = \langle V, init, tr\rangle$: program and $g$: leftover budget
**Output:** $ite(\psi_1, f_1, ite(\psi_2, f_2, ite(\ldots)))$ is a solution for ELBA-problem

```
1  let B be ∅;
2  while ⊤ do
3      let M be ∅; C be ∅, p be ⋀_{⟨ψ,_⟩∈B} ¬ψ; and φ be init ∧ p ∧ tr;
4      if ¬isSat(φ) then
5          B ← B ∪ {⟨p, 0⟩};
6          return ite(ψ1, f1, ite(ψ2, f2, ite(...))) where all ⟨ψi, fi⟩ ∈ B;
7      φ ← p;
8      for c ∈ [0, some threshold k] do
9          φ ← φ ∧ tr(V^(c), V^(c+1)) ∧ tc^(c+1) = tc^(c) − 1;
10         let φ′ be φ ∧ ¬(∃V′ . tr)(V^(c+1)) ∧ tc^(c+1) = g(V^(c+1)));
11         if isSat(φ′) then
12             M ← getModel(φ′);
13             C ← C ∪ {c};
14             if * then break;
15     if M = ∅ then F ← {λV . − 1};
16     else F ← invFromData(V ∪ {tc}, createDM(V ∪ {tc}, M, φ));
17     for f ∈ F do
18         ψ ← getPre(⟨V, init ∧ ⋀_{⟨ψ,_⟩∈B} ¬ψ, tr⟩, tc, f, g, C);
19         if ψ ≠ false then
20             B ← B ∪ {⟨ψ, f⟩};
21             break;
```

**step 1** [lines 3 - 6] Solve the formula $\phi \stackrel{\text{def}}{=} init \wedge \bigwedge_{\langle\psi,\_\rangle\in B} \neg\psi \wedge tr$: if unsat, then there are no transitions left. Then the transition system $\langle V, init \wedge \bigwedge_{\langle\psi,\_\rangle\in B} \neg\psi, tr\rangle$ is empty, and thus its bound is zero.

**step 2** [lines 8 - 14] Get an unrolling and its model of the instrumented transition system with `tc` that represents a viable execution of the transition system under the constraints from the previous step. Because there could be multiple suitable unrollings (each of which yields a distinct data matrix), our algorithm picks one nondeterministically (line 14).

**step 3** [lines 15-16] Get a data matrix from the model and find a bounded data invariant of form $\texttt{tc} = f(V)$ (see Def. 4.4 below). If there is no model, the algorithm *guesses* a negative bound.

**step 4** [lines 17 - 20] Find a suitable precondition under which the discovered function $f$ gives an exact bound. This is the most computationally heavy step where the actual synthesis is performed.

*Definition 4.4.* Given a data matrix $DM$ with $m$ rows and $n$ columns generated for program over variables $V$, a *data invariant* is a formula $\delta(V)$ such that for each $i \in [1, m]$, $\delta(DM[i, 1], \ldots, DM[i, n])$ evaluates to true (we say that it holds at each row of $DM$, for simplicity).

In step 3, the data invariant is synthesized by analyzing the bounded unrolling, guided by all previously derived preconditions to ensure exploration of novel executions of the program. As a convention, the transition counter (`tc`) values are stored in the $DM[i, n]$ entries of the data matrix

*DM*. We utilize the function InvFromData($V \cup \{\texttt{tc}\}, DM$) to derive a set of data invariants of the form $\texttt{tc} = f(V)$ from *DM* over the set of program variables $V$ and tc. As detailed in the penultimate paragraph of Section 3, and which we emphasize here, the technique used by InvFromData is based on examining variable pairs within the data matrix. By performing operations such as addition or subtraction on these pairs, we can derive new formulas that reveal relationships between variables. This specific approach effectively "connects" variables onto a *canonical equation of a line* [15], enabling the discovery of relationships that might otherwise be challenging to synthesize. Recall Example 3.8 in Sect. 3 and the found equality $\texttt{tc} = \texttt{n} - \texttt{x}$. During the enumeration, many other equalities are found, but it can be the case that some of these equalities do not capture the behavior of the loop correctly. For this reason, all of the equalities that are produced by this technique are checked against all of the rows in the data matrix from which they were derived. The equality $\texttt{tc} = \texttt{n} - \texttt{x}$ holds true for every row of the matrix, so it is kept to be checked as a bound. It is worth noting that while this "connecting" method is a focus of our technique for learning equalities from the data matrix, InvFromData could also utilize other algorithms, such as Gauss-Jordan elimination, to learn a relationship between tc and $V$. In practice, however, we find that Gauss-Jordan elimination was not as effective at finding sufficiently descriptive equalities for the purpose of finding an exact bound.

*Example 4.5.* Formula $\texttt{tc} = \texttt{n} - \texttt{x} + \texttt{m} - \texttt{y}$ is a data invariant for the matrix from Fig. 3 (left). Formula $\texttt{tc} = \texttt{n} - \texttt{x}$ is a data invariant for the matrix from Fig. 3 (right).

Next we give an example of the entire run of the algorithm that highlights the iterative nature of generating bounds and their preconditions.

*Example 4.6.* Recall the program in Fig. 1 (c). This example involves exploring two preconditions since either one of $y$ or $z$ can determine the loop's termination. For the first precondition, the algorithm finds a satisfiable transition from the initial state, i.e., a model of formula $y > 0 \wedge z > 0 \wedge y' = y - 1 \wedge z' = z - 1$. The algorithm nondeterministically picks an unrolling of length two of the instrumented transition system and finds its model, e.g., $y \mapsto 2, z \mapsto 3, y' \mapsto 1, z' \mapsto 2$ which is further used as the first row in the data matrix:

| y | z | tc |
|---|---|---|
| 2 | 3 | 2 |
| 1 | 2 | 1 |
| 0 | 1 | 0 |

The discovered data invariants $\texttt{tc} = \texttt{y}$ and $\texttt{tc} = \texttt{z} - 1$ populate the set $F$ with functions $\lambda y\,.\,y$ and $\lambda y\,.\,y - 1$, respectively. Alg. 1 picks $\lambda y\,.\,y$ (as a simpler one, syntactically) and proceeds to call Alg. 2 (getPre). For now, we leave out the details of getPre and take it at face value that it synthesizes the precondition $\psi \stackrel{\text{def}}{=} \lambda z, y\,.\,z \geq y \wedge y \geq 0$. This precondition is connected to the bound: $\texttt{if (z} \geq \texttt{y} \wedge \texttt{y} \geq \texttt{0) then y else} \ \Delta$. Here, $\Delta$ is a placeholder for the missing part of the exact bound yet to be discovered.

Alg. 1 proceeds through a second iteration. The found precondition is negated and conjoined with *tr*, thus giving us formula $\neg(z \geq y \wedge y \geq 0) \wedge y > 0 \wedge z > 0 \wedge y' = y - 1 \wedge z' = z - 1$. A new model is produced, $(y \mapsto 3, z \mapsto 2, y' \mapsto 2, z' \mapsto 1)$, and different behavior is observed in the data

matrix.

| y | z | tc |
|---|---|---|
| 3 | 2 | 2 |
| 2 | 1 | 1 |
| 1 | 0 | 0 |

From the data matrix, the function $f \stackrel{\text{def}}{=} z$ is found. GETPRE is visited again with the new function $f$ and it finds a precondition of $\psi \stackrel{\text{def}}{=} y < z \wedge z \geq 0$. This precondition and function are added to the bound: `if (z ≥ y ∧ y ≥ 0) then y else if (y < z ∧ z ≥ 0) then z else Δ`.

A third iteration of Alg. 1 begins, but this time the satisfiability check fails and the bound for zero itererations is added. The exact bound has now been found: `if (z ≥ y ∧ y ≥ 0) then y else if (y < z ∧ z ≥ 0) then z else 0`.

### 4.3 Precondition Inference Via Bounded Abduction and Generalization

The tricky part of the algorithm is in step 4 where a precondition needs to be synthesized and proven for potentially infinite subset of program executions. When this subset is finite, we can solve it using abduction (i.e., quantifier elimination), but otherwise, we cannot formulate it as an abduction query in first order logic. It is equivalent to solving a precondition inference problem: find $\psi$ such that the under-approximation $\langle V \cup \{\texttt{tc}\}, \mathit{init} \wedge \texttt{tc} = f(V) \wedge \psi, \mathit{tr} \wedge \texttt{tc}' = \texttt{tc} - 1\rangle$ is safe w.r.t. $\neg(\exists V' \,.\, \mathit{tr}) \wedge \neg(\texttt{tc} = 0)$. We present a new technique called *bounded abduction* to create preconditions for queries originating from (bounded) unrollings using abduction, and then generalizing them to preconditions for a possibly unbounded number of loop behaviors.

**step 4.1** [line 3-4] Prepare an unrolling (of some length $c$) and check its satisfiability.

**step 4.2** [line 5] Solve the *bounded abduction* query for $c$ steps taken from prevously successful unrollings and get some $\psi$.

**step 4.3** [line 6] Split the disjunctive result from abduction into each disjunct and store them in a list of sets.

**step 4.4** [line 8] Prepare a list of all possible combinations from the results from step 4.3 and store them in $D$.

**step 4.5** [lines 11] For each combination made in step 4.4, infer (Alg. 3) a set of constraints that together make up a possible precondition $\varphi$.

**step 4.6** [lines 13] Check safety of $\langle V \cup \{\texttt{tc}\}, \mathit{init} \wedge \texttt{tc} = f(V) \wedge \varphi, \mathit{tr} \wedge \texttt{tc}' = \texttt{tc} - 1\rangle$ w.r.t. $\neg(\exists V' \,.\, \mathit{tr}) \wedge \neg(\texttt{tc} = 0)$.

**step 4.7** [line 13] Try to weaken $\varphi$ ($\bigwedge_{p \in P} p$) from step 4.6 while maintaining the safety of the system. The result is the safe precondition returned to Alg. 1.

**step 4.8** [line 14-16] Check that the result $\beta$ is weaker than the previous result $\alpha$ (which is initially *false*) and update $\alpha$ accordingly.

We first discuss a simple version of this algorithm for the case when the results of abduction are disjunction-free. The discussion in Sect. 4.4 covers a more complicated case which allows us to extend the algorithm to programs with branching control flow.

*Example 4.7.* Again recall the program in Fig. 1 (c). In Ex. 4.6 we showed how satisfiable unrollings are used to create a data matrix and produce a function for the bound. We glossed over the steps taken in Alg. 2 to produce the precondition that covers the bound. Here we illustrate those steps in the context of the same program.

After the function $f \stackrel{\text{def}}{=} \lambda y \,.\, y$ is found from data, the algorithm continues to the abduction phase. For demonstration reasons, we assume set $C = \{1, 2\}$. Bounded abduction begins with $c = 1$, with

**Algorithm 2:** GETPRE$(T, \texttt{tc}, f, g)$

**Input:** $T = \langle V, init, tr\rangle$: program, tc: counter, $f$: initial budget, $g$: leftover budget, $C$: Set of satisfiable unrolling lengths

**Output:** solution of the precondition synthesis problem
$\langle V \cup \{\texttt{tc}\}, init \wedge \texttt{tc} = f(V), tr \wedge \texttt{tc}' = \texttt{tc} - 1\rangle, \neg\exists V' \,.\, tr \wedge \texttt{tc} \neq g(V)$

1 let $\phi$ be $init$; and $A$ be $nil$;
2 **for** $c \in C$ **do**
3 $\quad \phi \leftarrow init \wedge \bigwedge_{i\in[1,c]} tr(V^{(i)}, V^{(i+1)})$;
4 $\quad$ **if** $\neg$ISSAT$(\phi)$ **then continue**;
5 $\quad \psi \leftarrow$ ABDUCE$(V, \phi \wedge \neg(\exists V' \,.\, tr) \wedge f(V^{(c+1)}) = g(V^{(c+1)})$;
6 $\quad$ let $\psi$ be in DNF, i.e., $\psi = \psi_1 \vee \psi_2 \vee \ldots \vee \psi_n$;
7 $\quad A \leftarrow A :: \{\psi_i \mid i \in [1, n]\}$;
8 let $D$ be $\Pi_{i=1}^{N} A_i$ where $N$ is the length of $A$;
9 let $\alpha$ be $false$;
10 **for** $d \in D$ **do**
11 $\quad P \leftarrow$ INFERFROMPROJECTIONS$(d)$;
12 $\quad$ **if** $\neg$ISSAFE$(\langle V, init \wedge \bigwedge_{\gamma\in P} \gamma, tr\rangle, f(V) = g(V))$ **then continue**;
13 $\quad \beta \leftarrow$ WEAKENANDCHECK$(\langle V \cup \{\texttt{tc}\}, init \wedge \texttt{tc} = f(V), tr \wedge \texttt{tc}' = \texttt{tc} - 1\rangle, \neg\exists V' \,.\, tr \wedge \texttt{tc} \neq g(V), P)$;
14 $\quad$ **if** $\alpha \implies \beta$ **then** $\alpha \leftarrow \beta$;
15 $\quad$ **else if** $\beta \implies \alpha$ **then continue**;
16 $\quad$ **else** $\alpha \leftarrow \alpha \vee \beta$;
17 **return** $\alpha$;

the query constructed from an unrolling of the instrumented transition system conjoined with the termination condition. That is, we wish to find the weakest $\psi_1(y, z)$ such that:

$$\psi_1(y,z) \implies \exists y', z' \,.\, y > 0 \wedge z > 0 \wedge y' = y - 1 \wedge z' = z - 1 \wedge \neg(y' > 0 \wedge z' > 0) \wedge y' = 0$$

This yields $\psi_1 \stackrel{\text{def}}{=} \lambda z, y \,.\, z \geq 1 \wedge 1 = y$.

For $c = 2$, we wish to find the weakest $\psi_2(y, z)$ such that:

$$\begin{aligned}\psi_2(y,z) \implies \exists y', z', y'', z'' \,.\, & y > 0 \wedge z > 0 \wedge y' = y - 1 \wedge z' = z - 1 \wedge \\ & y' > 0 \wedge z' > 0 \wedge y'' = y' - 1 \wedge z'' = z' - 1 \wedge \neg(y'' > 0 \wedge z'' > 0) \wedge y'' = 0\end{aligned}$$

This yields $\psi_2 \stackrel{\text{def}}{=} \lambda z, y \,.\, z \geq 2 \wedge 2 = y$.

Our algorithm INFERFROMPROJECTIONS implements a simple heuristic to generalize from sequences (i.e., lists) of conjunctive formulas (projections $\psi_1$ and $\psi_2$), i.e.,

$$z \geq 1 \wedge 1 = y \text{ and}$$
$$z \geq 2 \wedge 2 = y$$

The order of elements in the list is crucial, and we know that the first (resp. second) element represents a precondition for one (resp. two) iteration(s) of the loop under a certain bound. First, it breaks the equality in $1 = y$ into $\{1 \leq y, 1 \geq y\}$ and adds them to $P$. Second, it generates new (in a sense, redundant) formulas by replacing 1 by $y$ in the formula $z \geq 1$ (line 5) and adds it to $P$. Then, it iteratively removes elements of $P$ if they are not implied by the second formula. Clearly, because

**Algorithm 3:** INFERFROMPROJECTIONS ($A$)

**Input:** $A$: list of $N$ results from bounded abduction (with increasing length of unrollings).
**Output:** $P$: A set of the weakest constraints from $A$ that make up a precondition.

```
1  Let P be ∅;
2  for each conjunct φ in A[0] do
3      if φ has form a = b then
4          P ← P ∪ {a ≤ b, a ≥ b};
5          A[0] ← A[0] ∧ A[0][a/b] ∧ A[0][b/a];
6      else
7          P ← P ∪ {φ};
8  for n ∈ [1, N) do
9      for ψ ∈ P do
10         if ¬(A[n] ⟹ ψ) then
11             P ← P \ {ψ};
12 return P;
```

$z \geq 2 \land 2 = y \implies z \geq y$ and $2 = y \implies 1 \leq y$, but $2 = y \not\Longrightarrow 1 \geq y$, INFERFROMPROJECTIONS returns $\{z \geq y, 1 \leq y\}$.

The next iteration of the main algorithm is largely similar in its operations for the abduction steps taken with the next bound $f \stackrel{\text{def}}{=} \lambda z \,.\, z$, which thus is omitted for brevity.

### 4.4 Support for Branching Control Flow

The algorithm offers sophisticated reasoning for cases of branching control flow, i.e., where the symbolic constraints require disjunctions. In the previous subsection, where we only considered cases with straightforward monolithic behaviors, we can use effective heuristics to generalize preconditions from bounded abduction. However, they expect the abduction results to be disjunction-free. If a loop has branching control flow (i.e., if-then-else statements) then the heuristics become ineffective. In this section, we explain how Alg. 2 overcomes this obstacle and goes over a more complicated process to synthesize a precondition for a given bound.

Recall that an abduction query is solved in line 5 of Alg. 2 on each of the unrollings, resulting in a formula that captures various constraints possible for a set of loop transitions. This formula is in general a disjunction where each disjunct describes a single branch taken in the loop. We assume that each disjunct has no nested disjunct and thus is referred to as a *projection*.

Projections are necessary to capture branching control flow in a loop. With the results from abduction collected, the algorithm moves to group the projections taken from each unrolling in all possible combinations. Grouping the projections in this way allows us to analyze the behavior of a particular symbolic execution of the loop. If instead we examined the full disjunction, the result would be a precondition that implies all of the loop conditions described by the disjunction. Instead we aim for a precondition that describes only particular sequences of the loop iterations and applies to the bound we are solving for.

We target such a precondition by splitting the disjunction produced by our abduction query. In this way our precondition is applicable to the bound that was found in Alg. 1. At the end of Alg. 2 we then perform a kind of weakening of the precondition through Alg. 4, and weaken further if necessary in lines 14-16. This is explained in more detail below.

Alg. 2 makes a call to Alg. 3 that processes the list $A$ containing logical formulas to produce a set of weakened preconditions $P$. Alg. 3 operates in two parts: First, it processes $A[0]$ by splitting

each equality constraint $a = b$ into two inequalities ($a \leq b$ and $a \geq b$) while preserving all other conjuncts. Second, it adds extra formulas to the consideration by exploiting equalities and making various substitutions. Then, it iteratively refines this set by removing any inequality that is not implied by subsequent elements $A[1]$ through $A[N-1]$. Upon termination, $P$ contains the set of weakened constraints that make up the precondition for $f(V)$.

*Example 4.8.* Consider the program in Fig. 2 (c). Assume a bound $f \stackrel{\text{def}}{=} n - x$ is an input to Alg. 2. The abduction query for an unrolling of length one of the transition system is as follows:

$$\psi_1(x, y, m, n) \implies \exists x', y', m', n' \,.\, x = 0 \wedge y = 0 \wedge x < n \wedge m = m' \wedge n = n' \wedge \\ y' = ite(y < m, y + 1, y) \wedge x' = ite(y < m, x, x + 1) \wedge x' \geq n' \wedge n' - x' = 0$$

The abduction yields the following formula:

$$n > 0 \wedge x = 0 \wedge y = 0 \wedge n > x \wedge n - x = 1 \wedge y \geq m$$

This result describes the case that when the loop can only perform one iteration, the branch that increments $y$ can never be reached, hence the part of the precondition $y \geq m$. It also correctly gives the initial values of variables $x$ and $y$.

The unrolling of length two is processed similarly, with one more transition through the loop:

$$\begin{aligned}\psi_2(x, y, m, n) \implies \exists x', y', m', n', x'', y'', m'', n'' \,.\, & x = 0 \wedge y = 0 \wedge x < n \wedge \\ & y' = ite(y < m, y + 1, y) \wedge x' = ite(y < m, x, x + 1) \wedge \\ & m = m' \wedge n = n' \wedge n' - x' = 0 \\ & y'' = ite(y' < m', y' + 1, y') \wedge x'' = ite(y' < m', x', x' + 1) \wedge \\ & m' = m'' \wedge n' = n'' \wedge x'' \geq n'' \wedge n'' - x'' = 0\end{aligned}$$

Abduction produces a disjunction now:

$$\psi_2 \stackrel{\text{def}}{=} x = 0 \wedge y = 0 \wedge ((y \geq m \wedge n = 2) \vee (m = 1 \wedge n = 1))$$

Here abduction has captured the two outcomes possible for an execution of the loop for two iterations. Either $y \geq m$ and the loop iterates twice with $n$ having a value of 2, or the loop iterates once with $y < m$ and then once with $n$ having the value 1. In the first case, $x$ increments on each iteration, and the loop terminates when $x = 2$. In the second case, $x$ has a final value of 1.

The disjunctive nature of the result of the second abduction query prevents us from effectively generalizing it with the result of the first abduction query. Our solution is to first: DNF-ize the results of abduction, create a list of combinations, and apply algorithm inferFromProjections for each (disjunction-free) combination. That is, first inferFromProjections gets

$$x = 0 \wedge y = 0 \wedge y \geq m \wedge n = 1 \text{ and}$$
$$x = 0 \wedge y = 0 \wedge y \geq m \wedge n = 2$$

and it returns $x = 0 \wedge y = 0 \wedge y \geq m \wedge n > 0$ which passes the safety check in Alg. 2 and furthermore can be weakened to $y \geq m \wedge n > 0$ because $x = 0 \wedge y = 0$ is part of the constraint describing the initial states. Second, it gets

$$x = 0 \wedge y = 0 \wedge y \geq m \wedge n = 1 \text{ and}$$
$$x = 0 \wedge y = 0 \wedge m = 1 \wedge n = 1$$

and it returns $x = 0 \wedge y = 0 \wedge n = 1$. However, this precondition is too restrictive, and the safety check in Alg. 2 returns false. Thus, only the first generalized abduction result is used as the output of getPre in Alg. 1.

**Algorithm 4:** weakenAndCheck$(T, \varphi, \Theta)$

**Input:** $T = \langle V, init, tr\rangle$: program, $\varphi$: post-condition, $P$: a set of expressions forming a precondition on $T$.
**Output:** $\beta$: a weakened precondition for the bound tc.

1 let $G$ be $P$ and $W$ be $\emptyset$;
2 **for** $\gamma \in G$ **do**
3 $\quad G \leftarrow G \setminus \{\gamma\}$;
4 $\quad$ **if** $\neg$isSafe$(\langle V, init \wedge \bigwedge_{\rho\in G} \rho \wedge \bigwedge_{\omega\in W} \omega, tr\rangle, \varphi)$ **then** $W \leftarrow W \cup \{\gamma\}$;
5 **return** $\bigwedge_{\omega\in W} \omega$;

### 4.5 Strategic Weakening While Maintaining Safety

Since $\varphi$ is based on a finite unrolling, it can be the case it is too strong and it overlooks states that are valid under the current bound $f(V)$. Alg. 4 attempts to find a weaker precondition that is still valid for $f(V)$.

Theorem 4.9. *Alg. 4 returns a (possibly weakened) precondition that maintains the safety of the system $T$.*

Theorem 4.9 is proven by looking at two cases. The first is when no weakening is done. Alg. 4 is only visited when the system is safe under the precondition $\bigwedge_{\gamma\in P} \gamma$, and it is clear that the system maintains safety if no changes to the precondition are made. In the next case, the algorithm only drops an entry $\gamma$ from $P$ when the new precondition $\bigwedge_{g\in G\setminus\{\gamma\}} g$ also passes a safety check. If, after dropping $\gamma$ the system becomes unsafe, $\gamma$ is added back to the precondition for all subsequent checks. Thus, in line 5, all expressions that led to non-safety when removed (namely those added to set $W$) from $P$ are returned as the new precondition. In other words, Alg. 4 only drops expressions from $P$ when they do not affect the safety of the system under the bound $\varphi$.

In line 4, Alg. 4 makes a check for the safety of the augmented transition system $T^f_{\mathtt{tc}}$. It is at this point that it is most obvious where our bound analysis via precondition synthesis is reduced to a task of safety verification. In doing so we adopt the guarantees provided when a safe inductive invariant is found. If this check returns safe then we know that the precondition and the bound are correct. Since it is possible for more than one set of projections to produce a good precondition for a bound, Alg.1 (lines 14-16) decides whether a newly found bound is weaker than one previously found. Further, Alg. 1 runs until the conjunction of negated preconditions is no longer satisfiable. When this occurs, it signals that the preconditions cover all possible executions of the program.

Theorem 4.10. *When Alg. 1 terminates, it returns a solution to the ELBA problem.*

We offer an informal proof of Thm. 4.10. For each precondition we find, and the bound associated with it, Alg. 2 performs a safety check for the corresponding underapproximation in line 13. The safety property ensures that after the loop tc must be exactly zero. When this check is passed, it tells us that transition counter tc always evaluates to zero at the end of the loop under the precondition $\psi$, confirming thus that the bound is exact. Finally, the algorithm terminates only when all the preconditions (and thus, underapproximations) have been exhaustively explored and cover all possible initial states.

### 4.6 Invariant Discovery

In this subsection, we elaborate on a subroutine of Alg. 2 to find safety invariants for each discovered precondition. The inability to find them (usually, after a timeout) triggers the algorithm to withdraw the precondition. In a nutshell, our invariant synthesizer is a *guess-and-check* loop that takes a set of candidate interpretations for *inv* as input, then substitutes the conjunction of them for *inv* find Def. 3.3, and removes the candidates that break the validity of at least one implication. We use a staged approach for enumerating candidates.

**Stage 1**. We begin with checking candidate invariants created as a by-product of our data learning. In Alg. 1, the method invFromData may generate equalities that do not involve `tc`. These equalities are not used as part of the bounds, they are rather held over for use in this stage of invariant synthesis.

**Stage 2**. We next check the candidates from literals of $\psi_i$ and formulas forward-propagated from $\psi_i$ to the next step. Specifically, let $\psi_i = \bigwedge_i c_i$. Then, for each $c_i$, we compute a predicate $c_i'$ using quantifier elimination (QE), i.e.,

$$c_i'(V') \stackrel{\text{def}}{=} \text{QE}(\exists V \,.\, c_i(V) \wedge \psi_i(V) \wedge tr(V,V'))$$

A set of variables $V \subseteq V \cup V'$ only excludes the primed variables occurring in $c_i$.

**Stage 3**. We synthesize predicates that are implied by $\psi_{i+1}$ by *mutating* (or *merging*) literals from $\psi_{i+1}$. In particular, assume two literals $c_i$ and $c_j$ from $\psi_{i+1}$ have linear combinations $\ell_i$ and $\ell_j$ as subterms, respectively. Then we introduce a fresh integer variable $v$ and replace $\ell_i$ and $\ell_j$ by $\ell_i + v$ and $\ell_j + v$, respectively in $c_i$ and $c_j$. Eliminating $v$ from the conjunction of the resulting literals yields a new invariant candidate. More formally,

$$c_{ij} \stackrel{\text{def}}{=} \text{QE}(\exists v \,.\, c_i[\ell_i + v/\ell_i] \wedge c_j[\ell_j + v/\ell_j]).$$

The formula under QE is an overapproximation of the original formula $c_i \wedge c_j$: indeed, if $v$ is instantiated by zero, we get exactly $c_i \wedge c_j$.

**Stage 4**. Lastly, we backward-propagate the whole $\psi_{i+1}$ similarly to what we did in stage 2, but using abduction:

$$c_i'(V) \stackrel{\text{def}}{=} \text{QE}(\forall V \,.\, \psi_i(V) \wedge tr(V,V') \implies c_i(V')).$$

The main intention is to first try candidates that are easier to discover, so if there exists an invariant that only needs ingredients from step 1, then the process converges faster. However, sometimes they might require helper lemmas that should be discovered during another stage. So in principle, given enough power resources, all the stages can be combined and executed at once.

Lastly, we need to mention that this staged invariant synthesis process is by no means complete. We only allow a finite (and relatively small) number of candidates, which in principle can be extended by guessing from grammars, interpolation, or data analysis. Our intention to keep it short is motivated by the fact that the main algorithm invokes the invariant synthesizer many times, and we often prefer to terminate it with the unknown result and continue with another candidate, rather than to wait until it finds an invariant that appears to be useless for our bound analysis.

In this section, we have outlined several techniques, some of which rely on the synthesis of expressions to identify preconditions or inductive invariants. It is worth noting that the synthesis process involved in these techniques presents a computationally intensive challenge, often characterized by exponential complexity. As a result, there may be scenarios where ELBA is unable to compute a solution within practical time limits, even though a solution could exist.

Table 1. Exact bound results: ELBA outperforms Loopus by over 2X in finding exact bounds. Loopus and KoAT analyze quickly, often under 1s.

| Tool | Exact bound |
|---|---|
| ELBA | 62 |
| Loopus | 28 |
| KoAT | 0 |
| FreqTerm | 10 |

Table 2. Total successful analyses: ELBA found exact bounds for 62 benchmarks, Loopus 59 upper bounds, KoAT 64 upper bounds, FreqTerm 60 ranking functions.

| Tool | Total Successes | Avg time (s) |
|---|---|---|
| ELBA | 62 | 16.8 |
| Loopus | 59 | $\epsilon$ |
| KoAT | 64 | $\epsilon$ |
| FreqTerm | 60 | 41.0 |

## 5 Implementation and Evaluation

We have implemented our approach in a tool called ELBA[4]. It has been developed within the FreqHorn [16] framework, and we make use of several features for managing the instrumentation of a program for our bound analysis. For our SMT queries ELBA uses the Z3 solver [13]. We have implemented our own data analysis to infer relationships between program variables, and we reuse the functionality of FreqHorn to create unrollings and discover invariants incrementally. The data learner keeps unrollings small in the interest of compute time. Larger unrollings could be generated in principle, but there is no benefit to the results since we are inferring relationships in targeted places in the loop. To verify our synthesized invariants, we use the Houdini method [19], which aggressively prunes expressions that do not pass initiation (1) and are not inductive (2).

*Benchmarks.* Our benchmark set is taken from the *Termination Problem Data Base*[5], and includes a subset of the benchmarks from Loopus, FreqTerm, and FreqHorn. Several multi-phase benchmarks come from the ImplCheck repository [40]. The benchmarks are in linear integer arithmetic and have a loop with deterministic guards. We have also excluded non-determinism within the loop, such as non-deterministic if-statements (because in these cases, exact bounds rarely exist).

We have examined ELBA on 75 benchmarks that cover many different types of loops. 49 benchmarks are single-phase loops, with the remaining 26 being multi-phase loops. This is a useful separation to observe since multi-phase loops usually require more analysis to capture all of the branches in which the loop can execute. It is this distinction that separates our algorithm from others since other approaches cannot find bounds for individual phases or "configurations" of input variable values. Further, other tools completely fail on many of the multi-phase benchmarks, with Loopus able to solve 16 of the 26 multi-phase benchmarks. KoAT solves 18 out of 26, and FreqTerm solves 19 of 26. Another characteristic in the benchmark set is programs that have an initial assignment to their variables and those that do not. When a program has a concrete assignment to some or all of its variables it is likely that the bound can boil down to a concrete value. 35 benchmarks have such a concrete assignment, with the remaining 40 having a non-deterministic initial assignment to its input variables. We handle them with our preprocessor that soundly avoids creating and solving long unrollings due to large constants (recall the footnote in Sect. 4.2).

Loopus and KoAT strive to find a loop upper bound. Loopus uses abstract interpretation for its analysis, and KoAT uses a runtime analysis to estimate bounds. For Loopus occasionally the upper bound it finds is in fact the exact bound (though it never checks for exactness), however this is only the case in simple examples. KoAT reports its results as loose upper bounds. FreqTerm performs termination analysis by iteratively guessing and checking candidates for a ranking function that can capture an exact bound in some cases, but again only on simple ones.

[4]The source code is available at https://github.com/freqhorn/elba.
[5]https://github.com/TermCOMP/TPDB/tree/master/C_Integer.

*Experiments.* ELBA computed exact bounds for 62 of the 75 single-loop benchmarks. There are 38 examples where only ELBA, out of the four tools in the comparison, finds an exact bound. These are the cases where another tool fails to find an upper bound. When ELBA could not find an exact bound, it likely fails to synthesize a precondition in the abduction step and/or fails to synthesize a relationship between the instrumented counter and the program's variables by the data learner. ELBA finds bounds for 3 benchmarks that the three tools in comparison do not succeed.

Loopus found approximate bounds for 59 benchmarks. KoAT found approximate bounds for 64 benchmarks, and FreqTerm found a ranking function for 60 benchmarks. Out of the comparison tools Loopus finds the most exact bounds, 28. FreqTerm's ranking functions represent exact bounds in 8 benchmarks. KoAT does not find an exact bound for any of the benchmarks. Table 1 summarizes the number of exact bounds found by each tool, while Table 2 summarizes the total number of benchmarks for which each tool came to a successful result.

Our practical contribution is a fast tool that leverages state-of-the-art symbolic reasoning techniques. We have implemented several new techniques to solve a *first-of-its-kind* exact bound problem. These new algorithms show an ability to find exact bounds for many types of loops, even finding an exact bound for programs that other bound analysis tools fail. This novel approach shows promise in its use for applications that require a more precise bounds analysis.

## 6 Related Work

*Resource Analysis.* There are many techniques for analyzing loop bounds [6, 7, 18, 20, 33, 36, 49] that mainly use abstraction and/or invariants to characterize the behavior of a program. If a loop has multiple phases, it can be difficult to synthesize invariants that describe those phases and the termination conditions associated with them. Specifically, [36] discovers a large nonlinear numerical invariant and extracts particular cases from it. The approaches of [33, 47] use static analysis and abstract interpretation to gather information about costs at various points in a program. Cost analysis is also done by quantitative abstract execution [3] which focuses attention on the loop's counter and abstracts the parts of a loop's body to perform its analysis. Loopus [47] exploits the difference constraint abstraction, relational inequalities that describe some upper bound on a next state variable, and oscillates between variable bound analysis (an upper bound on the values a variable can take) and transition bound analysis (an upper bound on the program transitions).

Lower bound analysis [2] on the *worst case cost* attempts to find a tight upper bound on a terminating loop. With the use of *quasi invariants* (inductive invariants that don't hold in the initial states) and *narrowing guards* this approach specializes a loop such that the analysis works through executions that are considered worst case executions. A tight upper bound, or in the case of [2], a lower bound on a worst-case execution, are not exact. If a lower bound and an upper bound are found to be the same, many claim this is exact, however we note that this is only the case for specific conditions on an execution of the program. An exact bound must consider all possible executions of the program. Our contribution makes an advancement in the bound analysis realm by reporting the exact number of iterations a loop may perform based on the program input.

For equivalence checking, a technique for exact bound analysis could be useful to construct the alignment of unbalanced loops. The work from Alien [24] attempts to find an exact number of iterations that a source and target loops will perform. If exact bounds can be found, then the technique attempts to align the loops. To perform alignment, the loops are reorganized by moving certain iterations to either before or after the loop. Their technique for finding exact bounds can find them for only range-based loops with a counter, and we believe ELBA could improve their approach by supporting more complicated loops.

*Termination Analysis.* Synthesizing an upper bound for a loop counter can be done with some good guesses about the behavior of the loop [10, 18]. These techniques aim to synthesize a sufficient upper bound on the loop by using abstraction or Syntax Guided Synthesis (SyGuS) techniques to prove that a loop will eventually terminate. However, these techniques do not capture an exact bound on the loop. Another technique is to analyze the greatest lower bound and the least upper bound of a loop [31]. The produced result assigns a value to the counter variable, and captures only the information about a specific configuration rather than a generalized figure on the input variables. Our method aims to synthesize a specification for an unknown precondition on a counter, and we expect that specification to capture the exact number of iterations a loop will perform.

*Data Driven Learning.* Data is being increasingly exploited in various automated reasoning approaches [17, 21, 34, 36–38, 45, 49]. Usually this data is derived from a simple unrolling to gather information about the early few iterations of a loop, and is used to learn simple facts about the program. The use of data to drive invariant synthesis has been studied as a way to analyze program behavior for clues as to the shape of invariants [16, 29, 34, 39, 45, 48]. Data-driven techniques often start by analyzing the feasible paths through a program, producing an unrolling up to some bound [16, 45], or by analyzing counterexamples to guide the invariant inference [21, 36]. In [37], data is gathered from program executions and linear regression is performed to guess an upper bound. Their algorithm attempts to prove that the upper bound is valid by performing linear regression on a modified set of data to obtain an inductive invariant. Limitations of this approach are in its ability to handle partitioning of program inputs, and of loops with multiple phases. Recent data-driven invariant discovery work extends the unrolling idea to *fast-forward* an unrolling to a particular point in a loop's execution [40].

*Specification/Invariant Synthesis.* Techniques to discover a specification and invariants share a close relationship [14, 42]. Synthesis of preconditions [12, 39, 43] (and ours) is a natural subproblem of specification synthesis that focuses on the initial stage of a program. More generally, this analysis can be done for arbitrary points of a program [1, 42]. Invariant synthesis is a widely studied problem [4, 5, 9, 14, 25, 27, 28, 30, 35], which is ultimately undecidable, but research on subsets of the problem have made significant progress, specifically using SMT techniques. However, there still remain large gaps in these subsets, requiring more work to be done on the subject. One such example is loops with multiple phases. These cases are particularly challenging since they often require disjunctive invariants [40, 44]. In the future, these techniques can strengthen our ELBA solver by making its strategic weakening more effective.

## 7 Conclusion

We have described a first-of-its-kind technique to compute exact bounds for a loop as a function over the input variables, i.e., before the loop begins. Our approach automatically discovers such a function by taking into account all possible executions of the program. For each class of executions that shares the same bound, our approach synthesizes a precondition using a novel abduction-based technique. The synthesized bounds are correct by construction, thanks to the discovery of safe inductive invariants that is performed on demand. The implementation in a tool called ELBA has been shown to outperform state-of-the-art bound analysis tools. Our future work will explore how our techniques can be adapted to discover under-approximations of (mainly nondeterministic) programs with exact bounds when no exact bound exists for the whole program.

*Artifact Available.* A virtual machine is available to reproduce the reported results [41].

*Acknowledgements.* This work was partially supported by the National Science Foundation grant 2106949.

## References


[1] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL*. ACM, 789–801.

[2] Elvira Albert, Samir Genaim, Enrique Martin-Martin, Alicia Merayo, and Albert Rubio. 2021. Lower-Bound Synthesis Using Loop Specialization and Max-SMT. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 863–886. https://doi.org/10.1007/978-3-030-81688-9_40

[3] Elvira Albert, Reiner Hähnle, Alicia Merayo, and Dominic Steinhöfel. 2021. Certified Abstract Cost Analysis. In *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12649)*, Esther Guerra and Mariëlle Stoelinga (Eds.). Springer, 24–45. https://doi.org/10.1007/978-3-030-71500-7_2

[4] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. 2014. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *CAV (LNCS, Vol. 8559)*. Springer, 831–848.

[5] Aaron R. Bradley. 2012. Understanding IC3. In *SAT (LNCS, Vol. 7317)*. Springer, 1–14.

[6] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 140–155. https://doi.org/10.1007/978-3-642-54862-8_10

[7] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 140–155.

[8] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *PLDI*. ACM, 1027–1040.

[9] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. 2017. Invariant Checking of NRA Transition Systems via Incremental Reduction to LRA with EUF. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*, Axel Legay and Tiziana Margaria (Eds.). 58–75. https://doi.org/10.1007/978-3-662-54577-5_4

[10] Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. 2011. SAT-based termination analysis using monotonicity constraints over the integers. *TPLP* 11, 4-5 (2011), 503–520.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *PLDI*. ACM, 415–426.

[12] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *VMCAI (LNCS, Vol. 7737)*. Springer, 128–148.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.

[14] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA*. ACM, 443–456.

[15] Grigory Fedyukovich and Aarti Gupta. 2019. Functional Synthesis with Examples. In *CP (LNCS, Vol. 11802)*. Springer, 547–564.

[16] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2018. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*. IEEE, 170–178.

[17] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2019. Quantified Invariants via Syntax-Guided Synthesis. In *CAV, Part I (LNCS, Vol. 11561)*. Springer, 259–277.

[18] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-Guided Termination Analysis. In *CAV, Part I (LNCS, Vol. 10981)*. Springer, 124–143.

[19] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini: an Annotation Assistant for ESC/Java. In *FME (LNCS, Vol. 2021)*. Springer, 500–517.

[20] Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8858)*, Jacques Garrigue (Ed.). Springer, 275–295. https://doi.org/10.1007/978-3-319-12736-1_15

[21] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL*. ACM, 499–512.

[22] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation,*

*PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 375–385. https://doi.org/10.1145/1542476.1542518

[23] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *PLDI*. ACM, 281–292.

[24] Ameer Hamza and Grigory Fedyukovich. 2023. Lockstep Composition for Unbalanced Loops. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 270–288. https://doi.org/10.1007/978-3-031-30820-8_18

[25] Dejan Jovanovic and Bruno Dutertre. 2016. Property-directed k-induction. In *FMCAD*. IEEE, 85–92.

[26] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, Deborah T. Marr and David H. Albonesi (Eds.). ACM, 158–169. https://doi.org/10.1145/2749469.2750392

[27] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *PLDI*. ACM, 248–262.

[28] Hari Govind Vediramana Krishnan, Grigory Fedyukovich, and Arie Gurfinkel. 2020. Word Level Property Directed Reachability. In *ICCAD*. IEEE, 1–9.

[29] Abolfazl Lavaei, Ameneh Nejati, Pushpak Jagtap, and Majid Zamani. 2021. Formal safety verification of unknown continuous-time systems: a data-driven approach. In *HSCC '21: 24th ACM International Conference on Hybrid Systems: Computation and Control, Nashville, Tennessee, May 19-21, 2021*, Sergiy Bogomolov and Raphaël M. Jungers (Eds.). ACM, 29:1–29:2. https://doi.org/10.1145/3447928.3456661

[30] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *PLDI*. ACM, 788–801.

[31] Tianhai Liu, Shmuel S. Tyszberowicz, Bernhard Beckert, and Mana Taghdiri. 2017. Computing Exact Loop Bounds for Bounded Program Verification. In *Dependable Software Engineering. Theories, Tools, and Applications - Third International Symposium, SETTA 2017, Changsha, China, October 23-25, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10606)*, Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang (Eds.). Springer, 147–163. https://doi.org/10.1007/978-3-319-69483-2_9

[32] Pedro López-García, Luthfi Darmawan, Maximiliano Klemen, Umer Liqat, Francisco Bueno, and Manuel V. Hermenegildo. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *TPLP* 18, 2 (2018), 167–223.

[33] Pedro López-García, Maximiliano Klemen, Umer Liqat, and Manuel V. Hermenegildo. 2016. A general framework for static profiling of parametric resource usage. *Theory Pract. Log. Program.* 16, 5-6 (2016), 849–865. https://doi.org/10.1017/S1471068416000442

[34] Hong Lu, Jiacheng Gui, Chengyi Wang, and Hao Huang. 2020. A Novel Data-Driven Approach for Generating Verified Loop Invariants. In *International Symposium on Theoretical Aspects of Software Engineering, TASE 2020, Hangzhou, China, December 11-13, 2020*, Toshiaki Aoki and Qin Li (Eds.). IEEE, 9–16. https://doi.org/10.1109/TASE49443.2020.00011

[35] Kenneth L. McMillan. 2014. Lazy Annotation Revisited. In *CAV (LNCS, Vol. 8559)*. Springer, 243–259.

[36] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided approach to finding numerical invariants. In *ESEC/FSE*. ACM, 605–615.

[37] Aditya V. Nori and Rahul Sharma. 2013. Termination proofs from tests. In *ESEC/FSE*. ACM, 246–256.

[38] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

[39] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *PLDI*. 42–56. https://doi.org/10.1145/2908080.2908099

[40] Daniel Riley and Grigory Fedyukovich. 2022. Multi-phase invariant synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 607–619. https://doi.org/10.1145/3540250.3549166

[41] Daniel Riley and Grigory Fedyukovich. 2025. Artifact for Exact Loop Bound Analysis. https://doi.org/10.5281/zenodo.15199489

[42] Sumanth Prabhu S, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. 2021. Specification synthesis with constrained Horn clauses. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1203–1217. https://doi.org/10.1145/3453483.3454104

[43] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. 2008. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. 295–306.
[44] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV (LNCS, Vol. 6806)*. Springer, 703–719.
[45] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP (LNCS, Vol. 7792)*. Springer, 574–592.
[46] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. In *OOPSLA*. ACM, 391–406.
[47] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *Journal of Automated Reasoning* (2017), 1–43. https://doi.org/10.1007/s10817-016-9402-4
[48] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *PLDI*. ACM, 707–721.
[49] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 280–297. https://doi.org/10.1007/978-3-642-23702-7_22

Received 2024-11-15; accepted 2025-03-06