Imperative Program Synthesis by Abstract Static Analysis and SMT Mutations

Aleksandar S. Dimovski
Mother Teresa University
Skopje, Macedonia
aleksandar.dimovski@unt.edu.mk

Abstract

This paper introduces a novel technique for synthesizing imperative programs that meet behavioral specifications given in the form of assumptions and assertions (logic formulas). In particular, we combine basic statement-directed enumerative search, static analysis via abstract interpretation, and expression-directed enumerative search via (incremental) SMT-based mutations to efficiently explore all candidate complete programs generated from an input program template (with statement and expression holes) until a solution is found. Firstly, the algorithm uses a basic enumerative search through the space of all possible statements, thus filling in all statement holes. In effect, we obtain partial programs with only missing (arithmetic and boolean) expressions, which are subsequently classified by a static analysis either as potential solutions or as definite failures. Finally, we repeatedly mutate the missing expressions in potential solutions and check if the resulting complete programs become bounded correct with respect to the given assertions.

We have implemented our technique in a prototype tool and evaluated it on a set of introductory C programs. The experimental results confirm the effectiveness of our technique for synthesizing various interesting C programs.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; *Software creation and management*; • **Theory of computation** → Semantics and reasoning.

Keywords: Imperative Program Synthesis, Abstract Static Analysis, Bounded Model Checking, SMT-based Mutations.

GPCE '25, Bergen, Norway

ACM ISBN 979-8-4007-1995-0/25/07
https://doi.org/10.1145/3742876.3742884

ACM Reference Format:
Aleksandar S. Dimovski. 2025. Imperative Program Synthesis by Abstract Static Analysis and SMT Mutations. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25), July 3–4, 2025, Bergen, Norway*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3742876.3742884

1 Introduction

Imperative program synthesis is a task of inferring a program within a given syntactic search space that matches a given behavioural specification [1, 25, 38, 39]. A *program template* provides the form of the desired program we want to automatically infer. It simplifies the synthesis task by limiting the search to programs of particular form. It only specifies a high-level structure for an unknown program with constraints on its control-flow, its available variables and constants. That is, a template is a partial program that contains unknown holes, which can represent arbitrary statements and (arithmetic or boolean) expressions. The inputs of our algorithm are a program template with statement and expression holes that need to be completed, sets of available variables and constants the synthesizer can use, and a formal specification stated in the form of assertions (logic formulas). The output is a complete program that satisfies the assertions.

Imperative program synthesis has been previously examined in the program synthesis community [25, 38]. However, these works mainly use input/output examples as the specification, so the correctness of a candidate program is established by checking all examples against it. The main problem of all example-based approaches is the generation of *overfitting* solutions that pass all given examples, but they break some unchecked required functionality of the desired programs. As a specification is the foundation of program synthesis, it is very important to define specifications as precisely as possible. In contrast to example-based approaches, our work uses logic formulas as formal specifications to guide the synthesis process. This way, it gives stronger guarantees on the correctness of inferred programs in a similar way that formal verification gives higher level of confidence in programs than testing. This makes the given imperative program synthesis problem more general, but also more challenging to solve.

In this paper, we propose a novel technique for imperative program synthesis from logical specifications and templates.

More specifically, our synthesis technique consists of three phases: *statement-directed enumerative search*, *static analysis* by abstract interpretation, and *expression-directed enumerative search* by (incremental) SMT-based mutations.

In the first phase, the algorithm enumerates every possible candidate program in increasing size by filling in the statement holes. That is, all statements including assignments, sequential composition, `if` and `while` statements, as well as their order are part of the structure that needs to be specified in this phase. As a result, we obtain partial programs with only missing expression holes.

Since enumerating all possible complete programs is too large to search entirely, we prune the search space using static analysis to detect partial programs that certainly fail to be solutions. In particular, the obtained partial programs with expression holes are statically analyzed in the second phase using an numerical abstract domain (e.g. Polyhedra) [4, 5, 30], such that all expression holes in them are treated as non-deterministic choices, $[-\infty, +\infty]$ for arithmetic and $\{\text{true}, \text{false}\}$ for boolean expressions. If the over-approximated static analysis gives definite invalid answers, meaning that the assertions are indeed invalid regardless of approximations, then the corresponding candidate partial programs are pruned out from the search. That is, all complete programs derived from them will certainly fail to be solutions. On the other hand, if the over-approximated static analysis gives definite valid answers then any derived complete program will represent a correct solution to the given problem. Finally, if no definite conclusion is made, then it is possible to derive a solution from the given partial program.

In the third phase, we perform an expression-directed enumerative search by substituting complete expressions for the expression holes in the partial program found in the second phase. Once the initial candidate (complete) program is generated, we attempt to repair it by applying syntactic mutations [13, 35]. In particular, the CBMC bounded model checker [3] is used to translate the candidate program and the negated assertion into a set of SMT formulas, whose conjunction represents the *program formula*. The formula is *satisfiable* iff the program violates the assertion, in which case a satisfying assignment is also returned, whereas the program formula is *unsatisfiable* iff the assertion is valid in the given program. Since each statement corresponds to one formula in the obtained set of SMT formulas, the mutation of an expression from a statement corresponds to changing the respective SMT formula. Therefore, the search in the space of mutants of the initial candidate (complete) program reduces to searching for an unsatisfiable set of SMT formulas. This search is performed using a SAT solver that enables to consider minimally mutated programs and a SMT solver that enables to verify the (bounded) correctness of mutated programs. Both SAT and SMT solvers are used in an *incremental way*, which means that we avoid recomputations by using the fact that the mutated programs are similar. If no correct complete program is found, we generate a new candidate (complete) program by substituting complete expressions with bigger size for the expression holes and the entire mutation procedure is then repeated.

We have implemented our approach in a prototype program synthesizer, called IMPSynth. The tool uses the numerical abstract domains (e.g., Polyhedra) from the APRON library [23] for static analysis; the (altered) CBMC bounded model checker [3] for translating programs to SMT formulas; and the (altered) AllRepair tool [35] for incremental SMT-based mutations. IMPSynth also calls the MiniCard SAT [26] and Z3 SMT [7] solvers. We evaluate this approach for synthesizing various introductory C programs taken from the literature [25, 38] and online forums.

This work makes the following contributions:

(1) We explore the idea of imperative program synthesis from logic formulas and program templates;
(2) We show how this problem can be solved using combination of statement-directed enumerative search, static analysis by abstract interpretation, and expression-directed enumerative search by SMT-based mutations;
(3) We build a prototype tool and evaluate it on a set of numerical C programs.

2 Motivating Examples

Let us illustrate our approach using a motivating example. Consider the program template `sum` shown in Fig. 1, which aims to compute the sum of all integers from 0 to a given input integer `n`. The unknown holes, denoted by `??`, are missing statements and expressions that need to be automatically computed. The specification in the form of an assertion indicates that the output represented by the variable `res` is $\texttt{n} * (\texttt{n}+1)/2$. Additionally, the synthesizer knows that integer variables $\{\texttt{n}, \texttt{res}, \texttt{i}\}$ and integer constants $\{0, 1\}$ are used. Given these input components, IMPSynth produces the correct solution in Fig. 4 in 1.455 seconds.

Our IMPSynth algorithm for synthesizing programs goes through three steps. We refer to the running example `sum` in Fig. 1 to demonstrate steps.

Step (1). The statement-directed enumeration search aims to fill in the statement holes in the input program template, but leaves the expression holes unfilled. This is done by searching through the space of all possible statements according to the language grammar in increasing size. To speed up the search, the algorithm employs some optimization techniques, such as exploring only terminating programs and avoiding to explore different but semantically equivalent programs. For example, the basic enumerative search applied to the template `sum` in Fig.1 generates the partial program with expression holes in Fig. 2, where commented statements are considered instead of the statements on the same lines. Subsequently, after applying the heuristic for exploring only

```
int sum(int n){
  assume(n ≥ 1);
  int res = 0;
  ??0
  while (??1) {
    ??2}
  assert (res==n*(n+1)/2);
  return res;
}
```

Figure 1. Program template sum.

```
int sum(int n){
⊤ assume(n ≥ 1); n ≥ 1
  int res = 0;
  int i = ??0;
n ≥ 1 ∧ res=0
  while (i<??1) { //while (??1) {
    res = ??2;
    i = i+??3; } //i = ??3; }
n ≥ 1
  assert (res==n*(n+1)/2);
  return res; }
```

Figure 2. Partial candidate program sum.

```
int sum(int n0){
  assume(n0 ≥ 1);
  int res0 = 0;
  int i0 = ??0;
  bool g0 = (i0<??1);
  if (g0) {
    res1 = ??2;
    i1 = i0+??3;
    g1 = (i1<??1);
    assume(¬g1); }
  res2 = g0?res1 : res0;
  assert (res2==n0*(n0+1)/2);
  return res2; }
```

Figure 3. Simplified program sum in SSA.

```
int sum(int n){
  assume(n ≥ 1);
  int res = 0;
  int i = 1;
  while (i ≤ n) {
    res = res+i;
    i = i+1; }
  assert (res==n*(n+1)/2);
  return res;
}
```

Figure 4. Correct program sum.

terminating programs we obtain the partial program in Fig. 2, where uncommented statements are considered.

Step (2). The partial program, in which missing expressions are treated as non-deterministic choices ($[-\infty, +\infty]$ for arithmetic and {true, false} for boolean expressions), is statically analyzed using a numerical abstract domain (e.g., Polyhedra). This way, we aim to prune away partial programs that will eventually fail to satisfy the given assertion. For example, we use boxed code, such as $\boxed{\top}$, to show the inferred invariants by the Polyhedra static analysis in some locations of the partial program in Fig. 2. From the invariant $\boxed{n \geq 1}$ before the final assertion, we cannot conclude whether the assertion is valid or not. The inferred result (conclusion) means that it is possible to find a correct program by filling in the missing expressions in the partial program in Fig. 2, so we continue to Step **(3)**.

However, for partial program in which variable res is not redefined after initialization, e.g.

$$\ldots \texttt{res = 0; i = ??}_0\texttt{; while (i<??}_1\texttt{) \{i = i+??}_3\texttt{; \}} \ldots$$

the abstract analysis will infer the invariant $\boxed{n \geq 1 \wedge \texttt{res=0}}$ before the assertion. This way, we obtain definite negative answer for the validity of the assertion, so continuing the search from this partial program will eventually fail. Hence, we go back to Step **(1)** to generate a new partial program. This shows that abstract semantics is often sufficient to prove the unrealizability of some partial programs (i.e. there is no solution that can be derived from them).

Step (3). The candidate partial program found in Step **(2)** is simplified (e.g., branch conditions are replaced with fresh boolean variables), unwinded by unrolling loop b times (the loop body is duplicated b times, where each copy is guarded by an if with the same condition), converted to static single assignment (SSA) form [6] (time-stamped versions of program variables are created, so that every variable is defined once), and finally translated to a SMT formula [3]. For example, the program in Fig. 3 is obtained by simplifying, unwinding for 1 time, and converting to SSA form the partial program in Fig. 2. Note that the condition ($\texttt{i0} < \texttt{??}_1$) is assigned to a fresh boolean variable g0, the first assignment to res is replaced by an assignment to res0, the second by an assignment to res1, etc. We use Φ-assignments (e.g., res2 = g0?res1 : res0) to determine which copy of res will be used after if-s. Inside the if we use assume(¬g1) to block all paths that go through the loop longer than the bound 1. The SMT formula corresponding to the simplified program

in Fig. 3 is a conjunction of all SMT formulas in the set:

$$S_{\texttt{sum}}=\{\texttt{n0}\geq\texttt{1},\texttt{res0=0},\big(\texttt{g0=(i0<}??_1\texttt{)}\wedge\neg(\texttt{g1=(i1<}??_1\texttt{)})\big), \\ \texttt{i0=}??_0\texttt{,res1=}??_2\texttt{, i1=i0+}??_3\texttt{,} \\ \texttt{res2=ite(g0,res1,res0)},\neg\texttt{(res2==n0*(n0+1)/2)}\} \quad (1)$$

Note that one formula in $S_{\texttt{sum}}$ encodes a single statement in the simplified program. Hence, `res2 = g0?res1 : res0` is converted to the formula `res2=ite(g0,res1,res0)`, which means $(\texttt{g0}\wedge\texttt{res2=res1})\vee(\neg\texttt{g0}\wedge\texttt{res2=res0})$, whereas `assume(n ≥ 1)` is converted to $(\texttt{n0}\geq 1)$ and the assertion `assert(res2==n0*(n0+1)/2)` is converted to the formula $\neg$`(res2==n0*(n0+1)/2)`. The set $S_{\texttt{sum}}$ is partitioned into two subsets: $S^{\texttt{soft}}_{\texttt{sum}}$ that includes all formulas with expression holes and $S^{\texttt{hard}}_{\texttt{sum}}$ that includes all other formulas. A complete program with all expression holes filled in satisfies all its assertions iff the corresponding program formula is unsatisfiable. Therefore, by making mutations in the set $S^{\texttt{soft}}_{\texttt{sum}}$ of SMT formulas (by replacing holes with complete expressions), we aim to construct an unsatisfiable program formula. For example, if the hole $??_0$ is replaced with constant `0` and we use mutations that replace a constant with any other constant from Γ (that is, 1) or with any parameter variable from V (that is, `n0`), then for the soft formula $(\texttt{i0=}??_0)$ we will generate the following mutations: $(\texttt{i0=0})$, $(\texttt{i0=1})$, and $(\texttt{i0=n0})$. Note that $\big(\texttt{g0=(i0<}??_1\texttt{)}\wedge\neg\texttt{(g1=(i1<}??_1\texttt{)})\big)$ is considered as a single formula in $S_{\texttt{sum}}$ corresponding to the `while`-guard in the program in Fig. 2 before simplification. Thus, all mutations will be uniformly applied to both sub-formulas. For example, if the hole $??_1$ is replaced with constant `0` and we use mutations that replace a constant with any other constant from Γ (that is, 1) or with any parameter variable from V (that is, `n0`), and additionally we use mutations that replace the relational operator $<$ with $\leq$, then for the soft formula $\big(\texttt{g0=(i0<}??_1\texttt{)}\wedge\neg\texttt{(g1=(i1<}??_1\texttt{)})\big)$ we will generate the following mutations: $\big(\texttt{g0=(i0<0)}\wedge\neg\texttt{(g1=(i1<0))}\big)$, $\big(\texttt{g0=(i0}\leq\texttt{0)}\wedge\neg\texttt{(g1=(i1}\leq\texttt{0))}\big)$, $\big(\texttt{g0=(i0<1)}\wedge\neg\texttt{(g1=(i1<1))}\big)$, $\big(\texttt{g0=(i0}\leq\texttt{1)}\wedge\neg\texttt{(g1=(i1}\leq\texttt{1))}\big)$, $\big(\texttt{g0=(i0<n0)}\wedge\neg\texttt{(g1=(i1<n0))}\big)$, and $\big(\texttt{g0=(i0}\leq\texttt{n0)}\wedge\neg\texttt{(g1=(i1}\leq\texttt{n0))}\big)$.

Finally, if we generate the following mutations $(\texttt{i0=1})$, $(\texttt{g0=i0}\leq\texttt{n0})$, $(\texttt{g1=i1}\leq\texttt{n0})$, $(\texttt{res1=res0+i0})$, $(\texttt{i1=i0+1})$ to the set $S_{\texttt{sum}}$, we obtain an unsatisfiable program formula. This way, we report the program in Fig. 4 corresponding to the above formulas as correct solution to the given problem.

3 Program Synthesis Problem

This section introduces the syntax of the target language, its concrete semantics, as well as the formal definition of the program synthesis problem.

Language. We use a simple C-like imperative language for writing general-purpose programs. The program variables *Var* are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. To encode templates, three hole constructs are used: $??^s$, $??^{ae}$, $??^{be}$ for unknown statements, arithmetic expressions, and boolean expressions. The hole constructs are placeholders that will get replaced with suitable statements and expressions, such that the resulting program will satisfy all assertions. The syntax of the language is:

$$\begin{array}{l} s\ (s\in \mathit{Stm}) ::= \texttt{skip} \mid \texttt{x=}ae \mid ??^s \mid s;s \mid \texttt{if}\ (be)\ s\ \texttt{else}\ s \\ \qquad\qquad \mid \texttt{while}\ (be)\ \texttt{do}\ s \mid \texttt{assume}(be), \\ ae\ (ae\in \mathit{AExp}) ::= n\in\mathbb{Z} \mid [n,n'] \mid \texttt{x}\in \mathit{Var} \mid ??^{ae} \mid ae\oplus ae, \\ be\ (be\in \mathit{BExp}) ::= ae\bowtie ae \mid ??^{be} \mid \neg be \mid be\wedge be \mid be\vee be \end{array} \quad (2)$$

where $[n,n']$ ranges over integer intervals, $\oplus\in\{+,-,*,\%,\div\}$, and $\bowtie\in\{<,\leq,>,\geq,==,!=\}$. W.l.o.g., we assume that a program $p\in \mathit{Prog}$ is a sequence of statements followed by a single assertion: "s; `assert`(be^f)". We say that a program (statement) is *complete* if it does not have any holes; otherwise, we say that it is *partial* (or *program template*). A partial program (statement) that contains only expression holes ($??^{ae}$, $??^{be}$) is called *semi-partial*.

$$\begin{array}{l} [\![\texttt{skip}]\!]\mathcal{E}=\mathcal{E} \\ [\![??^s]\!]\mathcal{E}=\Sigma \\ [\![\texttt{x}=ae]\!]\mathcal{E}=\{\sigma[\texttt{x}\mapsto n]\mid\sigma\in\mathcal{E}, n\in[\![ae]\!]\sigma\} \\ [\![s_1\ ;\ s_2]\!]\mathcal{E}=[\![s_2]\!]([\![s_1]\!]\mathcal{E}) \\ [\![\texttt{assume}(be)]\!]\mathcal{E}=\{\sigma\in\mathcal{E}\mid[\![be]\!]\sigma=\text{true}\} \\ [\![\texttt{if}\ (be)\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\mathcal{E}= \\ \quad [\![s_1]\!]\{\sigma\in\mathcal{E}\mid\text{true}\in[\![be]\!]\sigma\}\cup[\![s_2]\!]\{\sigma\in\mathcal{E}\mid\text{false}\in[\![be]\!]\sigma\} \\ [\![\texttt{while}\ (be)\ \texttt{do}\ s]\!]\mathcal{E}=\{\sigma\in\texttt{lfp}\ \phi_{\mathcal{E}}\mid\text{false}\in[\![be]\!]\sigma\} \\ \phi_{\mathcal{E}}(X)=\mathcal{E}\cup[\![s]\!]\{\sigma\in X\mid\text{true}\in[\![be]\!]\sigma\} \end{array}$$

Figure 5. Definitions of $[\![s]\!]:\mathcal{P}(\Sigma)\to\mathcal{P}(\Sigma)$.

Concrete Semantics. A memory *store*, denoted $\sigma\in\Sigma$, is a function mapping each variable to a value: $\Sigma=\mathit{Var}\to\mathbb{Z}$. The concrete domain is the powerset complete lattice, $\langle\mathcal{P}(\Sigma),\subseteq,\cup,\cap,\emptyset,\Sigma\rangle$. The semantics of arithmetic $[\![ae]\!]:\Sigma\to\mathcal{P}(\mathbb{Z})$ and boolean expressions $[\![be]\!]:\Sigma\to\mathcal{P}(\{\text{true},\text{false}\})$ are the sets of possible values for expressions ae and be in a given store σ. For example, $[\![??^{ae}]\!]\sigma=\mathbb{Z}$, $[\![[n,n']]\!]\sigma=\{n,\ldots,n'\}$, and $[\![??^{be}]\!]\sigma=\{\text{true},\text{false}\}$ for any $\sigma\in\Sigma$. Figure 5 presents the *invariance* (reachability) semantics $[\![s]\!]:\mathcal{P}(\Sigma)\to\mathcal{P}(\Sigma)$ that infers a set of reachable stores (invariants) from a given set of initial stores. Each function $[\![s]\!]$ takes as input a set of stores $\mathcal{E}\in\mathcal{P}(\Sigma)$ before statement s and returns a set of possible stores reached after executing s from $\mathcal{E}$. The semantics of `while` is given using the least fix-point operator `lfp` [4, 30], where the fixed-point functional $\phi_{\mathcal{E}}:\mathcal{P}(\Sigma)\to\mathcal{P}(\Sigma)$, parameterized with the initial set of stores $\mathcal{E}$, accumulates possible stores after another `while` iteration from a set of stores X. Note that the semantics of (statement and expression) holes is defined as non-deterministic choice of all values of the corresponding type (e.g., $[\![??^s]\!]\mathcal{E}=\Sigma$).

Problem Definition. The program synthesis problem is defined as triple:

$$(p_0, \Gamma, V)$$

where $p_0 \equiv s_0$; $\texttt{assert}(be^f)$ is an initial program template, $\Gamma \subseteq \mathbb{Z}$ is a set of integer constants, and $V \subseteq Var$ is a set of integer variables. A *solution* to the program synthesis problem is a complete program $p \equiv s$; $\texttt{assert}(be^f)$ derived from p_0, such that p uses constants and variables from Γ and V, and furthermore p satisfies the assertion be^f under all possible inputs (i.e., the initial set of stores Σ). That is, $[\![s]\!]\Sigma \subseteq \{\sigma \in \Sigma \mid [\![be^f]\!]\sigma = \{\text{true}\}\}$.

4 Synthesis Algorithm

Our language is too large to perform an exhaustive search of all possible complete programs. To make the search more practical, we use the following three techniques: basic statement directed enumeration, pruning via abstract static analysis, and expression directed enumeration via (incremental) SMT-based mutations. The following subsections discuss each of these three techniques in more detail, and then we present the overall synthesis algorithm for generating a correct-by-construction complete program.

4.1 Statement-directed Enumeration

We describe statement directed enumerative search, which is defined by the transition system:

$$(S, \rightarrow, s_0, F)$$

where S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation, $s_0 \in S$ is an initial state, and $F \subseteq S$ is a set of final states. A *state* $s \in S$ is a statement that may have holes, while an *initial state* $s_0 \in S$ is an initial partial statement that we want to fill in. The *transition relation* $\rightarrow \subseteq S \times S$ determines a next state reachable from a given state. We write $s_1 \rightarrow s_2$ whenever $(s_1, s_2) \in \rightarrow$. Figure 6 defines the transition relation via a set of inference rules. Intuitively, rules $s_1 \rightarrow s_2$ indicate that a statement hole in s_1 can be replaced by an arbitrary statement (excluding `assume` and `assert`) described by s_2. We write $\texttt{next}(s)$ for the states immediately reachable from s, $\texttt{next}(s) = \{s' \in S \mid s \rightarrow s'\}$. A *final state* $s \in F$ is a semi-partial statement with no statement holes, but with expression holes. The goal of the statement-directed enumerative search is to derive a final state.

Example 4.1. Let $\Gamma = \{0, 1\}$, $V = \{\texttt{x}\}$, and $s = (\texttt{x} = ??^{ae}; ??^{s})$ be given. Then, $\texttt{next}(s)$ consists of the following states:
$(\texttt{x} = ??^{ae}; \texttt{x} = ??^{ae})$, $(\texttt{x} = ??^{ae}; \texttt{skip})$,
$(\texttt{x} = ??^{ae}; \texttt{if}\ (??^{be})\ ??^{s}\ \texttt{else}\ ??^{s})$,
$(\texttt{x} = ??^{ae}; ??^{s}; ??^{s})$, $(\texttt{x} = ??^{ae}; \texttt{while}\ (??^{be})\ \texttt{do}\ ??^{s})$.

Example 4.2. Reconsider the running example `sum` from Section 2. Let $\Gamma = \{0, 1\}$, $V = \{\texttt{n}, \texttt{sum}, \texttt{i}\}$, $\texttt{sum}_0$ and $\texttt{sum}_1$ be partial statements in Fig. 1 and Fig. 2 respectively, where for $\texttt{sum}_1$ we use commented statements instead of uncommented ones in the same lines. Then, we have that $\texttt{sum}_0 \rightarrow^* \texttt{sum}_1$ and $\texttt{sum}_1 \in F$, where $\rightarrow^*$ is the transitive closure of $\rightarrow$.

$$\frac{s_1 \rightarrow s_1'}{s_1; s_2 \rightarrow s_1'; s_2} \qquad \frac{s_2 \rightarrow s_2'}{s_1; s_2 \rightarrow s_1; s_2'}$$

$$\frac{s_1 \rightarrow s_1'}{\texttt{if}\ (be)\ s_1\ \texttt{else}\ s_2 \rightarrow \texttt{if}\ (be)\ s_1'\ \texttt{else}\ s_2}$$

$$\frac{s_2 \rightarrow s_2'}{\texttt{if}\ (be)\ s_1\ \texttt{else}\ s_2 \rightarrow \texttt{if}\ (be)\ s_1\ \texttt{else}\ s_2'}$$

$$\frac{s \rightarrow s'}{\texttt{while}\ (be)\ \texttt{do}\ s \rightarrow \texttt{while}\ (be)\ \texttt{do}\ s'}$$

$$\frac{\texttt{x} \in V}{??^{s} \rightarrow \texttt{x=}??^{ae}} \qquad \frac{}{??^{s} \rightarrow \texttt{skip}} \qquad \frac{}{??^{s} \rightarrow ??^{s}; ??^{s}}$$

$$\frac{}{??^{s} \rightarrow \texttt{if}\ (??^{be})\ ??^{s}\ \texttt{else}\ ??^{s}}$$

$$\frac{}{??^{s} \rightarrow \texttt{while}\ (??^{be})\ \texttt{do}\ ??^{s}}$$

Figure 6. Statement-directed transition relation.

We also perform some optimization techniques in order to speed up the enumerative search. We maintain a set of previously explored states and never reconsider them. Whenever we explore a state with two consecutive assignments to the same variable, we normalize it to the state with one assignment. For example, the state $(\texttt{x:=}??^{ae}; \texttt{skip}; \texttt{x:=}??^{ae}; ??^{s})$ is normalized to $(\texttt{x:=}??^{ae}; ??^{s})$ and added to the workset only when the resulting state is not explored before.

In addition, after the search has produced final states, we use a heuristic to generate terminating programs and to exclude potentially non-terminating programs. We use one induction variable $\texttt{x} \in V$ for each `while` loop. Firstly, we only allow boolean expressions of the form `x<??` (resp., `x>??`) in `while` conditions. Secondly, the last statement of the `while` body must increase (resp., decrease) the induction variable `x`. Finally, `x` should not be defined elsewhere in the `while` body. For example, after applying this heuristic to the final state $\texttt{sum}_1$, we obtain the semi-partial program in Fig. 2 with uncommented statements, where the induction variable `i` is used to ensure the termination of the `while`-loop.

4.2 Pruning via Abstract Semantics

We say that a final state (i.e., semi-partial statement) s is a *failure* (resp., *success*) iff every complete program derived from s does not (resp., does) satisfy the assertion be^f, that is, $[\![s]\!]\Sigma \subseteq \{\sigma \in \Sigma \mid [\![be^f]\!]\sigma = \{\text{false}\}\}$ (resp., $[\![s]\!]\Sigma \subseteq \{\sigma \in \Sigma \mid [\![be^f]\!]\sigma = \{\text{true}\}\}$). Our goal is to find as many failure states as possible. But, if a success state is found, the algorithm reports any complete program derived from that state as solution.

However, the concrete semantics $[\![s]\!]$ is uncomputable, since our language is Turing complete. Therefore, we now define the computable abstract semantics [4, 30, 42] that is an

$$
\begin{array}{l}
[\![\texttt{skip}]\!]^\sharp d = d \\
[\![\texttt{??}^s]\!]^\sharp d = \top_{\mathbb{D}} \\
[\![\texttt{x = } ae]\!]^\sharp d = \texttt{ASSIGN}_{\mathbb{D}}(\texttt{x = } ae, d) \\
[\![s_1 \texttt{ ; } s_2]\!]^\sharp d = [\![s_2]\!]^\sharp([\![s_1]\!]^\sharp d) \\
[\![\texttt{assume}(be)]\!]^\sharp d = \texttt{FILTER}_{\mathbb{D}}(be, d) \\
[\![\texttt{if } (be) \texttt{ then } s_1 \texttt{ else } s_2]\!]^\sharp d = \\
\quad [\![s_1]\!]^\sharp(\texttt{FILTER}_{\mathbb{D}}(be, d)) \sqcup_{\mathbb{D}} [\![s_2]\!]^\sharp(\texttt{FILTER}_{\mathbb{D}}(\neg be, d)) \\
[\![\texttt{while } (be) \texttt{ do } s]\!]^\sharp d = \texttt{FILTER}_{\mathbb{D}}(\neg be, \texttt{lfp}^\sharp\, \phi_d^\sharp) \\
\phi_d^\sharp(x) = d \sqcup_{\mathbb{D}} [\![s]\!]^\sharp(\texttt{FILTER}_{\mathbb{D}}(be, x))
\end{array}
$$

Figure 7. Definitions of $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$.

over-approximation of the concrete semantics. We replace the computation in the concrete domain $\mathcal{P}(\Sigma)$ with a computation in some numerical abstract domain $\mathbb{D}$ that reasons on the numerical properties of variables, such that there exists a concretization-based abstraction $\langle \mathcal{P}(\Sigma), \subseteq \rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$. Note that concretization-based abstraction is a relaxation of the known Galois connection abstraction that is equipped with two monotonic functions: abstraction $\alpha_{\mathbb{D}}$ and concretization $\gamma_{\mathbb{D}}$. However, some abstract domains often used in practice (e.g., Polyhedra domain [5]) do not enjoy Galois connection, but we can still reason about their soundness through the concretization function $\gamma_{\mathbb{D}}$ and the relaxed concretization-based abstraction [30]. The concretization function $\gamma_{\mathbb{D}} : \mathbb{D} \to \mathcal{P}(\Sigma)$ expresses the meaning of abstract elements from $\mathbb{D}$ in terms of concrete elements from $\mathcal{P}(\Sigma)$. The abstract domain $\mathbb{D}$ is a set of computer-representable properties, called *abstract elements*, together with effective algorithms to implement sound abstract operators for static analysis. In particular, there are abstract operators for ordering $\sqsubseteq_{\mathbb{D}}$, least upper bound (join) $\sqcup_{\mathbb{D}}$, greatest lower bound (meet) $\sqcap_{\mathbb{D}}$, bottom $\bot_{\mathbb{D}}$, top $\top_{\mathbb{D}}$, widening $\triangledown_{\mathbb{D}}$, and narrowing $\triangle_{\mathbb{D}}$. There are transfer functions for forward assignments $\texttt{ASSIGN}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$ and tests $\texttt{FILTER}_{\mathbb{D}} : \mathit{BExp} \times \mathbb{D} \to \mathbb{D}$, which are sound over-approximations of the corresponding concrete functions. We let $\texttt{lfp}^\sharp$ denote an abstract fix-point operator, derived using widening $\triangledown_{\mathbb{D}}$ and narrowing $\triangle_{\mathbb{D}}$ that over-approximates the concrete lfp.

The operators of the abstract domain $\mathbb{D}$ can be used to define abstract invariance analysis that is over-approximation of the corresponding concrete analysis. For each statement s, we define its abstract invariance semantics $[\![s]\!]^\sharp$ in Fig. 7. For a while loop, $\texttt{lfp}^\sharp\, \phi_d^\sharp$ is the limit of the following increasing chain: $y_0 = d,\ y_{n+1} = y_n \triangledown_{\mathbb{D}} \phi_d^\sharp(y_n)$. All expression holes are treated as non-deterministic choices: $[-\infty, +\infty]$ for arithmetic, and {true, false} for boolean expressions. That is, we use $\texttt{FILTER}_{\mathbb{D}}(\texttt{??}^{be}, d) = d$ to handle boolean expression holes $\texttt{??}^{be}$, and $\texttt{ASSIGN}_{\mathbb{D}}(\texttt{x = ??}^{ae}, d) = \texttt{ASSIGN}_{\mathbb{D}}(\texttt{x = } [-\infty, +\infty], d)$ to handle arithmetic expression holes $\texttt{??}^{ae}$.

By using the soundness of the abstract domain $\mathbb{D}$ [4, 30], we prove the soundness of the abstract semantics with respect to concrete semantics. That is, $[\![s]\!]^\sharp$ over-approximates the set of reachable stores found by concrete semantics $[\![s]\!]$.

Proposition 4.3 ([4, 30]). *Let $\mathcal{I} = \gamma_{\mathbb{D}}(d_{\mathcal{I}})$ and s be a statement: $[\![s]\!]\mathcal{I} \subseteq \gamma_{\mathbb{D}}([\![s]\!]^\sharp d_{\mathcal{I}})$.*

Corollary 4.4. *Let s; $\texttt{assert}(be^f)$ be a semi-partial program and $[\![s]\!]^\sharp \top_{\mathbb{D}} \neq \bot_{\mathbb{D}}$. If $\texttt{FILTER}_{\mathbb{D}}(be^f, [\![s]\!]^\sharp \top_{\mathbb{D}}) = \bot_{\mathbb{D}}$, then s is a failure state. If $\texttt{FILTER}_{\mathbb{D}}(\neg be^f, [\![s]\!]^\sharp \top_{\mathbb{D}}) = \bot_{\mathbb{D}}$, then s is a success state.*

In practice, the domain $\mathbb{D}$ will be instantiated with some known numerical domains, such as the Polyhedra domain $\langle P, \sqsubseteq_P \rangle$ [5] whose elements are conjunctions of polyhedra linear constraints of the form $\alpha_1 x_1 + \ldots + \alpha_k x_k + \alpha_{k+1} \geq 0$, where $x_1, \ldots x_k \in \mathit{Var}, \alpha_1, \ldots, \alpha_k, \alpha_{k+1} \in \mathbb{Z}$.

4.3 Expression-directed Enumeration via SMT-based Mutations

From programs to SMT formulas. We now give an overview on the translation of a semi-partial program with expression holes into a set of SMT formulas using the CBMC bounded model checker [3]. The input semi-partial program undergoes three transformations: simplification, unwinding, and conversion to SSA form [6]. The simplification ensures that all if and while conditions are replaced with fresh boolean variables, while the SSA-form guarantees that each local variable has a single static point of definition. The unwinding with bound b means that all while loops are unrolled b times, so that we only consider *b-bounded paths* that are executing the body of the loop at most b times. After the above three transformations, in the generated simplified program all expressions from the original semi-partial program are right-hand sides (RHSs) of assignments, loops are replaced with if-s, and each variable is assigned once. For example, the simplified program in Fig. 3 is obtained from the semi-partial program in Fig. 2 by the above three transformations.

The simplified program is then converted to a set of SMT formulas S as follows. An assignment x=ae is converted to equation formula x=ae; a Φ-assignment x = be?x1 : x2 is converted to formula x=ite(be,x1,x2); an assume(be) is converted to formula be; and an assert(be) is converted to formula $\neg be$. This way, a formula in S encodes a statement in the simplified program. However, a statement that is part of a while body may be encoded by several formulas $\phi_1, \ldots, \phi_k$ in S due to the unwinding. In this case, instead of adding formulas $\phi_1, \ldots, \phi_k$ separately in S, we add one conjunctive formula $(\phi_1 \wedge \ldots \wedge \phi_k)$ in S. In effect, we obtain that one formula in the set S encodes single statement in the original program. For example, the set $S_{\texttt{sum}}$ in Eqn. (1) is obtained from the simplified program in Fig. 3.

The obtained set S is partitioned into two subsets: $S^{\texttt{soft}}$ that includes all formulas corresponding to statements with expressions holes, and $S^{\texttt{hard}}$ that includes all other formulas corresponding to assertions, assumptions, Φ-assignments, and assignments with complete expressions. For example, $S_{\texttt{sum}}$ in Eqn. (1) is partitioned as:

$$S^{\texttt{soft}}_{\texttt{sum}} = \{\texttt{i0=??}_0, (\texttt{g0=(i0<??}_1) \wedge \neg(\texttt{g1=(i1<??}_1))), \texttt{sum1=??}_2\texttt{i1=i0+??}_3\},$$
$$S^{\texttt{hard}}_{\texttt{sum}} = \{\texttt{n0} \geq 1, \texttt{sum0=0}, \texttt{sum2=ite(g0,sum1,sum0)}, \neg\texttt{(sum2=n0*(n0+1)/2)}\}$$

where formulas $(\texttt{i0=??}_0)$, $\big(\texttt{g0=(i0<??}_1)\wedge\neg\texttt{(g1=(i1<??}_1)\texttt{)}\big)$, $(\texttt{sum1=??}_2)$, $(\texttt{i1=i0+??}_3)$ correspond to statements with expression holes in the program in Fig. 2 before simplification.

Given a program p, the *program formula* φ^b_p is the conjunction of all formulas in S, which encodes all possible b-bounded paths through the program p. For example, in the simplified program in Fig. 3 and $S_{\texttt{sum}}$ we use $\texttt{assume}(\neg g1)$ and the SMT formula $(\neg g1)$ to block all paths longer than the unrolling bound 1. We say that a complete program p is *b-bounded correct* if all assertions in it are valid in all b-bounded paths of p. We omit to write p and b in φ^b_p when they are clear.

Proposition 4.5 ([3]). *Complete program p is b-bounded correct iff φ^b_p is unsatisfiable.*

To enable incremental SMT solving, the program formula φ is instrumented with boolean variables called *guard variables*. More specifically, a formula $\varphi = \phi_1 \wedge \ldots \wedge \phi_n$ is replaced with $\varphi' = (x_1 \implies \phi_1) \wedge \ldots \wedge (x_n \implies \phi_n)$, where $x_1, \ldots, x_n$ are fresh guard variables. In effect, the formula $(x_i \implies \phi_i)$ can be satisfied by setting x_i to false. Some guard variables called *assumptions* are conjuncted with φ' and passed to an incremental SMT solver. For example, $\varphi' \wedge x_1 \wedge x_2$ is satisfiable iff ϕ_1 and ϕ_2 are satisfiable, since the satisfying assignment will set $x_3, \ldots, x_n$ to false thus making the sub-formulas $(x_3 \implies \phi_3), \ldots, (x_n \implies \phi_n)$ true. Thus, an incremental SMT-solver checking the satisfiability of $\varphi' \wedge x_1 \wedge x_2$ will only check satisfiability of ϕ_1 and ϕ_2, thus essentially disabling formulas $\phi_3, \ldots, \phi_n$.

Finally, we will use a formula $\texttt{AtMost}(\{l_1, \ldots, l_n\}, k)$ (resp., $\texttt{AtLeast}(\{l_1, \ldots, l_n\}, k)$) to require that at most (resp., at least) k of the literals $l_1, \ldots, l_n$ are true [26]. They are called *Boolean cardinality formulas* encoding that $\sum_{i=1}^n l_i \leq k$ (resp., $\sum_{i=1}^n l_i \geq k$), where l_i is a literal assigned the value 1 if true and the value 0 if false, and $k \in \mathbb{N}$. We will use the MiniCard SAT-solver [26] to check their satisfiability.

Let H be a set of uniquely labelled expression holes $??_i$ in semi-partial program p. A control function h is a mapping from the set of holes H to *complete expressions*, which are defined to be expressions without expression holes ($??^{ae}$, $??^{be}$). We write $p[h]$ (resp., $\varphi^b_{p[h]}$) to denote the complete program (resp., complete program formula) obtained by substituting each $??_i$ with $h(??_i)$. Moreover, $S^{\texttt{soft}}[h] = \{\phi[h] \mid \phi \in S^{\texttt{soft}}\}$.

Algorithm 1: ExprMutate(p, Γ, V, b, r)

Input: Semi-partial program p, constants Γ, vars V, unrolling bound b, size r
Output: Solution p' or $\emptyset$

1 $l := 1$;
2 **while** $(l \leq r)$ **do**
3 $\quad h^l := \texttt{GenComplExps}_{\Gamma,V}(l)$;
4 $\quad (S^{\texttt{hard}}, S^{\texttt{soft}}[h^l]) := \texttt{CBMC}(p[h^l], b)$;
5 $\quad (S_1, \ldots, S_n) := \texttt{Mutate}(S^{\texttt{soft}}[h^l])$;
6 $\quad (S'_1, \ldots, S'_n, G_1, \ldots, G_n, G_{\texttt{orig}}) :=$ $\texttt{InstGuardVars}(S_1, \ldots, S_n)$;
7 $\quad \texttt{inst}(\varphi^b_{p[h^l]}) := (\wedge_{s \in S_{\texttt{hard}}} s) \wedge (\wedge_{s \in S'_1 \cup \ldots \cup S'_n} s)$;
8 $\quad \varphi := (\wedge_{i=1}^n \texttt{AtMost}(G_i, 1)) \wedge (\wedge_{i=1}^n \texttt{AtLeast}(G_i, 1))$;
9 $\quad k := 1$;
10 $\quad$ **while** $(k \leq n)$ **do**
11 $\quad\quad \varphi_k := \varphi \wedge \texttt{AtLeast}(G_{\texttt{orig}}, n-k)$;
12 $\quad\quad satres, G := \texttt{SAT}(\varphi_k)$;
13 $\quad\quad$ **if** $(satres)$ **then**
14 $\quad\quad\quad smtres := \texttt{SMT}(\texttt{inst}(\varphi^b_{p[h^l]}) \wedge \wedge_{g \in G} g)$;
15 $\quad\quad\quad$ **if** $(\neg smtres)$ **then return** $p[G]$;
16 $\quad\quad\quad \varphi := \varphi \wedge (\vee_{g \in G} \neg g)$;
17 $\quad\quad$ **else**
18 $\quad\quad\quad k := k + 1$;
19 $\quad\quad$ **if** $(Timeout)$ **then raise "Timeout"** ;
20 $\quad l := l + 1$;
21 **return** $\emptyset$;

Algorithm. We now describe our algorithm for expression-directed enumerative search. The ExprMutate procedure is given in Algorithm 1. It takes as input a semi-partial program p, sets of constants Γ and variables V, an unrolling bound b, and a size r that limits the search space to only expressions derived using at most r production rules from the expression grammars. The algorithm goes through an iterative generate-and-verify procedure. We use a SAT solver in the generation phase to find a candidate program from the search space, and a SMT solver in the verification phase to check if the candidate program is correct.

ExprMutate explores the space of all complete expressions substituted for expression holes in increasing size order using the variable l, which is initialized to 1 and increased after each iteration (lines 2–20). It calls the $\texttt{GenComplExp}_{\Gamma,V}(l)$ procedure to generate one control function h^l that maps each expression hole from H to a complete expression generated from the expression grammars in Eqn. (2), such that its derivation tree has height (size) l (line 3). The derivation tree of an expression is obtained by applying a sequence of production rules from the expression grammar to the tree whose root is the initial nonterminal ae or be. Note that the expression grammars are parameterized with Γ and V, so

that only expressions containing constants from Γ and variables from V can be derived. For example, the derivation tree of $x + 3$ has height 2, and is generated by the grammar in Eqn. (2) with the initial nonterminal *ae* and $\Gamma = \{3\}, V = \{x\}$.

Once the initial candidate (complete) program $p[h^l]$ is constructed, we call the CBMC translation procedure to generate the pair $(S^{\mathsf{hard}}, S^{\mathsf{soft}}[h^l])$ of sets of formulas corresponding to $p[h^l]$ (line 4). Next, we call the `Mutate` procedure to generate all possible mutations $S_1, \ldots, S_n$ of formulas in $S^{\mathsf{soft}}[h^l]$ (line 5), where S_i is a set of formulas obtained by mutating some $\phi_i \in S^{\mathsf{soft}}[h^l]$. Hence, $S_1, \ldots, S_n$ correspond to *n program locations* where expression holes occur in p. A *mutation* is a replacement of an expression with another expression of the same type. We maintain a fixed list of mutations for each type of expressions:

type of exp.	mutation
`arithmetic op.`	$\{+, -, *, \%, \div\}$
`relational op. while guards`	$\{<, \leq\},\ \{>, \geq\}$
`relational op.`	$\{<, \leq, >, \geq, ==, !=\}$
`logical op.`	$\{\&\&, \|\|\}$
`program vars` $x \in V$	$x \rightarrow n\ (n \in \Gamma), x \rightarrow y\ (y \in V)$
`int. constants` $n \in \Gamma$	$n \rightarrow m\ (m \in \Gamma), n \rightarrow x\ (x \in V)$

The mutation set $\{+, -, *, \%, \div\}$ means that any arithmetic operator from the set can be replaced with any other operator from the same set, while the mutation rule $x \rightarrow n\ (n \in \Gamma)$ means that the program variable $x \in V$ can be replaced with any constant $n \in \Gamma$. For example, arithmetic expression $x + 3$ can be replaced with $x - 3$, $x * 3$, $x\%3$, and $x \div 3$. Let us assume that mutations $M_1, \ldots, M_j$ can be applied to a formula $\phi \in S^{\mathsf{soft}}[h^l]$. Then, $\mathtt{Mutate}(\phi) = \{\phi, M_1(\phi), \ldots, M_j(\phi)\}$. Finally, we have

$$\mathtt{Mutate}(S^{\mathsf{soft}}[h^l]) = \big(\mathtt{Mutate}(\phi_1), \ldots, \mathtt{Mutate}(\phi_n)\big)$$

Then, we use the `InstGuardVars` procedure to instrument all formulas in $S_1, \ldots, S_n$ by fresh guard variables, so that the results are sets of instrumented formulas $S'_1, \ldots, S'_n$ and sets of fresh guard variables $G_1, \ldots, G_n$ used to guard formulas in $S'_1, \ldots, S'_n$ (line 6). Here $S'_i = \{(x \implies \phi) \mid \phi \in S_i, x \in G_i \text{ is a fresh guard variable}\}$. The set $G_{\mathtt{orig}}$ contains guard variables corresponding to original formulas in $S^{\mathsf{soft}}[h^l]$. The formula $\mathtt{inst}(\varphi^b_{p[h^l]})$ is the conjunction of all formulas in S^{hard} and all instrumented formulas from $S'_1 \cup \ldots \cup S'_n$ (line 7).

Subsequently, we search the space of all mutated formulas in increasing mutation size order, which measures how many changes to original formulas in $S^{\mathsf{soft}}[h^l]$ are applied, by using the variable k (lines 10–19). In particular, we generate the boolean formula φ_k (line 11) expressing that at most k guard variables are not original, i.e. at least $n - k$ are original (by using $\mathtt{AtLeast}(G_{\mathtt{orig}}, n - k)$). We use the boolean formula φ (line 8) to encode that there is exactly one guard variable selected for each of the n statements containing expression holes in the program (by using $\varphi \equiv \wedge_{i=1}^{n} \mathtt{AtMost}(G_i, 1) \wedge \wedge_{i=1}^{n} \mathtt{AtLeast}(G_i, 1)$). The boolean formula φ_k is sent to a SAT solver (line 12), where each satisfying assignment of φ_k corresponds to a mutated (complete) program in the search space with at most k changes applied to the original program $p[h^l]$. If φ_k is unsatisfiable, this means that there are no unexplored mutated program formulas of size k so we increase k by one (line 18) and start a new iteration of the nested `while` loop. If φ_k is satisfiable, we store in a set G all guard variables assigned true in the given satisfying assignment (model) of φ_k resulting from the SAT call (line 12). We call a SMT solver to check the program corresponding to $\mathtt{inst}(\varphi^b_{p[h^l]}) \wedge \wedge_{g \in G} g$, i.e. $\mathtt{inst}(\varphi^b_{p[h^l]})$ with all guards in G passed as assumptions (line 14). This is the same to checking the conjunction of all formulas in S^{hard} and all soft formulas guarded by variables in G, since all other soft formulas will get satisfied by setting their guard variables to false. If the result is false, we report the complete program $p[G]$, where all expression holes are substituted according to the guards in G and their respective formulas, as a solution (line 15). Otherwise, the mutated program is not correct so we block G from further exploration by adding the blocking clause $(\vee_{g \in G} \neg g)$ to the boolean formula φ (line 16). In the worst case, the nested `while` loop will terminate when the entire search space of mutated programs of $p[h^l]$ is inspected ($k > n$, line 10). The algorithm terminates when either the solution is found (line 15), or the space of programs with at most r-sized complete expressions substituted for holes is inspected ($l > r$, line 2) or a time limit is reached.

Theorem 4.6. *The* ExprMutate*(p, Γ, V, b, r) is b-bounded correct modulo the expression size r and terminating.*

Proof Sketch. By Proposition 4.5, it follows that every solution returned by ExprMutate is b-bounded correct (soundness). Moreover, every b-bounded correct program containing expressions of size at most r will eventually be returned by ExpMutate (relative completeness). That is, if there exists a correct program, we can take large enough bounds b and r so that we will eventually obtain the solution.

The ExprMutate always terminates as there are only finitely many expressions up to size r to be explored (i.e., the outer `while`-loop terminates). Moreover, for any candidate complete program $p[h^l]$ generated via $\mathtt{GenComplExp}_{\Gamma,V}(l)$ by expressions of some fixed size l ($l \leq r$), there are only finitely many mutations that can be applied so the algorithm enumerates all possible mutants until it either finds the minimal correct one or establishes that there is no solution (i.e., the inner `while`-loop terminates). □

Algorithm 2: IMPSynth(p_0, Γ, V, b, r)

Input: Synthesis problem (p_0, Γ, V), unrolling bound b, size r
Output: Solution p

```
1  W := {s_0} (where p_0 ≡ s_0; assert(be^f)) ;
2  while (W ≠ ∅) do
3      s := PickState(W); W := W\{s} ;
4      if (s ∉ F) then
5          W := W ∪ next(s) ;
6      else
7          d := [[s]]^♯ ⊤_D ;
8          if FILTER_D(be^f, d) = ⊥_D then continue ;
9          if FILTER_D(¬be^f, d) = ⊥_D) then
10             choose random h; return s[h];
11         else
12             s' := ExprMutate(s, Γ, V, b, r) ;
13             if s' ≠ ∅ then return s' ;
```

4.4 Putting It All Together

The IMPSynth procedure to solve the imperative synthesis problem (p_0, Γ, V) is given as Algorithm 2. It uses a *workset* W to store states that need to be inspected during the search. The algorithm starts by initializing the workset of states W with the initial state s_0 (line 1), where $p_0 \equiv s_0$; assert(be^f). Next, the PickState(W) procedure picks a state from W with the smallest cost and removes it from W (line 3). To calculate the cost of states, we use a cost model that describes the costs assigned to each construct and operation in the language. In particular, we use the cost model that assigns highest costs to (statement and expression) holes. Thus, the PickState(W) prefers states without holes.

If the picked state s is not final, we add the next states of s to the workset W (line 5), and the while loop is iterated. If the picked state s is final, then we statically analyze it by calling $[[s]]^\sharp \top_{\mathbb{D}}$ (line 7). When s is a failure state we prune it out from the search (line 8), whereas when s is a success state we report any complete program derived from s as final solution (lines 9–10). Otherwise, we call the ExprMutate(s, Γ, V, b, r) procedure to fill in the expression holes in s (line 12). If a solution is found, the algorithm terminates by reporting it (line 13). Otherwise, if ExprMutate terminates without returning a solution, the while is iterated.

Theorem 4.7. *The IMPSynth(p_0, Γ, V, b, r) is b-bounded correct modulo the expression size r and semi-terminating (terminates for solvable problems).*

Proof Sketch. The b-bounded correctness modulo the expression size r follows from the soundness of abstract semantics (Proposition 4.3 and Corollary 4.4) and the b-bounded correctness modulo the expression size r of ExprMutate (Theorem 4.6). The IMPSynth is semi-terminating since it will terminate and find a solution for solvable problems. That is, if there exists a solution, we can find sufficiently large bounds b and r, so that the correct program will be eventually reported. However, if the problem is not solvable (i.e. unrealizable), the IMPSynth may run infinitely by repeatedly picking new states from the workset W, for which the ExprMutate will report that there is no solution. □

5 Evaluation

We now evaluate our approach for imperative program synthesis. The evaluation aims to show that we can use our approach to automatically synthesize various interesting C programs with numerical data types.

Implementation. We have implemented our synthesis algorithm IMPSynth in a prototype tool. The abstract operations and transfer functions of the Polyhedra abstract domain are provided by the APRON library [23], while the ExprMutate procedure is implemented by modifying the CBMC model checker [3] written in C++ as well as the AllRepair [35, 36] tool written in Python. It also calls the MiniCard SAT solver [26] and the Z3 SMT solver [7]. In particular, we have changed the CBMC tool to implement the translation and mutation procedures from the ExprMutate. The altered CBMC (plus ~1K LOC) takes as input a semi-partial program, and produces as output a gsmt2 file containing SMT formulas corresponding to all statements in the input program. In case of a statement with expression holes, it generates SMT formulas for all possible mutations of the corresponding expression hole up to a given size. The altered AllRepair (~2K LOC) reads a gsmt2 file, creates formulas for SAT and SMT solving, and handles all calls to them as well as obtained results. The base of our IMPSynth algorithm is written in OCaml and consists of around 7K LOC. It maintains a workset of partial programs, calls the APRON library for static analysis, and generates semi-partial programs that are handled by the ExprMutate via altered CBMC and AllRepair.

Experiment setup and Benchmarks. All experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU, Xubuntu 22.04 VM, with 8 GB memory, and we use a timeout value of 150 sec. All times are reported as average over five executions. We compare our approach IMPSynth with the known program synthesizer SIMPL [38] that performs static analysis in tandem with enumeration. SIMPL employs heuristics that prefer smaller programs, and directly executes candidates to see if the specification given in the form of input-output examples is met. For both tools, we report Time which is the total time to resolve a given problem. For IMPSynth, in addition we report SMTTime which is the time taken by the ExprMutate procedure to check SMT-based mutations, LOC which is the number of locations in the resulting program, Iter given in the form n/m where n is the number of semi-partial programs extracted from the workset W and m is the number of calls to ExprMutate (note that $m < n$ since

some semi-partial programs are pruned out by static analysis), and Gsmt2 that ís the number of lines of the generated `gsmt2` file that includes all possible mutated formulas in the last call to ExprMutate when solution is found. The default values for unwinding bound is 6 and for abstract domain is Polyhedra. For SIMPL, we report Iter which is the number of explored candidate programs from the workset.

The evaluation is performed on several introductory C numerical problems collected from the literature [25, 38, 39] with simple-to-state logical specifications. In particular, we use two groups of benchmarks: basic problems without loops (`hello`, `abs`, `min`, `max`, `min3`, `max3`, `double10`), and problems with loops (`sum`, `sum_m2n`, `mult`, `power3`, `sum_accum`).

Performance Results. We now present the performance results of our empirical study and discuss the implications. Table 1 shows the performance results of synthesizing our benchmarks. The first group of benchmarks uses the program template: `int res=??; ??`. The problem `hello(int n)`, which is regarded as "Hello World" example of sketching [39], aims to infer a program satisfying the assertion (`res==2*n`) without using the multiplication operator. The correct program shown in Fig. 8 is generated in 0.387 sec in the first iteration. The problem `abs(int n)` wants to synthesize a program that returns the absolute value of the non-zero input `n`, which is specified by the assertion (`res>0` $\wedge$ (`res==n` $\vee$ `res==-n`)). ImpSynth uses 9 iterations and 3 calls to ExprMutate in order to infer the correct program shown in Fig. 9 in 1.315 sec. In contrast, SIMPL generates overfitting solutions for this problem. For input/output examples $1 \rightarrow 1, -2 \rightarrow 2, -5 \rightarrow 5$, it returns the program: `res=n+1; res=res/n; res=res-n` in 0.21 sec and 6617 iterations. By adding more input/output examples (e.g. $4 \rightarrow 4$), it generates overfitting solutions with more assignments (e.g. `res=n+1; res=res/n; res=res-n; n=n+n; r=r%n` in 15.4 sec and 288968 iterations). The problem `min(int n, int m)` should return the minimum of `n` and `m` as specified by the assertion (`res` $\leq$ `n` $\wedge$ `res` $\leq$ `m` $\wedge$ (`res==n` $\vee$ `res==m`)). The correct program given in Fig. 10 is generated after 1.309 sec, 7 iterations, and 3 calls to ExprMutate. The problem `max3(int n1, int n2, int n3)`, which calculates the maximum of three integers `n1`, `n2`, and `n3`, is specified by the assertion (`res` $\geq$ `n1` $\wedge$ `res` $\geq$ `n2` $\wedge$ `res` $\geq$ `n3` $\wedge$ (`res==n1` $\vee$ `res==n2` $\vee$ `res==n3`)). The correct program in Fig. 11 is inferred by performing 23 iterations, 11 calls to ExprMutate (we use only expression of size $l = 1$), and 6.369 sec. On the other hand, SIMPL timeouts for both `min` and `max3` by performing more than 2.6 million iterations. Similarly, we handle `max(int n, int m)` with specification (`res` $\geq$ `n` $\wedge$ `res` $\geq$ `m` $\wedge$ (`res==n` $\vee$ `res==m`)), and `min3(int n1, int n2, int n3)` with specification (`res` $\leq$ `n1` $\wedge$ `res` $\leq$ `n2` $\wedge$ `res` $\leq$ `n3` $\wedge$ (`res==n1` $\vee$ `res==n2` $\vee$ `res==n3`)). Finally, the problem `double10(int n)` with specification (`n<10` $\wedge$ `res==n`) $\vee$ (`n` $\geq$ `10` $\wedge$ `res==2*n`) is solved using 13 iterations and 7 calls to ExprMutate in 3.014 sec by generating the correct program shown in Fig. 12.

The second group of benchmarks use the same program template as `sum` in Fig. 1, which consists of initialization statements followed by an empty `while` loop. The problem `sum_m2n(int m, int n)` takes as input two integers `m` and `n`, and returns the sum $\texttt{m} + (\texttt{m}+1) + \ldots + \texttt{n}$. The specification asserts that the returned value `res` is equal to $\big(\texttt{n} * (\texttt{n}+1) - \texttt{m} * (\texttt{m}-1)\big)/2$. ImpSynth generates the same semi-partial program as for `sum` in Fig. 2, and then successfully synthesizes the correct program given in Fig. 14 in 1.286 sec. Similarly, we solve the problem `mult(int m, int n)` that calculates the product `res` of two integers `m` and `n` without using the multiplication operator, i.e. by using only linear operators + and -. The specification is given by the formula `res==m*n`, and ImpSynth synthesizes the correct program given in Fig. 13 in 1.203 sec. SIMPL generates a trivial one-statement solution without using `while` that is identical to the specification: `res==m*n`, since there is no option to exclude the multiplication operator from the search. The problem `power3(int n)` should return the cube of `n`, which is specified by the formula `res==n*n*n`. The correct program in Fig. 15 is synthesized in 4.110 sec. On the other hand, SIMPL generates a simple solution without using `while`: `t=n*n; res=n*t`, which has potential for overfitting when generalizing the problem to calculate the bigger powers of `n`. The problem `sum_accum(int n)` should return the sum $1 + (1+2) + \ldots (1+2+\ldots+\texttt{n})$, described by the formula $\texttt{n} * (\texttt{n}+1) * (\texttt{n}+2)/6$. ImpSynth successfully adds a nested `while` that is not present in the input template and synthesizes the correct program shown in Fig. 16 in 102.1 sec. In the last iteration, it generates a `gsmt2` file with 859 LOC, there are 7 expression holes, and the total space of mutants is 3072. SIMPL timeouts for `sum_accum` by performing more than 1.7 million iterations.

Discussion. In summary, we can conclude that ImpSynth can successfully synthesize various interesting introductory programs from assertions. SIMPL is usually faster on problems it can successfully resolve (`hello`, `sum`, `sum_m2n`), but fails to synthesize solutions on the other problems (`abs`, `min`, `max`, `min3`, `max3`, `sum_accum`). To check the correctness of proposed candidates, SIMPL directly executes them on the given input-output examples, which is much faster than checking the satisfiability of SMT formulas as done by ImpSynth. However, the two-phase search strategy of ImpSynth, where we first fill in statement holes and then expression holes by incremental SAT and SMT solving, seems to be better suited for programs that have more complex structures, like `if`-s and nested `while`-s. We observe that the ExprMutate procedure takes more than 90% of the total running time of all benchmarks. The performance of ExprMutate depends on the quality of the initial candidate program $p[h^l]$. If $p[h^l]$ is closer to the correct program, then

```
int hello(int n){
  int res = n+n;
  assert(res==2*n);
  return res;
}
```

Figure 8. hello.c.

```
int abs(int n){
  assume(n!=0);
  int res = 0;
  if (res < n)
    res = res+n;
  else res = 0-n;
  assert (res>0∧
   (res==n ∨ res==-n));
  return res; }
```

Figure 9. abs.c.

```
int min(int n, int m){
  int res = n;
  if (m < n)
    res = m;
  assert (res ≤ n ∧ res ≤ m
    ∧(res==n ∨ res==m));
  return res; }
```

Figure 10. min.c.

```
int max3(int n1, int n2, int n3){
  int res = n2;
  if (n3 ≤ n1 ∧ n2 ≤ n1)
    res = n1;
  else if (n2 ≤ n3)
        res = n3;
  assert (res ≥ n1∧res ≥ n2∧res ≥ n3
    ∧(res==n1 ∨ res==n2 ∨ res==n3))
  return res; }
```

Figure 11. max3.c.

```
int double10(int n){
  int res = 0;
  if (10 ≤ n)
    res = res+n;
  res = res+n;
  assert (res ≥ 10 ∧ res=2*n)
    ∨(res < 10 ∧ res==n));
  return res; }
```

Figure 12. double10.c.

```
int mult(int n, int m){
  assume(n > 1);
  assume(m > 1);
  int res = m;
  int i = 1;
  while (i ≤ m) {
    res = res+n;
    i = i+1; }
  assert (res==n*m);
  return res; }
```

Figure 13. mult.c.

```
int sum_mton(int m,int n){
  assume(m ≥ 1);
  assume(n > m);
  int res = 0;
  int i = m;
  while (i ≤ n) {
    res = res+i;
    i = i+1; }
  assert(res==(n*(n+1)-m*(m-1))/2);
  return res; }
```

Figure 14. sum_m2n.c.

```
int power3(int n){
  assume(n > 1);
  int res = 1;
  int i = 1;
  while (i < 4) {
    res = res*n;
    i = i+1; }
  assert (res==n*n*n);
  return res; }
```

Figure 15. power3.c.

```
int sum_accum(int n){
  assume(n > 1);
  int res = 0;
  int i = 1;
  while (i ≤ n) {
    int j = 1;
    while (j ≤ i) {
        res = res+j;
        j = j+1; }
    i = i+1; }
  assert (res==n*(n+1)*(n+2)/6);
  return res; }
```

Figure 16. sum_accum.c.

smaller number of mutations to $p[h^l]$ will be done and the correct program will be located sooner. We now use some heuristics to generate the initial candidate program $p[h^l]$, such as `while`-condition is (x<n), where n is a formal parameter and the variable is changed by x=x+c for some c∈ Γ.

6 Related Work

General program synthesis. Syntax-guided synthesis (SyGuS) [1, 2, 34] provides a logical framework for defining synthesis problems, where the search space is described using a context-free grammar of terms from a given logical theory and the behavioral specification is expressed using a formula in the same theory. The unified logical format stimulated researchers to design various SyGuS solvers [2, 34], employing different techniques ranging from enumerative search to constraint solvers and probabilistic search. However, the main limitation of SyGuS is that the semantics of terms is limited to a fixed decidable SMT theory, such as linear integer arithmetic (LIA) or bitvectors. SyGuS cannot express problems with semantics outside of a supported logical SMT theory, such as imperative programs. On the other hand, we address in ImpSynth the synthesis problem for imperative programs, where we use the standard semantics for assignments, `if`-s, `while`-s, and other constructs (see Fig. 5). To do so, we must deal with "memory store" and no logical theory of the programming language is used.

Type-directed synthesis is used for inferring recursive functional programs [20, 32, 33]. They use type systems to exclude ill-typed functions from the search space.

Table 1. Performance results of IMPSynth vs. SIMPL. All times in sec.

Benchmark	IMPSynth					SIMPL	
	Time	SMTTime	LOC	Iter	Gsmt2	Time	Iter
hello	0.387	0.355	5	1/1	74	0.004	18
abs	1.315	1.160	10	9/3	189	overfit/15.42	288968
min	1.309	1.073	9	7/3	127	timeout	2610000
max	1.110	0.989	9	7/3	127	timeout	2190000
min3	6.059	5.901	13	23/11	221	overfit/0.043	2160
max3	6.369	6.003	13	23/11	221	timeout	2920000
double10	3.041	2.708	10	13/7	205	overfit/0.484	25796
sum.c	1.455	1.383	12	2/1	257	0.005	81
sum_m2n.c	1.286	1.022	13	2/2	294	0.141	4031
mult.c	1.203	0.994	12	2/1	258	0.002	27
power3.c	4.110	3.922	12	2/1	260	0.031	1019
sum_accum.c	102.1	99.2	16	8/3	859	timeout	1860000

Program Repair. Automated program repair is a technique for repairing buggy programs [13, 21, 27, 29, 31, 35, 36], so that the resulting program does not exhibit any faults. There are two main approaches for program repair: testing-based and formal technique-based. Testing-based approaches [21, 27] use test suits as the only specification, so they iteratively generate a candidate from the repair search space and check its validity by running all tests in the test suite against it. Formal technique-based approaches [13, 29, 31, 35, 36] use some formal techniques to guide the repair process, such as symbolic execution, formal verification, deductive synthesis, constraint solving, etc. One such example is the AllRepair tool [13, 35, 36] that uses code mutations, bounded model checking, and SMT solving to generate a correct fix.

Imperative program synthesis. Imperative program synthesis has been a topic of research in recent times. A program sketch is a partial program with some missing holes that should be replaced with one of finitely many integer constants. One of the earliest and widely-known approach to solve the sketching problem is the `Sketch` tool [39, 40], which uses SAT-based counterexample-guided inductive synthesis. FamilySketcher [9, 18] solves the sketching problem by using a lifted (family-based) static analysis based on abstract interpretation. The key idea is that all possible sketch realizations represent a program family, and so the sketch search space is explored via an efficient lifted analysis of program families [16, 17, 19, 22]. This technique can also generate a solution that is optimal with respect to the number of execution steps to termination. The approach [11] uses abstract static analysis and logical abduction to solve the generalized program sketching problem where the missing holes can be replaced with arbitrary expressions, not only with integer constants as in the case of `Sketch` and FamilySketcher. The logical abduction [8] has also been applied for inferring the weakest specifications of unknown functions [15]. Furthermore, sketch-based synthesis has been applied to different problems, such as for inferring programs that use libraries [28], for generating automated feedback [37], etc. Sirastava et al. [41] require users to provide program templates that specify most of the program as input. They treat the synthesis problem as verification problem, and so are limited to the underlying program verifiers.

The closest to our approach for synthesizing imperative programs are SIMPL [38] and SemGuS [24, 25]. SIMPL [38] combines the enumerative search with static analysis, while the semantics-guided synthesis (SemGuS) [24, 25] uses a proof search over Constrained Horn Clauses (CHCs). SemGuS performs better on proving unrealizable problems by showing that no program from a given grammar exists that satisfies the specification, while SIMPL and our approach perform better as synthesizers. However, both tools mainly use input-output examples as specifications, thus suffer from the problem of generating overfitting solutions. In contrast, our approach uses logic formulas as specifications, thus giving stronger guarantees on the correctness of solutions.

7 Conclusion

We presented IMPSynth, an approach for automatic synthesis of imperative programs from logical specifications in the form of assertions. We have shown that combining enumerative search, abstract static analysis, and constraint solvers represents a promising way to address this problem.

In the future, we plan to investigate various techniques to speed-up the performance of IMPSynth. For example, we can adapt the fault localization technique [10, 12] into the ExprMutate procedure in order to prune more aggressively the search space when a buggy mutant is generated.

8 Data-Availability Statement

The tool artifact [14] is available from: https://zenodo.org/records/15587140.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013*. IEEE, USA, 1–8. http://ieeexplore.ieee.org/document/6679385/

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Proceedings, Part I (LNCS, Vol. 10205)*. Springer, Sweden, 319–336. doi:10.1007/978-3-662-54577-5_18

[3] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings (LNCS, Vol. 2988)*. Springer, Spain, 168–176. doi:10.1007/978-3-540-24730-2_15

[4] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Record of the Fourth ACM Symposium on POPL*. ACM, USA, 238–252. doi:10.1145/512950.512973

[5] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on POPL'78*. ACM Press, USA, 84–96. doi:10.1145/512760.512770

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. doi:10.1145/115372.115320

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Hungary, 337–340. doi:10.1007/978-3-540-78800-3_24

[8] Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *Computer Aided Verification - 25th International Conference, CAV 2013. Proceedings (LNCS, Vol. 8044)*. Springer, Russia, 684–689. doi:10.1007/978-3-642-39799-8_46

[9] Aleksandar S. Dimovski. 2022. Quantitative Program Sketching using Lifted Static Analysis. In *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Proceedings (LNCS, Vol. 13241)*. Springer, Germany, 102–122. doi:10.1007/978-3-030-99429-7_6

[10] Aleksandar S. Dimovski. 2023. Error Invariants for Fault Localization via Abstract Interpretation. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings (LNCS, Vol. 14284)*. Springer, Portugal, 190–211. doi:10.1007/978-3-031-44245-2_10

[11] Aleksandar S. Dimovski. 2023. Generalized Program Sketching by Abstract Interpretation and Logical Abduction. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings (LNCS, Vol. 14284)*. Springer, Portugal, 212–230. doi:10.1007/978-3-031-44245-2_11

[12] Aleksandar S. Dimovski. 2024. Fault localization by abstract interpretation and its applications. *J. Comput. Lang.* 80 (2024), 101288. doi:10.1016/J.COLA.2024.101288

[13] Aleksandar S. Dimovski. 2024. Mutation-Based Lifted Repair of Software Product Lines. In *38th European Conference on Object-Oriented Programming, ECOOP 2024 (LIPIcs, Vol. 313)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Austria, 12:1–12:24. doi:10.4230/LIPICS.ECOOP.2024.12

[14] Aleksandar S. Dimovski. 2024. Tool Artifact for "Imperative Program Synthesis by Abstract Static Analysis and SMT Mutations". *Zenodo* (2024). doi:10.5281/zenodo.15587140

[15] Aleksandar S. Dimovski. 2025. Weakest Safe Context Synthesis by Symbolic Game Semantics and Logical Abduction. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025*. ACM, Italy, 1990–1997. doi:10.1145/3672608.3707849

[16] Aleksandar S. Dimovski and Sven Apel. 2021. Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation. In *35th European Conference on Object-Oriented Programming, ECOOP 2021 (LIPIcs, Vol. 194)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Denmark, 14:1–14:28. doi:10.4230/LIPIcs.ECOOP.2021.14

[17] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2021. A Decision Tree Lifted Domain for Analyzing Program Families with Numerical Features. In *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings (LNCS, Vol. 12649)*. Springer, Luxembourg, 67–86. https://arxiv.org/abs/2012.05863

[18] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2021. Program Sketching using Lifted Analysis for Numerical Program Families. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings (LNCS, Vol. 12673)*. Springer, USA, 95–112. doi:10.1007/978-3-030-76384-8_7

[19] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2016. Finding Suitable Variability Abstractions for Family-Based Analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings (LNCS, Vol. 9995)*. Springer, Cyprus, 217–234. doi:10.1007/978-3-319-48989-6_14

[20] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, USA, 802–815. doi:10.1145/2837614.2837629

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[22] Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2017. Effective Analysis of C Programs by Rewriting Variability. *Programming Journal* 1, 1 (2017), 1. doi:10.22152/programming-journal.org/2017/1/1

[23] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings (LNCS, Vol. 5643)*. Springer, France, 661–667. doi:10.1007/978-3-642-02658-4_52

[24] Keith J. C. Johnson, Andrew Reynolds, Thomas W. Reps, and Loris D'Antoni. 2024. The SemGuS Toolkit. In *Computer Aided Verification - 36th International Conference, CAV 2024, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 14683)*. Springer, Canada, 27–40. doi:10.1007/978-3-031-65633-0_2

[25] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. doi:10.1145/3434311

[26] Mark H. Liffiton and Jordyn C. Maglalang. 2012. A Cardinality Solver: More Expressive Constraints for Free - (Poster Presentation). In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Proceedings (LNCS, Vol. 7317)*. Springer, Italy, 485–486. doi:10.1007/978-3-642-31612-8_47

[27] Fan Long and Martin C. Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on ESEC/FSE 2015*. ACM, Italy, 166–178. doi:10.1145/2786805.2786811

[28] Benjamin Mariano, Josh Reese, Siyuan Xu, ThanhVu Nguyen, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2019. Program synthesis with algebraic library specifications. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 132:1–132:25. doi:10.1145/3360558

[29] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. ACM, USA, 691–701. doi:10.1145/2884781.2884807

[30] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372. doi:10.1561/2500000034

[31] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13*. IEEE Computer Society, USA, 772–781. doi:10.1109/ICSE.2013.6606623

[32] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*. ACM, USA, 619–630. doi:10.1145/2737924.2738007

[33] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. ACM, USA, 522–538. doi:10.1145/2908080.2908093

[34] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II (LNCS, Vol. 9207)*. Springer, USA, 198–216. doi:10.1007/978-3-319-21668-3_12

[35] Bat-Chen Rothenberg and Orna Grumberg. 2016. Sound and Complete Mutation-Based Program Repair. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings (LNCS, Vol. 9995)*. Springer, Cyprus, 593–611. doi:10.1007/978-3-319-48989-6_36

[36] Bat-Chen Rothenberg and Orna Grumberg. 2020. Must Fault Localization for Program Repair. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II (LNCS, Vol. 12225)*. Springer, USA, 658–680. doi:10.1007/978-3-030-53291-8_33

[37] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*. ACM, USA, 15–26. doi:10.1145/2491956.2462195

[38] Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis - 24th International Symposium, SAS 2017, Proceedings (LNCS, Vol. 10422)*. Springer, USA, 364–381. doi:10.1007/978-3-319-66706-5_18

[39] Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495. doi:10.1007/s10009-012-0249-7

[40] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM, USA, 281–294. doi:10.1145/1065010.1065045

[41] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2013. Template-based program verification and program synthesis. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 497–518. doi:10.1007/s10009-012-0223-4

[42] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Static Analysis - 21st International Symposium, SAS 2014. Proceedings (LNCS, Vol. 8723)*. Springer, Germany, 302–318. doi:10.1007/978-3-319-10936-7_19

Received 2025-04-10; accepted 2025-05-19