Preprocessing of the 20th Conference on Computer
Science and Intelligence Systems (FedCSIS) pp. 517–528

While-guard Synthesis by Abstract Static Analysis and CHC Solving

Aleksandar S. Dimovski
0000-0002-3601-2631
Mother Teresa University
Skopje, North Macedonia
aleksandar.dimovski@unt.edu.mk

***Abstract*—This paper introduces a novel technique for automatically synthesizing assertion-safe `while`-guards in imperative programs. Given a partial program (sketch) with missing `while`-guards, the proposed algorithm synthesizes complete Boolean expressions for the missing ones, such that the obtained complete program satisfies the given assertions. To solve this problem, our technique uses forward and backward abstract static analyses of programs to generate (logical) constraints with unknown predicates that are subsequently solved by employing the logical Constraint Horn Clause (CHC) solvers.**

We have implemented our synthesis algorithm in a proof-of-concept tool, and evaluated it on a set of C programs with missing `while`-guards. By experiments we prove the effectiveness of the proposed technique for synthesizing arbitrary Boolean expressions defined over program variables for some interesting program sketches written in C.

***Index Terms*—Program Synthesis, Abstract Static Analysis, Constraint Horn Clause (CHC) solving.**

I. Introduction

PROGRAM synthesis [1] is a process of automatically constructing a program from a given specification. Program sketches [2], [3] are partial programs with missing (arithmetic and Boolean) expressions called *holes* (`??`). They simplify the synthesis problem by limiting the search to programs of particular form. The earliest approaches to program sketching [2], [3], [4] can automatically synthesize only integer "constants" for the holes. In particular, given a sketch and an assertion, the synthesizer (e.g., `Sketcher` [2], [3]) automatically finds integer constants for the holes such that the synthesized complete program satisfies the assertion. Subsequently, the generalized program sketcher [5] is introduced, which uses abstract interpretation [6], [7], [8] and logical abduction [9] to infer arbitrary expressions, rather than only integer constants, for the holes in program sketches. The missing holes occur either as guards (Boolean expressions) in `if`-s and `while`-s or as right-hand sides (arithmetic expressions) in assignments. However, in the case of missing `while`-guards, this approach [5] is cumbersome due to the difficulty to infer precise inductive invariants for `while`-s, and so it often fails to generate satisfactory solutions.

In this paper, we explore the problem of inferring missing `while`-guards in program sketches by introducing a novel algorithm based on abstract static analysis of programs [6], [7], [8] and logical Constraint Horn Clause (CHC) solving [10], [11]. In particular, our synthesis algorithm consists of two phases: *constraint generation* and *constraint solving*. In *the first phase*, we employ forward and backward abstract static analyses of programs, whose aim is to determine dynamic program properties in a fully automatic manner. The forward abstract analysis [6], [7], which is similar to strongest postcondition inference [12], works by propagating the initial facts towards the final locations, thus computing the invariant postconditions before the holes. The backward abstract analysis [6], [7], which is similar to weakest precondition inference [12], works by propagating the final (assertion) facts towards the initial locations, thus computing the sufficient preconditions that ensure assertion validity after the holes. The numerical abstract domains, such as Intervals [6], Octagons [13], Polyhedra [14], are widely used in practice for static analyses to infer information about the possible values of program variables together with the possible relations between them. Thus, they are employed for computing invariant postconditions and sufficient preconditions in the form of linear integer constraints over program variables. Subsequently, in *the second phase*, we employ the logical constraints generated in the first phase by abstract static analyses as well as the logical CHC solving technique to synthesize the missing Boolean expressions. That is, the `while`-guard synthesis problem is encoded as a set of CHCs, which represent a class of first-order logical formulas modulo some background theories, like linear integer arithmetic (LIA). More specifically, a CHC is a logical implication formula containing unknown predicates. This way, all holes in the sketch as well as inductive invariants for the `while`-s are represented by unknown predicates in the CHCs. A solution to a set of CHCs assigns a logical formula to each unknown predicate in such a way that all CHCs become valid formulas, and moreover their validity guarantees the correctness of the resulting complete program. By solving the obtained CHCs with off-the-shelf CHC solvers, such as `Eldarica` [11], `Spacer` [10], `Duality` [15], and `FreqHorn` [16], we can synthesize complete Boolean expressions for the missing holes `??` in the program sketch, which will guarantee the assertion validity of the resulting complete program.

We have implemented the proposed algorithm in a proof-of-concept tool, called $\texttt{WhileSketching}_{\texttt{CHC}}$. For the constraint generation phase, we use the numerical abstract domains

```
void main() {
① int x = 10, y = 0;
② while ⓗ (??) do {
③   x = x-1;
④   y = y+1;}
⑤ assert (y==10);
//assert (y>x);
//assert (y==x);
//assert (x=10); }
//assert (x>15); }
```

Fig. 1. `while.c`.

(e.g., Polyhedra [14]) from the APRON library [17], while for the constraint solving phase, we use the `Eldarica` [11] and `Z3` [18] tools for CHC- and SMT-solving. We perform an experimental evaluation on a selected set of benchmarks written in C, which are taken from the SV-COMP (https://sv-comp.sosy-lab.org/) and the literature [19], [5]. The experiments demonstrate the practicality of our technique to successfully synthesize assertion-safe `while`-guards in some interesting C sketches.

This work brings the following contributions:

(1) We present the `while`-guard synthesis problem for automatically synthesizing arbitrary complete Boolean expressions for missing holes in sketches;
(2) We introduce a novel algorithm for solving this problem by interaction between the abstract static analysis and the logical CHC solving;
(3) We implement our algorithm and evaluate it on a set of interesting program sketches written in C.

This work is organized in the following way. Section II presents a small example illustrating how our algorithm infers missing `while`-guards. Section III defines a small imperative language that is used for the formal development as well as its concrete and abstract semantics. Moreover, the CHC problem is also defined here. Section IV presents the main `while`-guard synthesis algorithm. Section V briefly describes another existing approach for resolving program sketches based on logical abduction. Section VI presents an experimental evaluation of the introduced approach, comparing our tool with a program sketching tool based on logical abduction. Finally, Section VII and VIII discuss related work, summarize the introduced approach, and highlight future research direction.

II. Illustration

We now illustrate our approach through one motivating example taken from [5]. The program sketch given in Fig. 1, called `while.c`, contains one unknown hole `??` in the `while`-guard. Our aim is to replace the unknown hole at loc. ⓗ with a Boolean expression, such that the assertion at loc ⑤ is valid.

Our approach consists of two phases: constraint generation and constraint solving. In the first phase, our technique starts by performing forward and backward abstract static analyses based on the Polyhedra abstract domain [14]. The over-approximating forward static analysis starts with the initial fact $\top$ (i.e., true) at loc. ①, and by propagating it forwards it infers the invariants $(\texttt{x=10} \wedge \texttt{y=0})$ at loc. ② before the `while` as well as $(\texttt{x} \leq 10 \wedge \texttt{x+y=10})$ at loc. ⓗ. The under-approximating backward static analysis starts with the assertion fact $(\texttt{y=10})$ at loc. ⑤, and by propagating it backwards it infers the preconditions $(\texttt{y=10})$ at loc. ⑤ after the `while` as well as $\bot$ (i.e., false) at loc. ⓗ. The loc. ⓗ is reached upon entering the `while`-loop and after each `while`-iteration, so the facts inferred by both analyses at loc. ⓗ correspond to the *inductive `while`-invariant*, which expresses a relationship between program variables that is preserved by any execution of the `while`'s body [12]. However, since both analyses treat the unknown hole `??` as non-deterministic choice $\{\text{true}, \text{false}\}$, the forward analysis infers the over-approximation of the inductive `while`-invariant, whereas the backward analysis infers the under-approximation of the inductive `while`-invariant. Thus, we use the inferred facts by forward and backward static analyses to generate the following implication constraints, called Constrained Horn Clauses (CHCs):

$$(\texttt{x=10} \wedge \texttt{y=0}) \implies Inv(\texttt{x},\texttt{y}) \quad (1)$$

$$Inv(\texttt{x},\texttt{y}) \wedge R(\texttt{x},\texttt{y}) \wedge \texttt{x'=x-1} \wedge \texttt{y'=y+1} \implies Inv(\texttt{x'},\texttt{y'}) \quad (2)$$

$$Inv(\texttt{x},\texttt{y}) \wedge \neg R(\texttt{x},\texttt{y}) \implies (\texttt{y=10}) \quad (3)$$

$$Inv(\texttt{x},\texttt{y}) \implies (\texttt{x} \leq 10 \wedge \texttt{x+y=10}) \quad (4)$$

$$\text{false} \implies Inv(\texttt{x},\texttt{y}) \quad (5)$$

$Inv(\texttt{x},\texttt{y})$ encodes an unknown predicate for the inductive `while`-invariant, and $R(\texttt{x},\texttt{y})$ encodes an unknown predicate for the missing `while`-guard `??`. Both unknown predicates $Inv(\texttt{x},\texttt{y})$ and $R(\texttt{x},\texttt{y})$ are defined over program variables `x` and `y` that are in the scope of the `while`. The implications (1) and (2) state that the `while`-invariant $Inv(\texttt{x},\texttt{y})$ is inductive. That is, (1) states that $Inv(\texttt{x},\texttt{y})$ is implied by the `while`'s precondition at loc. ②, whereas (2) states that $Inv(\texttt{x},\texttt{y})$ is preserved in each iteration of the `while`'s body, i.e. if $Inv(\texttt{x},\texttt{y})$ and the `while`-guard $R(\texttt{x},\texttt{y})$ are true and we execute the `while`'s body then $Inv(\texttt{x'},\texttt{y'})$ will be true after it, where `x',y'` are updated values of `x,y` after executing the `while`'s body. The implication (3) stipulates that when the `while` terminates, and so $Inv(\texttt{x},\texttt{y})$ and $\neg R(\texttt{x},\texttt{y})$ are true, then they are strong enough to imply the `while`'s postcondition at loc. ⑤ ensuring that the assertion must be valid. The implication (4) (resp., (5)) states that the analysis fact inferred at loc. ⓗ by forward (resp., backward) analysis is over- (resp., under-) approximation of $Inv(\texttt{x},\texttt{y})$.

In the second phase, our aim is to solve the generated CHCs by inferring interpretations (logical formulas) for $Inv(\texttt{x},\texttt{y})$ and $R(\texttt{x},\texttt{y})$ under which all CHCs (1) – (5) are valid. The CHC solving algorithm [11] reports the following solutions for the unknown predicates: $Inv(\texttt{x},\texttt{y}) \equiv (\texttt{x} \leq 10 \wedge \texttt{x+y=10})$ and $R(\texttt{x},\texttt{y}) \equiv (\texttt{x} \leq 10 \wedge \texttt{x+y=10} \wedge \texttt{x} \neq 0)$. Then, we call a SMT solver to simplify the formula $R(\texttt{x},\texttt{y}) \wedge \neg Inv(\texttt{x},\texttt{y})$, thus excluding sub-formulas of $R(\texttt{x},\texttt{y})$ that are also present in the invariant $Inv(\texttt{x},\texttt{y})$. Thus, the synthesized `while`-guard

will be only the strengthening of $R(\mathtt{x},\mathtt{y})$ with respect to $Inv(\mathtt{x},\mathtt{y})$ that is not part of the invariant $Inv(\mathtt{x},\mathtt{y})$. Hence, we fill the hole `??` at loc. ⓗ with the solution $(\mathtt{x} \neq 0)$.

In a similar vein, we can synthesize concrete `while`-guards in `while.c` (Fig. 1) with respect to different assertions. Assume that we consider `assert`(be). The CHCs (1), (2), (4), (5) stay the same as above, and we only change the right-hand side of CHC (3) that becomes: $Inv(\mathtt{x},\mathtt{y}) \wedge \neg R(\mathtt{x},\mathtt{y}) \implies (be)$. For example, for assertions $(\mathtt{y} > \mathtt{x})$, $(\mathtt{y}{=}\mathtt{x})$, $(\mathtt{x}{=}10)$, and $(\mathtt{x} > 15)$, the synthesized `while`-guards are: $(\mathtt{x} \geq 5)$, $(\mathtt{x} \neq 5)$, (false), and the problem is unrealizable (i.e., there is no solution that can be derived), respectively. Note that for `assert`$(\mathtt{x} > 15)$, we obtain the same solutions for $Inv(\mathtt{x},\mathtt{y})$ and $R(\mathtt{x},\mathtt{y})$, that is $Inv(\mathtt{x},\mathtt{y}) \equiv R(\mathtt{x},\mathtt{y}) \equiv (\mathtt{x} \leq 10 \wedge \mathtt{x}{+}\mathtt{y}{=}10)$. For example, the implication (3): $Inv(\mathtt{x},\mathtt{y}) \wedge \neg R(\mathtt{x},\mathtt{y}) \implies (\mathtt{x} > 15)$ becomes vacuously valid in this case as the left-hand side of the above implication is (false). However, $(R(\mathtt{x},\mathtt{y}) \wedge \neg Inv(\mathtt{x},\mathtt{y})) \equiv$ false, so we conclude that there is no (non-vacuous) solution for this assertion (i.e., the problem is unrealizable). On the other hand, for `assert`$(\mathtt{x}{=}10)$, we obtain the CHC solution $Inv(\mathtt{x},\mathtt{y}) \equiv (\mathtt{x} \leq 10 \wedge \mathtt{x}{+}\mathtt{y}{=}10)$ and $R(\mathtt{x},\mathtt{y}) \equiv$ (false), so we infer the solution (false) for `??` in this case.

III. Background

This section introduces the language we consider as well as its concrete and abstract semantics. Furthermore, we also formally define the CHC problem.

A. 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. We use the hole constructs $\mathtt{??}_i$ to encode unknown `while`-guards. The holes $\mathtt{??}_i$ are placeholders that the synthesizer must replace with suitable Boolean expressions, such that the resulting complete program will satisfy all assertions. The syntax of the language is:

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

where n ranges over integers $\mathbb{Z}$, $\mathtt{x}$ ranges over program variables *Var*, $[n, n']$ over integer intervals, and $\oplus \in \{+, -, *, /\}$, $\bowtie \in \{<, \leq, =, \neq\}$. Integer intervals $[n, n']$ denote a random choice of an integer in the interval. We assume that statements are tagged with unique syntactic labels $l \in \mathbb{L}$. Note that `while` has an extra syntactic label $l_h \in \mathbb{L}$ at the `while`-head, i.e. between the `while` keyword and the guard. We assume, without loss of generality, that a program p is a sequence of statements followed by a single assertion "$l_i : s; l_f : \mathtt{assert}(be^f)$". We say that the program p satisfies its assertion (or the assertion is valid) if Boolean expression be^f evaluates to (true) whenever the control reaches the label l_f. Otherwise, the assertion is violated. We say that a program (statement) is *complete* if it contains no holes; otherwise, we say that it is *partial* (or *program sketch*).

$$
\begin{array}{l}
\overrightarrow{[\![\mathtt{skip}]\!]} S = S \\
\overrightarrow{[\![\mathtt{x} = ae]\!]} S = \{\sigma[\mathtt{x} \mapsto n] \mid \sigma \in S, n \in [\![ae]\!]\sigma\} \\
\overrightarrow{[\![s_1\ ;\ s_2]\!]} S = \overrightarrow{[\![s_2]\!]}(\overrightarrow{[\![s_1]\!]} S) \\
\overrightarrow{[\![\mathtt{assume}(be)]\!]} S = \{\sigma \in S \mid \mathsf{true} \in [\![be]\!]\sigma\} \\
\overrightarrow{[\![\mathtt{if}\ (be)\ \ s_1\ \mathtt{else}\ s_2]\!]} S = \overrightarrow{[\![s_1]\!]}\{\sigma \in S \mid \mathsf{true} \in [\![be]\!]\sigma\} \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad \overrightarrow{[\![s_2]\!]}\{\sigma \in S \mid \mathsf{false} \in [\![be]\!]\sigma\} \\
\overrightarrow{[\![\mathtt{while}\ (be)\ \mathtt{do}\ s]\!]} S = \{\sigma \in \mathtt{lfp}\ \overrightarrow{\phi_S} \mid \mathsf{false} \in [\![be]\!]\sigma\} \\
\overrightarrow{\phi_S}(X) = S \cup \overrightarrow{[\![s]\!]}\{\sigma \in X \mid \mathsf{true} \in [\![be]\!]\sigma\}
\end{array}
$$

Fig. 2. Definitions of $\overrightarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$.

Remark. Note that in programs p we allow assumptions (`assume`(be)) to freely occur in a sequence of statements s, while the assertion (`assert`(be)) occurs only at the final location l_f. An assertion `assert`(be) specifies that every execution reaching it is expected to evaluate be to true, in which case we say that the assertion is valid. Otherwise, if the assertion is not valid, the program terminates early with an error. On the other hand, an assumption `assume`(be) specifies that every execution reaching it with be evaluated to false is terminated. However, unlike before, this termination is not erroneous, but it means that the programmer does not want to consider the rest of this execution when verifying the correctness of the program.

Let H be a set of uniquely labelled holes $\mathtt{??}_i$ in program p. A *control function* ϕ is a mapping from the set of holes H to complete Boolean expressions be'. We say that ϕ is *complete* if $dom(\phi) = H$, i.e. ϕ is defined for each hole in the program. Otherwise, if $dom(\phi) \subset H$, i.e. ϕ is $\perp$ (undefined) for some holes, we say that ϕ is a *partial* control function. We write $p[\phi]$ to denote the program obtained by substituting each $\mathtt{??}_i$ with $\phi(\mathtt{??}_i)$, if $\phi(\mathtt{??}_i)$ is defined.

Definition 3.1: A complete control function ϕ is a *solution* to the `while`-guard synthesis problem defined by program sketch p if $p[\phi]$ is a complete program that satisfies its assertion.

B. Concrete Semantics and Analyses

The concrete semantics describes the properties of the possible executions of a program. We now define the concrete semantics of our language, and use it to construct *reachability* or *invariance* (forward) as well as *sufficient condition* (backward) *concrete analyses*.

A memory *store*, denoted $\sigma \in \Sigma$, is a function mapping each program variable to an integer value: $\Sigma = \mathit{Var} \to \mathbb{Z}$. The concrete domain is the complete lattice of the powerset of stores [1] $\langle \mathcal{P}(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma \rangle$. The semantics of arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$ and Boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\mathsf{true}, \mathsf{false}\})$ are the sets of possible (numerical

[1] The powerset of a set S, denoted $\mathcal{P}(S)$, is the set of all subsets of S.

$$
\begin{array}{l}
\overleftarrow{[\![\texttt{skip}]\!]}S = S \\
\overleftarrow{[\![\texttt{x = } ae]\!]}S = \{\sigma \mid \forall n \in [\![ae]\!]\sigma, \sigma[\texttt{x} \mapsto n] \in S\} \\
\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]}S = \overleftarrow{[\![s_1]\!]}(\overleftarrow{[\![s_2]\!]}S) \\
\overleftarrow{[\![\texttt{assume}(be)]\!]}S = S \cup \{\sigma \mid [\![be]\!]\sigma = \{\text{false}\}\} \\
\overleftarrow{[\![\texttt{if } (be) \ \ s_1 \texttt{ else } s_2]\!]}S = \big(\overleftarrow{[\![s_1]\!]}S \cup \{\sigma \mid [\![be]\!]\sigma = \{\text{false}\}\}\big) \cap \\
\qquad\qquad \big(\overleftarrow{[\![s_2]\!]}S \cup \{\sigma \mid [\![\neg be]\!]\sigma = \{\text{false}\}\}\big) \\
\overleftarrow{[\![\texttt{while } (be) \texttt{ do } s]\!]}S = \texttt{gfp}\ \overleftarrow{\phi_S} \\
\overleftarrow{\phi_S}(X) = \big(S \cup \{\sigma \mid [\![\neg be]\!]\sigma = \{\text{false}\}\}\big) \cap \big(\overleftarrow{[\![s]\!]}X \cup \{\sigma \mid [\![be]\!]\sigma = \{\text{false}\}\}\big)
\end{array}
$$

Fig. 3. Definitions of $\overleftarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$.

and Boolean) values for expressions ae and be in a given store σ. For example, $[\![n]\!]\sigma = n$ and $[\![[n, n']]\!]\sigma = \{n, \ldots, n'\}$ for arithmetic expressions n and $[n, n']$, whereas $[\![\texttt{??}_i]\!]\sigma = \{\text{true}, \text{false}\}$ for Boolean expression $\texttt{??}_i$. A semantics definition associates with each statement (program) a concrete semantics property from $\mathcal{P}(\Sigma)$ representing some characteristics of its possible executions. We consider two semantics of statements (programs): a *reachability or invariance* (forward) semantics $\overrightarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ that infers a set of reachable stores (invariants) from a given set of initial stores; and a *sufficient condition* (backward) semantics $\overleftarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ that infers a set of stores (sufficient condition) from which only stores satisfying a given postcondition are reached. The definitions of $\overrightarrow{[\![s]\!]}$ and $\overleftarrow{[\![s]\!]}$ are given in Fig. 2 and Fig. 3, respectively. The reachability semantics [6] is built forward, so each function $\overrightarrow{[\![s]\!]}$ takes as input a set of stores S before statement s and returns a set of possible stores reached after executing s from S. The sufficient condition semantics [7] is built backward, so each function $\overleftarrow{[\![s]\!]}$ takes as input a set of stores S after statement s and returns a set of possible stores before s from which only stores from S are reached after executing s. For example, given a set of stores $S \in \mathcal{P}(\Sigma)$, we have that $\overrightarrow{[\![\texttt{x = } ae]\!]}S$ [2] is the set of final stores reachable after executing $\texttt{x = } ae$, whereas $\overleftarrow{[\![\texttt{x = } ae]\!]}S$ is the set of initial stores such that any execution of $\texttt{x = } ae$ from them leads to a store in S. The semantics of a `while` statement is given in a standard fixed-point formulation [6], [7] using the least and greatest fix-point operators `lfp` and `gfp`, where the fixed-point functionals $\overrightarrow{\phi_S}, \overleftarrow{\phi_S} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, parameterized with the input set of stores S, accumulate possible stores after another `while` iteration from a set of stores X going in a forward and backward direction, respectively.

C. Abstract Semantics

The language we consider is Turing complete, thus making the concrete semantics and analyses uncomputable. We now define computable abstract analyses [6], [7], [8] that are approximations of the concrete semantics and analyses. The first basic choice in abstract interpretation is to design an abstract semantics domain $\mathbb{D}$, which is an approximate version of the concrete semantics domain $\mathcal{P}(\Sigma)$. More specifically, 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 $\langle \mathcal{P}(\Sigma), \subseteq \rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$. [3] This abstraction works only with a concretization function $\gamma_{\mathbb{D}} : \mathbb{D} \to \mathcal{P}(\Sigma)$ expressing 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 forward and backward analyses. In particular, they have 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 forward transfer functions for assignments $\texttt{ASSIGN}_{\mathbb{D}} : \textit{Stm} \times \mathbb{D} \to \mathbb{D}$ and tests $\texttt{FILTER}_{\mathbb{D}} : \textit{BExp} \times \mathbb{D} \to \mathbb{D}$, which are sound over-approximations of the corresponding concrete functions. We let $\texttt{lfp}^{\#}$ denote an abstract fix-point operator, derived using widening $\triangledown_{\mathbb{D}}$ and narrowing $\triangle_{\mathbb{D}}$, that over-approximates the concrete `lfp`. There are also backward transfer functions for assignments $\texttt{B-ASSIGN}^u_{\mathbb{D}} : \textit{Stm} \times \mathbb{D} \to \mathbb{D}$, tests $\texttt{B-FILTER}^u_{\mathbb{D}} : \textit{BExp} \times \mathbb{D} \to \mathbb{D}$, and a lower widening $\underline{\triangledown}_{\mathbb{D}}$ [7], which are sound under-approximations of the corresponding concrete functions. We let $\texttt{gfp}^{\#}$ denote an abstract fixpoint operator, derived using lower widening $\underline{\triangledown}_{\mathbb{D}}$, that under-approximates the concrete `gfp`.

The abstract domain $\mathbb{D}$ can be employed to derive abstract reachability (resp., abstract sufficient condition) analysis that represents over-approximation (resp., under-approximation) of the corresponding concrete analysis. For each statement s, we define the abstract reachability semantics $\overrightarrow{[\![s]\!]}^{\sharp}$ and the abstract sufficient condition semantics $\overleftarrow{[\![s]\!]}^{\sharp}$ in Fig. 4 and Fig. 5, respectively. For a `while` loop, $\texttt{lfp}^{\sharp}\ \overrightarrow{\phi}^{\sharp}_d$ is the limit of the following increasing chain: $y_0 = d$, $y_{n+1} = y_n \triangledown_{\mathbb{D}} \overrightarrow{\phi}^{\sharp}_d(y_n)$ for forward analysis; whereas $\texttt{gfp}^{\sharp}\ \overleftarrow{\phi}^{\sharp}_d$ is the limit of the following decreasing chain: $y_0 = \texttt{B-FILTER}^u_{\mathbb{D}}(\neg be, d)$, $y_{n+1} = y_n \underline{\triangledown}_{\mathbb{D}} \overleftarrow{\phi}^{\sharp}_d(y_n)$ for backward analysis. Since $\texttt{FILTER}_{\mathbb{D}}(be, d) \sqsubseteq_{\mathbb{D}} d$ and $\texttt{B-FILTER}^u_{\mathbb{D}}(be, d) \sqsupseteq_{\mathbb{D}} d$, we use $\texttt{FILTER}_{\mathbb{D}}(\texttt{??}_i, d) = d$ and $\texttt{B-FILTER}^u_{\mathbb{D}}(\texttt{??}_i, d) = d$ to handle the Boolean expression hole $\texttt{??}_i$.

By the soundness of the operators of abstract domain $\mathbb{D}$ [6], [7], we prove the soundness of abstract semantics with respect to concrete semantics. That is, $\overrightarrow{[\![s]\!]}^{\sharp}$ over-approximates the set of reachable stores obtained by concrete semantics $\overrightarrow{[\![s]\!]}$, whereas $\overleftarrow{[\![s]\!]}^{\sharp}$ under-approximates the set of sufficient condition stores obtained by concrete semantics $\overleftarrow{[\![s]\!]}$.

Proposition 3.2 ([6], [7]): We have: $\overrightarrow{[\![s]\!]}\gamma_{\mathbb{D}}(d) \subseteq \overrightarrow{[\![s]\!]}^{\sharp}d$ and $\overleftarrow{[\![s]\!]}\gamma_{\mathbb{D}}(d) \supseteq \overleftarrow{[\![s]\!]}^{\sharp}d$.

[2] $\sigma[\texttt{x} \mapsto n]$ denotes the store that updates σ at variable `x` to be equal to n.

[3] Concretization-based abstraction, which uses only a concretization function $\gamma_{\mathbb{D}}$ (e.g. Polyhedra domain), represents a relaxation of the well-known Galois connection that uses both abstraction and concretization functions.

$$\overrightarrow{[\![\texttt{skip}]\!]^\sharp} d = d$$
$$\overrightarrow{[\![\texttt{x = } ae]\!]^\sharp} d = \texttt{ASSIGN}_{\mathbb{D}}(\texttt{x = } ae, d)$$
$$\overrightarrow{[\![s_1 \texttt{ ; } s_2]\!]^\sharp} d = \overrightarrow{[\![s_2]\!]^\sharp}(\overrightarrow{[\![s_1]\!]^\sharp} d)$$
$$\overrightarrow{[\![\texttt{assume}(be)]\!]^\sharp} d = \texttt{FILTER}_{\mathbb{D}}(be, d)$$
$$\overrightarrow{[\![\texttt{if } be \texttt{ } s_1 \texttt{ else } s_2]\!]^\sharp} d = \overrightarrow{[\![s_1]\!]^\sharp}(\texttt{FILTER}_{\mathbb{D}}(be, d)) \sqcup_{\mathbb{D}} \overrightarrow{[\![s_2]\!]^\sharp}(\texttt{FILTER}_{\mathbb{D}}(\neg be, d))$$
$$\overrightarrow{[\![\texttt{while } be \texttt{ do } s]\!]^\sharp} d = \texttt{FILTER}_{\mathbb{D}}(\neg be, \texttt{lfp}^\sharp\ \overrightarrow{\phi}^\sharp_d)$$
$$\overrightarrow{\phi}^\sharp_d(x) = d \sqcup_{\mathbb{D}} \overrightarrow{[\![s]\!]^\sharp}(\texttt{FILTER}_{\mathbb{D}}(be, x))$$

Fig. 4. Definitions of $\overrightarrow{[\![s]\!]^\sharp} : \mathbb{D} \to \mathbb{D}$.

$$\overleftarrow{[\![\texttt{skip}]\!]^\sharp} d = d$$
$$\overleftarrow{[\![\texttt{x = } ae]\!]^\sharp} d = \texttt{B-ASSIGN}^u_{\mathbb{D}}(\texttt{x = } ae, d)$$
$$\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]^\sharp} d = \overleftarrow{[\![s_1]\!]^\sharp}(\overleftarrow{[\![s_2]\!]^\sharp} d)$$
$$\overleftarrow{[\![\texttt{assume}(be)]\!]^\sharp} d = \texttt{B-FILTER}^u_{\mathbb{D}}(be, d)$$
$$\overleftarrow{[\![\texttt{if } be \texttt{ } s_1 \texttt{ else } s_2]\!]^\sharp} d = \texttt{B-FILTER}^u_{\mathbb{D}}(be, \overleftarrow{[\![s_1]\!]^\sharp} d) \sqcap_{\mathbb{D}} \texttt{B-FILTER}^u_{\mathbb{D}}(\neg be, \overleftarrow{[\![s_2]\!]^\sharp} d)$$
$$\overleftarrow{[\![\texttt{while } be \texttt{ do } s]\!]^\sharp} d = \texttt{gfp}^\sharp\ \overleftarrow{\phi}^\sharp_d$$
$$\overleftarrow{\phi}^\sharp_d(x) = \texttt{B-FILTER}^u_{\mathbb{D}}(\neg be, d) \sqcap_{\mathbb{D}} \texttt{B-FILTER}^u_{\mathbb{D}}(be, \overleftarrow{[\![s]\!]^\sharp} x)$$

Fig. 5. Definitions of $\overleftarrow{[\![s]\!]^\sharp} : \mathbb{D} \to \mathbb{D}$.

The abstract domain $\mathbb{D}$ can be instantiated with different numerical abstract domains including Intervals [6], Octagons [7], and Polyhedra [14]. They differ in precision and computational complexity. In this work, we will mainly use the Polyhedra domain due to its precision. The *Polyhedra domain* [14] is a fully relational numerical abstract domain, which allows manipulating conjunctions of linear inequalities (constraints) of the form $\alpha_1 \texttt{x}_1 + \ldots + \alpha_n \texttt{x}_n \geq \beta$, where $\texttt{x}_1$, ..., $\texttt{x}_n$ are program variables and $\alpha_i, \beta \in \mathbb{Q}$ (rationals). The abstract operations, transfer functions, and the soundness of the Polyhedra domain are defined in [14].

Example 3.3: Consider the program sketch `while.c` in Fig. 1. Assume that the analysis fact $\top$ (true) holds at loc. ① and we want to perform forward reachability analysis based on the Polyhedra domain. In order to enforce the convergence of the analysis, we apply the widening operator at loc. ⓗ. The analysis results inferred by our forward analysis at locs. from ① to ⑤ are shown in Fig. 6 (left).

Similarly, we assume that the analysis fact $(\texttt{y=10})$ holds at loc. ⑤ and we want to perform backward sufficient condition analysis based on the Polyhedra domain. To enforce the convergence, we apply the lower widening operator at loc. ⓗ. The analysis results inferred by our backward analysis at locs. from ⑤ to ① are shown in Fig. 6 (right).

① $\top$	⑤ $(\texttt{y=10})$
② $(\texttt{x=10} \wedge \texttt{y=0})$	ⓗ $\bot$
ⓗ $(\texttt{x} \leq 10 \wedge \texttt{x+y=10})$	④ $\bot$
③ $(\texttt{x} \leq 10 \wedge \texttt{x+y=10})$	③ $\bot$
④ $(\texttt{x} \leq 9 \wedge \texttt{x+y=9})$	② $\bot$
⑤ $(\texttt{x} \leq 10 \wedge \texttt{x+y=10})$	① $\bot$

Fig. 6. Analysis results of `while.c` inferred by forward (left) and backward (right) analysis.

D. Constrained Horn Clauses (CHCs)

Given a formula ϕ in a first-order theory $\mathcal{T}$, the SMT (Satisfiability Modulo Theory) solvers [18] decide the satisfiability of ϕ, i.e. they check if there is an assignment of values to variables in ϕ that makes ϕ true. To check the validity of ϕ, they check the unsatisfiability of $\neg\phi$. Examples of first-order theories are: linear integer arithmetic (LIA), arrays, etc.

Constrained Horn Clauses (CHCs) are a fragment of first-order theory $\mathcal{T}$ defined over a finite set of unknown predicates $\mathcal{R}$. A CHC is a formula that has one of the following forms:

$$\phi(\overrightarrow{\mathbf{x}_0}) \implies R_0(\overrightarrow{\mathbf{x}_0}) \quad (6)$$
$$\wedge_{0 \leq i \leq n} R_i(\overrightarrow{\mathbf{x}_i}) \wedge \phi(\overrightarrow{\mathbf{x}_0}, \ldots, \overrightarrow{\mathbf{x}_{n+1}}) \implies R_{n+1}(\overrightarrow{\mathbf{x}_{n+1}}) \quad (7)$$
$$\wedge_{0 \leq i \leq n} R_i(\overrightarrow{\mathbf{x}_i}) \implies \phi(\overrightarrow{\mathbf{x}_0}, \ldots, \overrightarrow{\mathbf{x}_n}) \quad (8)$$

where $R_i(\overrightarrow{\mathbf{x}_i}) \in \mathcal{R}$ $(0 \leq i \leq n+1)$ is an unknown predicate defined over a vector of variables $\overrightarrow{\mathbf{x}_i}$, and $\phi(\overrightarrow{\mathbf{x}_0}, \ldots, \overrightarrow{\mathbf{x}_n})$ is a first-order formula that does not contain any unknown predicates. The *body* (resp., *head*) of a CHC represents the left (resp., right) side of the implication. A CHC of type (6) is called *fact*, of type (7) is called *inductive*, and of type (8) is called *query*. A CHC is called *linear* if the number of unknown predicates in its body is one, otherwise it is called *non-linear*.

A *solution* to a set of CHCs defined over unknown predicates $\mathcal{R}$ is a mapping $\mathcal{M}$ from $\mathcal{R}$ to first-order formulas, such that all implications in the set become valid. That is, for each $R(\overrightarrow{\mathbf{x}}) \in \mathcal{R}$, $\mathcal{M}(R)$ represents a first-order formula ϕ defined over the variables $\overrightarrow{\mathbf{x}}$. The CHC solvers, like `Eldarica` [11], `Spacer` [10], `Duality` [15], and `FreqHorn` [16], contain algorithms, denoted `CHC-solve`, for computing the correct solutions to sets of CHCs.

IV. Synthesis Algorithm

In this section, we present our algorithm for solving the `while`-guard synthesis problem. In particular, we employ the abstract static analyses, $\overrightarrow{[\![s]\!]^\sharp}$ and $\overleftarrow{[\![s]\!]^\sharp}$, as well as the CHC solving procedure, `CHC-solve`, to automatically find Boolean expressions for the missing holes in a program sketch, so that the resulting complete program satisfies its assertion. That is, we establish a link between checking an assertion in a program and solving CHCs via abstract static analyses, which provide a way to transition from programs to CHCs.

A. Description

The $\texttt{WhileSketching}_{\texttt{CHC}}(p, H)$ synthesis procedure is shown in Algorithm 1. The procedure takes as input two parameters: a program sketch p, and a set of holes H in p representing missing `while`-guards. It returns as output a solution (complete control function ϕ) or the empty mapping $\emptyset$ if no solution exists. For each hole $\texttt{??}_i$ in H, we first generate an initial solution, which is a complete Boolean expression be'_i, such that all other holes are treated as non-deterministic

Algorithm 1: WhileSketching$_{\texttt{CHC}}(p, H)$

Input: Program sketch p, and a set of holes H
Output: Complete control function ϕ or an empty mapping $\emptyset$

1 $\phi := \phi' := \emptyset$;
2 **for** *(* $??_i \in H$*)* **do**
3 $(l\colon \texttt{while}\, l_h : (??_i)\, \texttt{do}\, \{l_b : s_b\}\, l'\colon) := \texttt{Extract}(p, ??_i)$;
4 $\phi_i^{ndet} := [??_i \mapsto \bot] \uplus \{[??_j \mapsto \mathcal{P}(\{\text{true}, \text{false}\})] \mid j \neq i\}$;
5 $be_i' := \texttt{OneHoleSynth}(p[\phi_i^{ndet}], s_b, l, l_h, l')$;
6 **if** $(be_i' = \emptyset)$ **then return** $\emptyset$ **;**
7 $\phi := \phi \uplus [\mathbf{??}_i \mapsto be_i']$
8 **while** $(\phi \neq \phi')$ **do**
9 **if** $(\phi' \neq \emptyset)$ **then** $\phi = \phi'; \phi' = \emptyset$;
10 **for** *(* $??_i \in H$*)* **do**
11 $(l\colon \texttt{while}\, l_h : (??_i)\, \texttt{do}\, \{l_b : s_b\}\, l'\colon) := \texttt{Extract}(p, ??_i)$;
12 $\phi_i := \phi[??_i \mapsto \bot]$;
13 $be_i'' := \texttt{OneHoleSynth}(p[\phi_i], s_b, l, l_h, l')$;
14 **if** $(be_i'' = \emptyset)$ **then return** $\emptyset$ **;**
15 $\phi' := \phi' \uplus [\mathbf{??}_i \mapsto be_i'']$
16 **return** ϕ

Algorithm 2: `OneHoleSynth`(p, s_b, l, l_t, l')

Input: Program sketch $p \equiv l_i : s; l_f : \texttt{assert}(be^f)$, `while`-body s_b, and labels l, l_h, l'
Output: Complete Boolean expression be' or an empty mapping $\emptyset$

1 $\texttt{Reach}^\sharp := \overrightarrow{[\![s]\!]}^\sharp \top_{\mathbb{D}}$;
2 $\texttt{Cond}^\sharp := \overleftarrow{[\![s]\!]}^\sharp \texttt{FILTER}_{\mathbb{D}}(be^f, \top_{\mathbb{D}})$;
3 $C_1 := (\texttt{Reach}^\sharp(l) \implies Inv(\overrightarrow{\mathbf{x}}))$;
4 $C_2 := (Inv(\overrightarrow{\mathbf{x}}) \wedge R(\overrightarrow{\mathbf{x}}) \wedge (\overrightarrow{[\![\overrightarrow{\mathbf{x'}} = \overrightarrow{\mathbf{x}}; s_b[\overrightarrow{\mathbf{x'}}/\overrightarrow{\mathbf{x}}]]\!]^\sharp} \top_{\mathbb{D}}) \implies Inv(\overrightarrow{\mathbf{x'}}))$;
5 $C_3 := (Inv(\overrightarrow{\mathbf{x}}) \wedge \neg R(\overrightarrow{\mathbf{x}}) \implies \texttt{Cond}^\sharp(l'))$;
6 $C_4 := (Inv(\overrightarrow{\mathbf{x}}) \implies \texttt{Reach}^\sharp(l_h))$;
7 $C_5 := (\texttt{Cond}^\sharp(l_h) \implies Inv(\overrightarrow{\mathbf{x}}))$;
8 $CHC := \{C_1, C_2, C_3, C_4, C_5\}$;
9 $\mathcal{M} := \texttt{CHC-solve}(CHC)$;
10 **if** $(\mathcal{M}(R(\overrightarrow{\mathbf{x}})) \Leftrightarrow \mathcal{M}(Inv(\overrightarrow{\mathbf{x}})))$ **then return** $\emptyset$ **;**
11 **return** `SMTSimpl`$((\mathcal{M}(R(\overrightarrow{\mathbf{x}})) \wedge \neg \mathcal{M}(Inv(\overrightarrow{\mathbf{x}}))))$

choices over Booleans, $\mathcal{P}(\{\text{true}, \text{false}\})$ (lines 2–7). First, we identify the statement "$(l$: `while` l_h : $(??_i)$ `do` $\{l_b$: $s_b\}$ l' :)" in which $??_i$ occurs by using $\texttt{Extract}(p, ??_i)$ procedure (line 3). Second, we construct a partial control function ϕ_i^{ndet}, such that $\phi_i^{ndet}(??_i) = \bot$ (undefined) and $\phi_i^{ndet}(??_j) = \mathcal{P}(\{\text{true}, \text{false}\})$ for all other $??_j \in H$ (line 4), and we generate a program sketch $p[\phi_i^{ndet}]$ in which the hole $??_i$ is the only one. This has the effect of treating any hole other than $??_i$ in $p[\phi_i^{ndet}]$ as a non-deterministic choice. Finally, we call the one-hole synthesis procedure, called $\texttt{OneHoleSynth}(p[\phi_i^{ndet}], s_b, l, l_h, l')$ to generate the solution be_i' corresponding to hole $??_i$ (line 5). This way, we generate an *initial* complete control function $\phi : [??_i \mapsto be_i']$ by using the above initial solutions for all holes $??_i$ in H.

Subsequently, Algorithm 1 goes through an iterative `while`-loop (lines 8-15) performing the weakening of the current solution ϕ, thus generating more general solutions ϕ'. The algorithm terminates when the refinement is no longer enabled, i.e. $\phi = \phi'$ (line 8). In particular, in each `while`-iteration we weaken the current solution ϕ by iteratively weakening ϕ-solutions for all holes (lines 10–15). To weaken the ϕ-solution for each $??_i$, we fix the ϕ-solutions be_j' for all other $??_j$ in a partial control function ϕ_i, such that $\phi_i(??_i) = \bot$ and $\phi_i(??_j) = \phi(??_j)$ for all other $??_j \in H$ (line 12). Then, we generate a program sketch $p[\phi_i]$ in which the hole $??_i$ is the only one. Finally, we call $\texttt{OneHoleSynth}(p[\phi_i^{ndet}], s_b, l, l_h, l')$ to find a weaker Boolean expression be_i'' for $??_i$ (line 13), which is located at statement $(l$: `while` l_h : $(??_i)$ `do` $\{l_b$: $s_b\}$ l' :). The generated be_i'' will be included in the updated solution ϕ' (line 15). In particular, we use the solutions given by the current control function for all other holes and infer more general Boolean expression for $??_i$ that implies the assertion validity.

B. One-hole Synthesis Procedure

The $\texttt{OneHoleSynth}(p, s_b, l, l_t, l')$ procedure, shown in Algorithm 2, takes five parameters: a program sketch $p \equiv l_i : s; l_f : \texttt{assert}(be^f)$ with one hole in it, a body s_b of the `while` with missing guard in s, and three labels: l, l_h, and l', where the `while`-guard $??$ we want to handle is located (l and l' are locations before and after the `while`, and l_h is location at the `while`-head). We first call the forward abstract analysis $\overrightarrow{[\![s]\!]}^\sharp \top_{\mathbb{D}}$ (line 1) and the backward abstract analysis $\overleftarrow{[\![s]\!]}^\sharp d_{\mathcal{F}}$ (line 2), where $d_{\mathcal{F}} = \texttt{FILTER}_{\mathbb{D}}(be^f, \top_{\mathbb{D}})$ represents the assertion fact, that proceed by structural induction on the program syntax of s. This way, we compute the invariants $\texttt{Reach}^\sharp$ as well as the sufficient conditions $\texttt{Cond}^\sharp$ in all locations of p. Assume that $\texttt{Reach}^\sharp(l)$ is the inferred invariant at loc. l before the `while`, while $\texttt{Cond}^\sharp(l')$ is the inferred sufficient condition at loc. l' after the `while` that ensures assertion-safety of the code that comes after the hole $??$. Moreover, we assume that $\texttt{Reach}^\sharp(l_h)$ and $\texttt{Cond}^\sharp(l_h)$ are the inferred analysis facts at loc. l_h for the inductive `while`-invariant by the forward and backward analysis, respectively. By using the inferred $\texttt{Reach}^\sharp(l)$, $\texttt{Reach}^\sharp(l_h)$, $\texttt{Cond}^\sharp(l')$, and $\texttt{Cond}^\sharp(l_h)$, we then construct CHCs with two unknown predicates: $Inv(\overrightarrow{\mathbf{x}})$ and $R(\overrightarrow{\mathbf{x}})$ corresponding to the inductive `while`-invariant and the guard of the `while` with the hole $??$, where $\overrightarrow{\mathbf{x}}$ is a vector of all program variables defined in the scope of the given `while`. More specifically, we generate the following five CHC queries for all cases involving the given `while`. The inductive `while`-invariant $Inv(\overrightarrow{\mathbf{x}})$ expresses a relationship between the program variables that is implied by the `while`'s precondition and is preserved by any execution of the `while`-body. For this aim, we use the CHCs C_1 and C_2 to ensure that the `while`-invariant $Inv(\overrightarrow{\mathbf{x}})$ is inductive. In particular, C_1 states that the invariant at the `while`'s precondition $\texttt{Reach}^\sharp(l)$ implies $Inv(\overrightarrow{\mathbf{x}})$. To state

that $Inv(\overrightarrow{\mathbf{x}})$ is preserved in each iteration of the `while`'s body, we need a way of remembering the initial values of all variables $\overrightarrow{\mathbf{x}}$ in the scope of the given `while`. To do so, we analyze the program: $\overrightarrow{\mathbf{x'}} = \overrightarrow{\mathbf{x}}; s_b[\overrightarrow{\mathbf{x'}}/\overrightarrow{\mathbf{x}}]$, where $\overrightarrow{\mathbf{x'}} = \overrightarrow{\mathbf{x}}$ assign all program variables $\overrightarrow{\mathbf{x}}$ to their primed versions $\overrightarrow{\mathbf{x'}}$ and $s_b[\overrightarrow{\mathbf{x'}}/\overrightarrow{\mathbf{x}}]$ means that all variables $\overrightarrow{\mathbf{x}}$ in s_b are renamed to their primed versions $\overrightarrow{\mathbf{x'}}$. This way, we build the required CHC C_2 stating that if $Inv(\overrightarrow{\mathbf{x}})$ and $R(\overrightarrow{\mathbf{x}})$ hold and we analyze $\overrightarrow{\mathbf{x'}} = \overrightarrow{\mathbf{x}}; s_b[\overrightarrow{\mathbf{x'}}/\overrightarrow{\mathbf{x}}]$, then $Inv(\overrightarrow{\mathbf{x'}})$ will hold after that. That is, executing `while`'s body when $R(\overrightarrow{\mathbf{x}})$ holds re-establishes the `while`-invariant $Inv(\overrightarrow{\mathbf{x}})$. The CHC C_3 states that on `while` exit, the `while`-invariant $Inv(\overrightarrow{\mathbf{x}})$ strengthen with $\neg R(\overrightarrow{\mathbf{x}})$ are sufficiently strong to imply the `while`'s postcondition $\texttt{Cond}^\sharp(l')$ ensuring that the assertion is correct. The CHCs C_4 and C_5 state that the analysis facts $\texttt{Reach}^\sharp(l_h)$ and $\texttt{Cond}^\sharp(l_h)$ inferred at loc. l_h by forward and backward analysis are over- and under-approximations of $Inv(\overrightarrow{\mathbf{x}})$, respectively.

Finally, we call the `CHC-solve` procedure to find the solution (mapping) $\mathcal{M}$ of the constructed set of CHCs, such that $\mathcal{M}(Inv)$ and $\mathcal{M}(R)$ are first-order formulas defined over variables $\overrightarrow{\mathbf{x}}$. Since the solution $\mathcal{M}(R)$ represents a strengthen formula of $\mathcal{M}(Inv)$, we can use $\mathcal{M}(R) \wedge \neg\mathcal{M}(Inv)$ as the final solution be' to replace the given Boolean expression hole `??` (line 11). However, if $(\mathcal{M}(R) \Leftrightarrow \mathcal{M}(Inv))$, i.e. $\mathcal{M}(R)$ and $\mathcal{M}(Inv)$ are equivalent, then the given problem is not solvable and an empty mapping is returned (line 10).

C. Correctness

The following theorem states correctness of the $\texttt{WhileSketching}_{\texttt{CHC}}$ algorithm.

Theorem 1: $\texttt{WhileSketching}_{\texttt{CHC}}(p, H)$ is correct and terminates.

Proof: The procedure $\texttt{WhileSketching}_{\texttt{CHC}}(p, H)$ terminates since all steps in it are terminating. The `while`-loop (lines 8-15) terminates since in the worst case it will generate the weakest solutions true for all holes. The correctness of $\texttt{WhileSketching}_{\texttt{CHC}}(p, H)$ follows from the soundness of $\texttt{Reach}^\sharp$ and $\texttt{Cond}^\sharp$ (see Proposition 3.2) and the correctness of `CHC-solve` [11], [10], [15], [16]. ■

V. Abduction-based Synthesis

We now give an overview of the logical abduction-based approach for `while`-guard synthesis, called $\texttt{GenSketching}_{\texttt{Abd}}$, introduced in [5]. The logical abduction [9] is a technique for inferring a single unknown predicate $R(\overrightarrow{\mathbf{x}})$, which is defined over a vector of program variables $\overrightarrow{\mathbf{x}}$ from an implication formula:

$$R(\overrightarrow{\mathbf{x}}) \wedge \chi \implies C$$

where χ and C are first-order logical formulas. The solution to the above abduction problem is formula ϕ, which represents an interpretation of $R(\overrightarrow{\mathbf{x}})$, such that (1) $\phi \wedge \chi \not\models$ false; and (2) $\phi \wedge \chi \models C$. The procedure $\texttt{Abduce}(\chi, C, \overrightarrow{\mathbf{x}})$ that finds the logically weakest solution containing a fewest number of variables to the above abduction problem is implemented in the `EXPLAIN` tool [9].

We now show how to handle the case when there is a single hole in a sketch p, which occurs in the statement of the form: "l : (`while` l_h : (`??`) `do` $\{l_b : s_b\})\, l'$:". The case when there are more than one holes in a program sketch is handled similarly as in Algorithm 1. First, we perform forward and backward abstract static analyses of p to compute the invariants $\texttt{Reach}^\sharp$ and the sufficient conditions $\texttt{Cond}^\sharp$ in all locations of p. Then, we construct an abduction query where the premise is $\texttt{Reach}^\sharp(l_h)$, the desired conclusion is $\texttt{Cond}^\sharp(l')$, and the unknown predicate is $R_{false}(\overrightarrow{\mathbf{x}})$, that is: $\texttt{Reach}^\sharp(l_h) \wedge R_{false}(\overrightarrow{\mathbf{x}}) \implies \texttt{Cond}^\sharp(l')$. The meaning of this query is that when `while` terminates, the `while`-invariant $\texttt{Reach}^\sharp(l_h)$ and the negation of the missing `while`-guard, encoded via the unknown predicate $R_{false}(\overrightarrow{\mathbf{x}})$, should imply the sufficient condition $\texttt{Cond}^\sharp(l')$ ensuring that the code after `while` guarantees the assertion validity.

Next, we find $R_{true}(\overrightarrow{\mathbf{x}})$ that is the negation of the formula found for $R_{false}(\overrightarrow{\mathbf{x}})$, i.e. $R_{true}(\overrightarrow{\mathbf{x}}) \equiv (\neg R_{false}(\overrightarrow{\mathbf{x}}))$. Assume that $R_{true}(\overrightarrow{\mathbf{x}})$ is of the form: $\phi_1 \vee \ldots \vee \phi_n$, where ϕ_i is a conjunction of formulas. We perform n backward analyses defined as follows: $\texttt{Cond}^{\sharp,1} = [\![p[\texttt{??} \mapsto (\phi_1 \wedge \texttt{??})]]\!]^\sharp(\texttt{FILTER}_{\mathbb{D}}(be^f, \top_{\mathbb{D}})), \ldots, \texttt{Cond}^{\sharp,n} = [\![p[\texttt{??} \mapsto (\phi_n \wedge \texttt{??})]]\!]^\sharp(\texttt{FILTER}_{\mathbb{D}}(be^f, \top_{\mathbb{D}}))$, where $p[\texttt{??} \mapsto (\phi_i \wedge \texttt{??})]$ is the sketch p in which the hole `??` is replaced with $(\phi_i \wedge \texttt{??})$. We create n abduction queries: $\texttt{Reach}^\sharp(l_h) \wedge R^1_{true}(\overrightarrow{\mathbf{x}}) \implies \texttt{Cond}^{\sharp,1}(l_b)$, ..., $\texttt{Reach}^\sharp(l_h) \wedge R^n_{true}(\overrightarrow{\mathbf{x}}) \implies \texttt{Cond}^{\sharp,n}(l_b)$. Finally, the solution for `??` is the Boolean expression: $(\phi_1 \wedge R^1_{true}(\overrightarrow{\mathbf{x}})) \vee \ldots \vee (\phi_n \wedge R^n_{true}(\overrightarrow{\mathbf{x}}))$. In the implementation, we can make some optimizations to reduce the number of calls to the abduction solver. For example, if $\texttt{Cond}^{\sharp,i}(l_b) =$ false then we can conclude that $R^i_{true}(\overrightarrow{\mathbf{x}}) \equiv$ (false) without calling the abduction solver.

Example 5.1: Consider the sketch `while.c` given in Fig. 1. The forward analysis infers the invariant $(\texttt{x} \leq 10) \wedge (\texttt{x+y=10})$ at loc. l_h. We construct the abduction query $(\texttt{x} \leq 10) \wedge (\texttt{x+y=10}) \wedge R_{false}(\texttt{x,y}) \implies (\texttt{y=10})$. The reported solution by EXPLAIN is $R_{false}(\texttt{x,y}) \equiv (\texttt{x=0})$. We compute $R_{true}(\texttt{x,y}) \equiv \neg R_{false}(\texttt{x,y}) \equiv (\texttt{x<0}) \vee (\texttt{x>0})$. Hence, we perform two backward analysis of `while.c` in which the `while`-guard is $(\texttt{x<0} \wedge \texttt{??}_1)$ and $(\texttt{x>0} \wedge \texttt{??}_2)$, respectively. They start with the fact $(\texttt{x=0} \wedge \texttt{y=10})$ at loc. ⑤. The first backward analysis for $(\texttt{x<0} \wedge \texttt{??}_1)$ infers the bottom ($\bot$) condition after the guard at loc. ③, whereas the second backward analysis for $(\texttt{x>0} \wedge \texttt{??}_2)$ infers $(1 \leq \texttt{x} \leq 11) \wedge (\texttt{x+y=10})$ after the guard at loc. ③. We construct two abduction queries: (1) $(\texttt{x} \leq 10 \wedge \texttt{x+y=10}) \wedge R^1_{true}(\texttt{x,y}) \implies$ false, and (2) $(\texttt{x} \leq 10 \wedge \texttt{x+y=10}) \wedge R^2_{true}(\texttt{x,y}) \implies (1 \leq \texttt{x} \leq 11) \wedge (\texttt{x+y=10})$. The obtained solutions are $R^1_{true}(\texttt{x,y}) \equiv$ false and $R^2_{true}(\texttt{x,y}) \equiv$ true. Hence, we fill the hole with $(\texttt{x>0})$.

VI. Evaluation

We now evaluate our approach based on abstract static analysis and CHC solving for synthesizing missing `while`-guards in numerical sketches written in C. We compare its performances against the approach that uses logical abduction for constraint solving.

```
void main(int n){
  assume(n ≥ 0)
  int i = 0;
  while (??) do{
      i = i+2;
  }
  assert(i==2*n);
 // assert(i==4*n);
 // assert(i==0);
 // assert(i<0);
}
```

Fig. 7. eq2.c.

```
void main(int x) {
  assume(x ≥ 0)
  int a = x;
  int y = 0;
  while(??) do{
    y = y+1;
    a = a-1;
  }
  assert(x==y);
 // assert(a==0);
 // assert(a+2==x);
 // assert(y<0);
}
```

Fig. 9. copy.c.

```
void main() {
  int i = 0;
  int j = 0;
  while(??) do{
    i = i+1;
    j = j+1;
  }
  assert(i+j==20);
 // assert(i+j>20);
 // assert(i+j≤15);
 // assert(i+j==15);
}
```

Fig. 8. loop2.c.

```
void main() {
  int x = 0;
  int y = 0;
  while(??) do{
    x = x+1;
    y = y+2;
  }
  assert(y==2*m);
 // assert(y==x);
 // assert(y==x && x>0);
 // assert(y==2*m+1);
}
```

Fig. 10. hola.c.

A. Implementation

We have implemented the proposed synthesis algorithm in a proof-of-concept tool, called $\texttt{WhileSketching}_{\texttt{CHC}}$. The abstract operations and transfer functions of the Polyhedra abstract domain [14] are provided by the APRON library [17]. The CHC and SMT queries are solved by the `Eldarica` [11] and the `Z3` [18] tools. Our proof-of-concept tool is written in OCAML and consists of around 7K lines of code. The tool accepts programs written in a subset of C. It currently provides only a limited support for arrays, pointers, `struct` and `union` types. The only basic data type is mathematical integers. The tool calls the APRON library for static analysis of the input program, then generates CHC constraints and calls the `Eldarica` tool to solve them. Finally, it handles the obtained results and reports the solutions.

B. Experiment setup and Benchmarks

All experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU, VM Lubuntu 18.04, with 16 GB memory, and we use a timeout value of 60 sec. All times are reported as average over five independent executions. We compare two approaches:

(1) $\texttt{WhileSketching}_{\texttt{CHC}}$ based on abstract static analysis and CHC solving; and
(2) $\texttt{GenSketching}_{\texttt{Abd}}$ based on abstract static analysis and abduction solving.

We report TIME which is the total time to resolve a given problem, SOLVTIME which is the time taken by logical (CHC or abduction) solvers to solve the logical queries in the given synthesis task, and PREC which shows the precision of the reported solution. We use: ✓ when the tool finds correct solution for a given assertion; and × denotes the opposite, i.e. the tool cannot find a correct solution for the assertion.

The evaluation is performed on several C numerical sketches collected from SV-COMP, (https://sv-comp.sosy-lab.org/) and the literature [19], [5] In particular, we use the following benchmarks: `while.c` (Fig. 1), `eq2.c` (Fig. 7), `loop2.c` (Fig. 8), `copy.c` (Fig. 9), and `hola.c` (Fig. 10). In all benchmarks, we assume that `while`-guards are missing and we employ the aforementioned approaches to synthesize suitable Boolean expressions for them.

C. Performance Results

Table I shows the performance results of $\texttt{WhileSketching}_{\texttt{CHC}}$ and $\texttt{GenSketching}_{\texttt{Abd}}$.

We described in Section II how $\texttt{WhileSketching}_{\texttt{CHC}}$ successfully handles the sketch `while.c` given in Fig. 1. $\texttt{GenSketching}_{\texttt{Abd}}$ also finds the correct solutions for `while.c`, but for the first two assertions it needs two calls to the abduction solver, which results in slower running times.

The program sketch `eq2.c` in Fig. 7 has one local variable `i`, which is initialized to `0` and increased by `2` in every `while`-iteration. The forward static analysis infers the invariant $(\texttt{i=0} \wedge \texttt{n} \geq \texttt{0})$ before the `while` and $(\texttt{i} \geq \texttt{0} \wedge \texttt{n} \geq \texttt{0})$ at the `while`-head. For the specification `assert(i==2*n)`, $\texttt{WhileSketching}_{\texttt{CHC}}$ constructs the corresponding CHCs

TABLE I
PERFORMANCE RESULTS OF WHILESKETCHING$_{CHC}$ VS. GENSKETCHING$_{ABD}$. ALL TIMES IN SEC.

Bench.	assert	WhileSketching$_{CHC}$			GenSketching$_{Abd}$		
		TIME	SOLVTIME	Prec	TIME	SOLVTIME	Prec
while.c	(y=10)	0.920	0.904	✓	2.011	1.836	✓
while.c	(y>x)	0.924	0.901	✓	2.082	1.874	✓
while.c	(x=10)	0.817	0.796	✓	0.746	0.611	✓
while.c	(x>15)	0.921	0.900	✓	0.744	0.630	✓
eq2.c	(i=2⋆n)	0.939	0.889	✓	1.337	1.251	✓
eq2.c	(i=4⋆n)	0.901	0.871	✓	1.578	1.402	✓
eq2.c	(i=0)	0.813	0.789	✓	0.742	0.637	✓
eq2.c	(i<0)	0.903	0.872	✓	0.745	0.631	✓
loop2.c	(i+j=20)	0.945	0.911	✓	1.502	1.364	✓
loop2.c	(i+j>20)	0.936	0.902	✓	1.558	1.404	✓
loop2.c	(i+j<=20)	0.814	0.786	✓	1.543	1.376	✓
loop2.c	(i+j=15)	0.951	0.910	✓	0.751	0.665	✓
copy.c	(x=y)	0.979	0.937	✓	1.693	1.542	×
copy.c	(a=0)	0.954	0.921	✓	1.511	1.382	×
copy.c	(a+2=x)	0.950	0.918	✓	1.451	1.329	×
copy.c	(y<0)	0.947	0.916	✓	0.757	0.679	✓
hola.c	(y=2⋆m)	0.948	0.913	✓	1.511	1.364	×
hola.c	(y=x)	0.825	0.764	✓	1.542	1.350	✓
hola.c	(y=x∧x>0)	0.952	0.921	✓	0.756	0.667	✓
hola.c	(y=2⋆m+1)	0.926	0.902	✓	0.748	0.664	×

and generates the solution (i ≠ n) for the while-guard ??. For the assertions (i==4⋆n), (i==0), and (i<0), we synthesize the following while-guards: (i ≠ 2⋆n), false, and the problem is unrealizable, respectively. On the other hand, GenSketching$_{Abd}$ reports the correct solutions for eq2.c, but it shows slower performance times than WhileSketching$_{CHC}$ for the first two assertions.

Consider the program sketch loop2.c in Fig. 8. The forward static analysis infers the invariant (i=0∧j=0) before the while and (i=0 ∧ i=j) at the while-head. Hence, WhileSketching$_{CHC}$ infers the solutions (i ≠ 10), (i ≤ 10), and false for the assertions (i+j==20), (i+j>20), and (i+j ≤ 15), respectively. However, for the assertion (i+j==15), we obtain that the problem is unrealizable (i.e., it has no solution). GenSketching$_{Abd}$ finds the correct solutions as well, but is slower for three assertions.

The program sketch copy.c in Fig. 9 copies the input value of the argument x into the local variable y. This is specified by the final assertion (x==y). The forward static analysis infers the invariant (x ≥ 0 ∧ y=0 ∧ x=a) before the while and (x ≥ 0 ∧ y=x−a ∧ x−a ≥ 0) at the while-head. Hence, WhileSketching$_{CHC}$ infers the solution (x ≠ y) for the missing while-guard ??. For the assertions (a==0), (a+2==x), and (y<0), it generates the while-guards: (x−y ≥ 1), (y ≠ 2), and the problem is unrealizable, respectively. On the other hand, GenSketching$_{Abd}$ fails to find solutions for three assertions.

Consider the program sketch hola.c in Fig. 10 [19]. The forward static analysis infers the invariant (x=0 ∧ y=0) before the while and (x ≥ 0 ∧ y=2⋆x) at the while-head. This way, WhileSketching$_{CHC}$ finds the solution (m ≠ x) for the assertion (y==2⋆m). For assertion (y==x) the solution is (false), whereas for assertions (y==x)∧(x>0) and (y==2⋆m+1) we obtain that the problems are unrealizable. GenSketching$_{Abd}$ fails to produce correct solutions for two assertions in this case.

Discussion: In summary, as Table I shows, the proposed technique WhileSketching$_{CHC}$ is quite effective at successfully synthesizing the missing while-guards ?? in all considered benchmarks. Note that the tasks that synthesize the solution (false) are the fastest ones for WhileSketching$_{CHC}$ achieving around 10% speed-up with respect to the other tasks for the same benchmark. Moreover, the CHC and abduction solvers take more than 95% of the total synthesis time for both WhileSketching$_{CHC}$ and GenSketching$_{Abd}$. The synthesis time of GenSketching$_{Abd}$ depends on how many times the abduction solver is called. When the Abduction solver is called once, then GenSketching$_{Abd}$ and WhileSketching$_{CHC}$ achieve comparable synthesis times, since WhileSketching$_{CHC}$ always calls the CHC solver once. However, we can see that GenSketching$_{Abd}$ cannot find solutions for five assertions, thus returning empty sets. On the one hand, this is due to the limitation of the abduction solver and the difficulty of static analyzers to infer precise inductive while-invariants in the presence of missing while-guards, but on the other hand, this is also due to the suitability of CHC solvers for handling complex control flow.

The current tool supports an interesting subset of C, so we can handle many interesting programs. The selected benchmarks are chosen to show some distinctive properties of our approach and its ability to be applied in practice.

VII. RELATED WORK

We divide our discussion of related work into three categories: program sketching; abstract static analysis; and CHC solving.

A. Program sketching

The widely-known SKETCH tool [2], [3] represents one of the earliest attempts to resolve program sketches in which missing holes can be only integer constants. It uses SAT-based counterexample-guided inductive synthesis. In particular, SKETCH iteratively generates a finite set of inputs and performs SAT queries to identify values for the holes such that the obtained complete program satisfies all assertions for the considered inputs. Additional SAT queries are then checked to establish whether the obtained program is correct on all possible inputs. SKETCH handles `while`-s by unrolling them, which makes its performance very sensitive to the degree of unrolling. Since it is based on SAT solvers, SKETCH is especially suited for handling bit-manipulating programs.

FAMILYSKETCHER [4], [20], [21] is another tool for solving the sketching problem by using lifted (family-based) static analysis based on abstract interpretation [22], [23]. The key idea underlying this approach is that the set of all possible sketch realizations can be represented as a program family (Software Product Line) with numerical features [24], [25], [26], which is subsequently statically analyzed to establish assertion validity. This way, the effort of conducting an effective search of all possible hole realizations is delegated to an efficient lifted static analyzer for program families (SPLs), which uses a specifically designed decision tree abstract domain. Since FAMILYSKETCHER is based on numerical abstract domains, it can handle unbounded `while`-s via widening and narrowing, and moreover it is very suited for handling numerical programs with integer data types.

However, SKETCH and FAMILYSKETCHER can only resolve program sketches in which unknown holes can be replaced by one constant value from a finite set of integers. $\texttt{GenSketching}_{\texttt{Abd}}$ [5] considers so-called generalized sketching problem, where each hole can be replaced by an arbitrary expression. It uses abstract static analysis for constraint generation and abduction solvers for constraint solving. Similarly to FAMILYSKETCHER, $\texttt{GenSketching}_{\texttt{Abd}}$ can handle unbounded `while`-s and numerical programs. However, this approach is not very effective in handling missing `while`-guards, and it often fails to find any solution. In this paper, we pursue this line of work by using CHC solvers rather than abduction solvers, which are more suitable for automatically handling more complex `while`-loops. The abduction solvers have also been applied in program synthesis to infer missing `if`-guards from low-level C code such that all buffer accesses are memory safe [27], to infer missing `#if`-guards from numerical program families so that a given assertion is valid in all members of the family [28], to find the weakest specifications of unknown (library) functions that ensure the safety of the program calling those external functions [29], [30], etc.

B. Abstract static analysis

Abstract interpretation [6], [14], [7], [31], [32] is a general theory for approximating the semantics of programs. It provides sound and efficient static analyses for inferring run-time properties of programs. The most popular static analyzers employ the well-known numerical abstract domains of Intervals [6], Octagons [7], and Polyhedra [14], that can be used for inferring numerical properties of program variables. Mine [32] has introduced the under-approximating backward polyhedral analysis for inferring sufficient conditions of non-deterministic programs, which has been implemented as part of the APRON library in the BANAL tool [32].

Several successful static analyzers based on abstract interpretation have been developed recently for verifying real-world programs. For example, ASTREE [33] is a static analyzer for verifying avionics software; ULITIMATE TAIPAN [34] is a software model checker that uses abstract interpretation to derive invariants for the path program corresponding to a given spurious counterexample; PAGAI [35] combines SMT-solving with abstract interpretation, etc.

Several approaches use a combination of forward and backward static analyses to infer more precise interesting properties of programs. More specifically, Rival [36] uses this combination to detect a set of traces leading to a bug; Bourdoncle [31] to generate preconditions ensuring validity of invariant and intermittent assertions; Dimovski and Legay [37] to compute the probability that a given assertion is satisfied (or violated); and Urban and Miné [8] to find ranking functions for proving termination.

C. Constraint Horn Clauses

Recently, the problem of CHC solving has gained a lot of attention due to their wide applications in program verification and program synthesis. Various algorithms and tools for CHC solving have been developed. Some of the most prominent tools are: `Eldarica` [11] that combines predicate abstraction with counterexample-guided abstraction refinement; `Spacer` [10] that uses generalized property-driven reachability analysis; `Duality` [15] that uses predicate abstraction and Craig interpolation for abstraction refinement; `FreqHorn` [16] that uses Syntax-Guided Synthesis (SyGuS) [38]. Optimal CHC solving [39], which calculates maximal and minimal solutions of a given set of CHCs, uses techniques from program termination analysis to check the optimality of a solution of CHCs. Algorithms for solving a richer formalism of existentially quantified Horn clauses have also been studied [40], [41].

CHC solving has found a number of applications in program verification and program synthesis. SEAHORN [42] and JAYHORN [43] are tools for verification of safety properties of C and Java programs by combining abstract interpretation and CHC solving. They encode Hoare logic-style verification conditions as CHCs, which are subsequently solved via CHC solvers. CHC solving has also been used for determining refinement types of higher-order functional programs [44], for analysis of business processes expressed as Petri nets [45], etc. The existential Horn clauses have been used for the weakest strategy synthesis of safety games [46].

CHC solving has also been applied for synthesizing imperative programs from input-output examples as specifications in the semantics-guided synthesis (SemGuS) framework [47], [48]. The syntax, semantics, and specifications of imperative

programs are represented as CHCs, thus effectively reducing the program synthesis problem into a proof search over CHCs. This approach is specialized on proving unrealizable problems by showing that no imperative program from a given grammar exists that satisfies the specification. The SemGuS framework is also used for synthesizing regular expressions, Boolean formulas, and SyGuS problems. There are other approaches for imperative program synthesis that are specialized for solving realizable problems. SIMPL [49] combines the enumerative search with abstract static analysis to infer programs from input-output examples, while IMPSYNTH [50] combines enumerative search, abstract static analysis, and SMT mutations [51] to infer programs from logic specifications.

VIII. CONCLUSION

In this work, we introduce a synthesis algorithm for automatically inferring arbitrary Boolean expressions for unknown `while`-guards in program sketches. It is implemented by interaction between abstract static analyses and CHC solving. By means of experiments, we demonstrate the effectiveness of our algorithm on a set of interesting C benchmarks that cannot be handled by other existing tools.

In the future, we plan to extend our approach by considering program sketches in which apart from `while`-guards, unknown holes can also be `if`-guards, arithmetic expressions in assignments, and ultimately arbitrary statements. We envision achieving this aim by further interplay between abstract static analyses, CHC solving, and abduction solving.

REFERENCES

[1] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1989, pp. 179–190. [Online]. Available: https://doi.org/10.1145/75277.75293

[2] A. Solar-Lezama, "Program sketching," *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0249-7

[3] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 281–294. [Online]. Available: https://doi.org/10.1145/1065010.1065045

[4] A. Dimovski, S. Apel, and A. Legay, "Program sketching using lifted analysis for numerical program families," in *NASA Formal Methods, NFM 2021*, ser. LNCS, vol. 12673. Springer, 2021, pp. 95–112. [Online]. Available: https://doi.org/10.1007/978-3-030-76384-8_7

[5] A. S. Dimovski, "Generalized program sketching by abstract interpretation and logical abduction," in *Static Analysis - 30th International Symposium, SAS 2023, Proceedings*, ser. LNCS, vol. 14284. Springer, 2023, pp. 212–230. [Online]. Available: https://doi.org/10.1007/978-3-031-44245-2_11

[6] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL'77*. ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[7] A. Miné, "Tutorial on static inference of numeric invariants by abstract interpretation," *Foundations and Trends in Programming Languages*, vol. 4, no. 3-4, pp. 120–372, 2017. [Online]. Available: https://doi.org/10.1561/2500000034

[8] C. Urban and A. Miné, "A decision tree abstract domain for proving conditional termination," in *Static Analysis - 21st International Symposium, SAS 2014. Proceedings*, ser. LNCS, vol. 8723. Springer, 2014, pp. 302–318. [Online]. Available: https://doi.org/10.1007/978-3-319-10936-7_19

[9] I. Dillig and T. Dillig, "Explain: A tool for performing abductive inference," in *Computer Aided Verification - 25th International Conference, CAV 2013. Proceedings*, ser. LNCS, vol. 8044. Springer, 2013, pp. 684–689. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_46

[10] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," in *Computer Aided Verification - 26th International Conference, CAV 2014. Proceedings*, ser. LNCS, vol. 8559. Springer, 2014, pp. 17–34. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_2

[11] H. Hojjat and P. Rümmer, "The ELDARICA horn solver," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018*. IEEE, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8603013

[12] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," *Acta Informatica*, vol. 2, pp. 335–355, 1973.

[13] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006. [Online]. Available: https://doi.org/10.1007/s10990-006-8609-1

[14] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Conference Record of the Fifth Annual ACM Symposium on POPL'78*. ACM Press, 1978, pp. 84–96. [Online]. Available: https://doi.org/10.1145/512760.512770

[15] N. S. Bjørner, K. L. McMillan, and A. Rybalchenko, "On solving universally quantified horn clauses," in *Static Analysis - 20th International Symposium, SAS 2013. Proceedings*, ser. LNCS, vol. 7935. Springer, 2013, pp. 105–125. [Online]. Available: https://doi.org/10.1007/978-3-642-38856-9_8

[16] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Quantified invariants via syntax-guided synthesis," in *Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings, Part I*, ser. LNCS, vol. 11561. Springer, 2019, pp. 259–277. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_14

[17] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *21st Int. Conf., CAV 2009. Proceedings*, ser. LNCS, vol. 5643. Springer, 2009, pp. 661–667. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_52

[18] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *14th International Conference, TACAS 2008. Proceedings*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[19] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, "Inductive invariant generation via abductive inference," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*. ACM, 2013, pp. 443–456. [Online]. Available: https://doi.org/10.1145/2509136.2509511

[20] A. S. Dimovski, "Quantitative program sketching using lifted static analysis," in *25th Int. Conf. FASE 2022, Proceedings*, ser. LNCS, vol. 13241. Springer, 2022, pp. 102–122. [Online]. Available: https://doi.org/10.1007/978-3-030-99429-7_6

[21] A. Dimovski, "Quantitative program sketching using decision tree-based lifted analysis," *J. Comput. Lang.*, vol. 75, p. 101206, 2023. [Online]. Available: https://doi.org/10.1016/j.cola.2023.101206

[22] A. S. Dimovski, "A binary decision diagram lifted domain for analyzing program families," *J. Comput. Lang.*, vol. 63, p. 101032, 2021. [Online]. Available: https://doi.org/10.1016/j.cola.2021.101032

[23] A. S. Dimovski, S. Apel, and A. Legay, "Several lifted abstract domains for static analysis of numerical program families," *Sci. Comput. Program.*, vol. 213, p. 102725, 2022. [Online]. Available: https://doi.org/10.1016/j.scico.2021.102725

[24] A. S. Dimovski, C. Brabrand, and A. Wasowski, "Finding suitable variability abstractions for family-based analysis," in *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, ser. LNCS, vol. 9995. Springer, 2016, pp. 217–234. [Online]. Available: https://doi.org/10.1007/978-3-319-48989-6_14

[25] A. S. Dimovski and A. Wasowski, "From transition systems to variability models and from lifted model checking back to UPPAAL," in *Models, Algorithms, Logics and Tools*, ser. LNCS, vol. 10460. Springer, 2017, pp. 249–268. [Online]. Available: https://doi.org/10.1007/978-3-319-63121-9_13

[26] B. Atanasovski, M. Bogdanovic, G. Velinov, L. Stoimenov, A. S. Dimovski, B. Koteska, D. Jankovic, I. Skrceska, M. Kon-Popovska, and B. Jakimovski, "On defining a model driven architecture for an enterprise

e-health system," *Enterp. Inf. Syst.*, vol. 12, no. 8-9, pp. 915–941, 2018. [Online]. Available: https://doi.org/10.1080/17517575.2018.1521996

[27] T. Dillig, I. Dillig, and S. Chaudhuri, "Optimal guard synthesis for memory safety," in *Computer Aided Verification - 26th International Conference, CAV 2014. Proceedings*, ser. LNCS, vol. 8559. Austria: Springer, 2014, pp. 491–507. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_32

[28] A. S. Dimovski, "On synthesizing presence conditions in numerical software product lines," in *Proceedings of the 29th ACM International Systems and Software Product Line Conference - Volume A, SPLC 2025*. ACM, 2025. [Online]. Available: https://doi.org/10.1145/3744915.3748474

[29] A. Albarghouthi, I. Dillig, and A. Gurfinkel, "Maximal specification synthesis," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*. St. Petersburg: ACM, 2016, pp. 789–801. [Online]. Available: https://doi.org/10.1145/2837614.2837628

[30] A. S. Dimovski, "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, 2025, pp. 1990–1997. [Online]. Available: https://doi.org/10.1145/3672608.3707849

[31] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1993, pp. 46–55. [Online]. Available: https://doi.org/10.1145/155090.155095

[32] A. Miné, "Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions," *Sci. Comput. Program.*, vol. 93, pp. 154–182, 2014. [Online]. Available: https://doi.org/10.1016/j.scico.2013.09.014

[33] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The astreé analyzer," in *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings*, ser. LNCS, vol. 3444. Springer, 2005, pp. 21–30. [Online]. Available: https://doi.org/10.1007/978-3-540-31987-0_3

[34] M. Greitschus, D. Dietsch, M. Heizmann, A. Nutz, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski, "Ultimate taipan: Trace abstraction and abstract interpretation - (competition contribution)," in *23rd International Conference, TACAS 2017, Proceedings, Part II*, ser. LNCS, vol. 10206, 2017, pp. 399–403. [Online]. Available: https://doi.org/10.1007/978-3-662-54580-5_31

[35] J. Henry, D. Monniaux, and M. Moy, "PAGAI: A path sensitive static analyser," *Electron. Notes Theor. Comput. Sci.*, vol. 289, pp. 15–25, 2012. [Online]. Available: https://doi.org/10.1016/j.entcs.2012.11.003

[36] X. Rival, "Understanding the origin of alarms in astrée," in *Static Analysis, 12th International Symposium, SAS 2005, Proceedings*, ser. LNCS, vol. 3672. Springer, 2005, pp. 303–319. [Online]. Available: https://doi.org/10.1007/11547662_21

[37] A. S. Dimovski and A. Legay, "Computing program reliability using forward-backward precondition analysis and model counting," in *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Proceedings*, ser. LNCS, vol. 12076. Springer, 2020, pp. 182–202. [Online]. Available: https://doi.org/10.1007/978-3-030-45234-6_9

[38] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design, FMCAD 2013*. IEEE, 2013, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/6679385/

[39] Y. Gu, T. Tsukada, and H. Unno, "Optimal CHC solving via termination proofs," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 604–631, 2023. [Online]. Available: https://doi.org/10.1145/3571214

[40] T. A. Beyene, C. Popeea, and A. Rybalchenko, "Solving existentially quantified horn clauses," in *Computer Aided Verification - 25th International Conference, CAV 2013. Proceedings*, ser. LNCS, vol. 8044. Springer, 2013, pp. 869–882. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_61

[41] T. Kuwahara, R. Sato, H. Unno, and N. Kobayashi, "Predicate abstraction and CEGAR for disproving termination of higher-order functional programs," in *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II*, ser. LNCS, vol. 9207. Springer, 2015, pp. 287–303. [Online]. Available: https://doi.org/10.1007/978-3-319-21668-3_17

[42] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I*, ser. LNCS, vol. 9206. Springer, 2015, pp. 343–361. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_20

[43] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf, "Jayhorn: A framework for verifying java programs," in *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part I*, ser. LNCS, vol. 9779. Springer, 2016, pp. 352–358. [Online]. Available: https://doi.org/10.1007/978-3-319-41528-4_19

[44] K. Hashimoto and H. Unno, "Refinement type inference via horn constraint optimization," in *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, ser. Lecture Notes in Computer Science, vol. 9291. Springer, 2015, pp. 199–216. [Online]. Available: https://doi.org/10.1007/978-3-662-48288-9_12

[45] J. Leroux, P. Rümmer, and P. Subotic, "Guiding craig interpolation with domain-specific abstractions," *Acta Informatica*, vol. 53, no. 4, pp. 387–424, 2016. [Online]. Available: https://doi.org/10.1007/s00236-015-0236-z

[46] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko, "A constraint-based approach to solving games on infinite graphs," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*. ACM, 2014, pp. 221–234. [Online]. Available: https://doi.org/10.1145/2535838.2535860

[47] J. Kim, Q. Hu, L. D'Antoni, and T. W. Reps, "Semantics-guided synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–32, 2021. [Online]. Available: https://doi.org/10.1145/3434311

[48] K. J. C. Johnson, A. Reynolds, T. W. Reps, and L. D'Antoni, "The semgus toolkit," in *Computer Aided Verification - 36th International Conference, CAV 2024, Proceedings, Part III*, ser. LNCS, vol. 14683. Springer, 2024, pp. 27–40. [Online]. Available: https://doi.org/10.1007/978-3-031-65633-0_2

[49] S. So and H. Oh, "Synthesizing imperative programs from examples guided by static analysis," in *Static Analysis - 24th International Symposium, SAS 2017, Proceedings*, ser. LNCS, vol. 10422. Springer, 2017, pp. 364–381. [Online]. Available: https://doi.org/10.1007/978-3-319-66706-5_18

[50] A. S. Dimovski, "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 2025, Bergen, Norway, July 3-4, 2025*. ACM, 2025, pp. 27–40. [Online]. Available: https://doi.org/10.1145/3742876.3742884

[51] A. Dimovski, "Mutation-based lifted repair of software product lines," in *38th European Conference on Object-Oriented Programming, ECOOP 2024*, ser. LIPIcs, vol. 313. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 12:1–12:24. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2024.12