Automated Program Analysis: Revisiting Precondition Inference through Constraint Acquisition

Grégoire Menguy\textsuperscript{1}, Sébastien Bardin\textsuperscript{1}, Nadjib Lazaar\textsuperscript{2} and Arnaud Gotlieb\textsuperscript{3}

\textsuperscript{1}Université Paris-Saclay, CEA, List, Palaiseau, France
\textsuperscript{2}LIRMM, University of Montpellier, CNRS, Montpellier, France
\textsuperscript{3}Simula Research Laboratory, Oslo, Norway

\{gregoire.menguy, sebastien.bardin\}@cea.fr, nadjib.lazaar@lirmm.fr, arnaud@simula.no

Abstract

Program annotations under the form of function pre/postconditions are crucial for many software engineering and program verification applications. Unfortunately, such annotations are rarely available and must be retrofit by hand. In this paper, we explore how Constraint Acquisition (CA), a learning framework from Constraint Programming, can be leveraged to automatically infer program preconditions in a black-box manner, from input-output observations. We propose \textsc{PreCA}, the first ever framework based on active constraint acquisition dedicated to infer memory-related preconditions. \textsc{PreCA} overpasses prior techniques based on program analysis and formal methods, offering well-identified guarantees and returning more precise results in practice.

1 Introduction

Program annotations under the form of function pre/postconditions [Hoare, 1969; Floyd, 1993; Dijkstra, 1968] are crucial for the development of correct-by-construction systems [Meyer, 1988; Burdy \textit{et al.}, 2005] or program refactoring [Ernst \textit{et al.}, 2001]. They can benefit both a human or an automated program analyzer, typically in software verification where they enable scalable (modular) analysis [Kirchner \textit{et al.}, 2015; Godefroid \textit{et al.}, 2011]. Unfortunately, annotations are rarely available and must be retrofit by hand into the code, limiting their interest – especially for black-box third-party components.

\textbf{Problem.} Efforts have been devoted to \textit{automatically infer} preconditions from the code, and contract inference is now a hot topic in Program Analysis and Formal Methods [Cousot \textit{et al.}, 2013; Ernst \textit{et al.}, 2001; Padhi \textit{et al.}, 2016; Astorga \textit{et al.}, 2018; Gehr \textit{et al.}, 2015]. Since this problem is undecidable (as most program analysis problems), the goal is to design principled methods with good practical results. Yet, the state-of-the-art is still not satisfactory. While \textit{white-box approaches} leveraging standard static analysis [Hoare, 1969; Floyd, 1993; Dijkstra, 1968; Cousot \textit{et al.}, 2013] can be helpful, they quickly suffer from precision or scalability issues, have a hard time dealing with complex programming features (loops, recursion, dynamic memory) and cannot cope with black-box components. On the other hand \textit{black-box methods}, leveraging test cases to dynamically infer (likely) function contracts [Ernst \textit{et al.}, 2001; Padhi \textit{et al.}, 2016; Gehr \textit{et al.}, 2015], overcome static analysis limitations on complex codes and have drawn attention from the software engineering community [Zhang \textit{et al.}, 2014]. Yet, they heavily depend on the quality of the underlying test cases, which are often simply generated at random, given by the users [Ernst \textit{et al.}, 2001] (passive learning), or automatically generated during the learning process – but without any clear coupling between sampling and learning [Padhi \textit{et al.}, 2016; Gehr \textit{et al.}, 2015] – and so, show no clear guarantee on the inference process.

\textbf{Constraint Acquisition.} Constraint programming (CP) [Rossi \textit{et al.}, 2006] has made considerable progress over the last forty years, becoming a powerful paradigm for modelling and solving combinatorial problems. However, modelling a problem as a constraint network still remains a challenging task that requires expertise in the field. Several constraint acquisition (CA) systems have been introduced to support the uptake of constraint technology by non-experts. Especially, rooted in version space learning, \textsc{ConACQ} is presented in its passive and active versions [Bessiere \textit{et al.}, 2017]. Based on solutions and non-solutions labelled by the user (acting as an oracle), the system learns a set of constraints that correctly classifies all examples given so far. This is an active field of research, with many proposed extensions, for example allowing partial queries [Bessiere \textit{et al.}, 2013]. However, even though \textsc{ConACQ} enjoys strong theoretical foundations, such CA systems are hard to put in practice, as they require to submit \textit{thousands of queries to a user}. In automated program analysis, the huge number of queries is not a problem as long as a program plays the oracle.

\textbf{Goal and contributions.} In this paper, we explore the potential of Constraint Acquisition for \textit{black-box precondition inference}. To the best of our knowledge, this is the first application of CA to program analysis and our overall results show its potential there. Our main contributions are the following:

- We propose \textsc{PreCA}, the first ever (\textsc{ConACQ}-like) framework based on active constraint acquisition and dedicated to infer preconditions (Section 4). We show in Section 4.3 that \textsc{PreCA} enjoys much better theoretical correctness properties than prior black-box approaches.
Indeed, if our learning language is expressive enough, PRECA is guaranteed to infer the weakest precondition:

- We describe a specialization of PRECA to the important case of memory-related preconditions (Section 5). Especially, we propose a dedicated constraint language (including memory constraints) for the problem at hand, as well as domain-based strategies to make the approach more efficient in practice (Section 5.2);

- We experimentally evaluate the benefits of our technique on several benchmark functions (Section 6.1). The results show that PRECA significantly outperforms prior precondition learners, be it black-boxes or white-boxes – which came as a surprise. For example, PRECA with 5s budget per sample performs better than prior approaches with 1h per sample.

Overall, it turns out that seeing the precondition inference problem as a Constraint Acquisition task is beneficial, leading to good theoretical properties and beating prior techniques.

2 Background

2.1 Preconditions and Weakest Preconditions

A program function, or simply a function, $F : \text{In} \to \text{Out}$ can be seen as a partial mapping from inputs to outputs. Given an input $x \in \text{In}$, execution of $F$ over $x$ can: (i) terminate and return $y \in \text{Out}$, noted $F(x) = y$; (ii) diverge (i.e., never terminate) or raise a runtime error, in that case $F$ is not defined over $x$. Given a function $F$ and a predicate $Q$ over $F$ outputs called a postcondition, Hoare logic [Hoare, 1969] defines the precondition (a predicate over $F$ inputs) of $F$ w.r.t. $Q$.

**Definition 1** (Precondition). Given a function $F$ and a postcondition $Q$, $P$ is a precondition of $F$ w.r.t. $Q$ iff for all $x$ s.t. $x \models P$, $F(x) = y$ and $y \models Q$, noted $\{P\} F \{Q\}$.

A function $F$ can have several preconditions for a given postcondition $Q$. Still, not all preconditions are useful, some being too restrictive. Thus, we aim for the most generic one, called the *weakest precondition* (WP) [Hoare, 1969]. Automatically computing the weakest precondition of $F$ w.r.t. $Q$ has been a strong drive for program analysis since the 70’s. Yet, as the whole problem is undecidable, standard approaches must rely on manual annotations or approximations.

**Definition 2** (Weakest precondition). Let a function $F$ and a postcondition $Q$. The weakest precondition of $F$ w.r.t. $Q$ noted $\text{WP}(F, Q)$ is the most generic precondition i.e. for all $P$ s.t. $\{P\} F \{Q\}$, $P \Rightarrow \text{WP}(F, Q)$.

**Example 1.** Let $\text{int} \ \text{div}(\text{int} \ a) \ \{\text{return} \ 100/a\}$ the function under analysis. Here, $\text{In} = \text{Out} = [-2^{n-1}, 2^{n-1}-1]$ with $n$ the size of an int. Note that it is undefined when $a = 0$. Hence, for postcondition $Q_1 = \text{true}$, a precondition could be $a = 5$. However, it is too restrictive as other values of $a$ can return safely. The less restrictive one, i.e. $\text{WP}(F, Q_1)$, is $a \neq 0$. Now, consider the postcondition $Q_2 = \text{the return value must be } \geq 0$. Then $\text{WP}(F, Q_2)$ is $a > 0$.

2.2 Constraint Acquisition

The constraint acquisition (CA) process can be seen as an interplay between the learner and the user. For that, the learner needs to share some common *vocabulary* to communicate with the user. This vocabulary is a finite set of variables $X$ taking values in a finite domain $D$. A constraint $c$ is defined on a subset of variables and a relation specifying which values are allowed. A constraint network is a set $C$ of constraints. An example $e \in D^X$ satisfies a constraint $c$ if the projection of $e$ on $c$ variables is in $c$. An example $e$ is a solution of $C$ if and only if it satisfies all constraint in $C$.

In addition to the vocabulary, the learner owns a *language* $\Gamma$ of bounded arity relations from which it can build constraints on specified sets of variables. The *constraint bias*, denoted by $B$, is a set of constraints built from $\Gamma$ on $(X, D)$, from which the learner builds a constraint network. A concept is a Boolean function $f$ over $D^X$. A representation of a concept $f$ is a constraint network $C$ for which $f^{-1}(\text{true})$ equals the solutions set of $C$. A membership query takes an example $e$ and asks the user to classify it. The answer is *yes* iff $e$ is a solution of the user concept. For any example $e, \kappa(e)$ denotes the set of all constraints in $B$ rejecting $e$.

We now define convergence. Given a set $E$ of examples labelled by the user *yes* or *no*, we say that a network $C$ agrees with $E$ if $C$ accepts all examples labelled *yes* in $E$ and does not accept those labelled *no*. The learning process has converged on the network $L \subseteq B$ if (i) $L$ agrees with $E$ and (ii) for every other $L' \subseteq B$ agreeing with $E$, we have $L' \equiv L$.

CONACQ is a CA system that submits membership queries to a user. CONACQ uses a concise representation of the learner’s version space into a clausal formula. Formally, any constraint $c \in B$ is associated with a Boolean atom $a(c)$ stating if $c$ must be in the learned network. CONACQ starts with an empty theory and iteratively expands it by generating and submitting to the user an informative example. An informative example ensures to reduce the learner’s version space independently from the user answer. If no informative example remains, this means that we converged and CONACQ returns the theory encoding the learned network.

3 Motivation

We focus on memory-related preconditions – e.g., predicates stating on which inputs a function can be executed without leading to a memory violation – in a black-box manner. Let us consider the prototype of function `find_first_of` in Listing 1 (from Frama-C [Kirchner et al., 2015] test suite). We aim to infer which values of $a, m, b$ and $n$ are accepted without relying on the source code – still we can execute the code over chosen input and observe results.

```c
void find_first_of (int *a, int m, int *b, int n)
```

Listing 1: Function prototype

From white-box to black-box. White-box analysis (such as P-Gen [Seghir and Kroening, 2013]) uses the program source code to infer preconditions. Yet several practical scenarios are impractical for white-box methods. First, having the whole source code is often unrealistic (many projects embed third-party components). Second, in practice program analyzers focus on a single programming language, but many projects use combinations of them (e.g., inline assembly in C code). Third, despite huge progress in the past decades,
white-box program analysis still suffers on large or complex codes (unbounded loops, recursion, dynamic allocation, etc.) possibly leading to serious scalability or precision issues. Fourth, obfuscation is common in certain ecosystems to make reverse engineering harder and thwart white-box analysis. In all these scenarios, black-box methods are the sole option (cf. experiments on Section 6, RQ4). Yet, as generalization is involved, black-box methods can compute incorrect preconditions (i.e., formula actually being not preconditions). Black-box passive learning is not enough. Black-box methods should exercise the function under analysis on a representative set of test cases to infer relevant preconditions. A solution is to assume that users can provide such tests and leverage passive learning. Yet, this is often unrealistic—especially when the source code is not available. Moreover, random testing is rarely satisfactory, e.g., with 100 random test cases, both Daikon [Ernst et al., 2001] and PIE [Padhi et al., 2016] infer here an incorrect precondition for find_first_of.

Active learning. Gehr et al. [Gehr et al., 2015] performs active learning, generating test cases automatically. Such approaches are more actionable and less sensitive to user bias. Still, methods developed so far lack theoretical guarantees. Indeed, they cannot ensure that all useful test cases have been considered. Gehr et al. method infers in ≈ 700s an incorrect precondition for find_first_of, generating 177 test cases.

PRECA insights. Our method performs black-box precondition inference through active constraint acquisition [Bessiere et al., 2017]. Unlike previous active approaches, PRECA mixes the sampling and learning phases which enables to show good theoretical properties. Indeed, when a test case is generated, PRECA directly observes how the function behaves on it and updates its search space accordingly. As such, given a set of constraints B called the bias, PRECA will generate all test cases to ensure convergence modulo B. Thus, if all queries can be exactly classified and if B is expressive enough, PRECA returns the weakest precondition. Regarding our example, it infers the (correct) weakest precondition \((m > 0 \Rightarrow valid(a)) \land (m > 0 \land n > 0 \Rightarrow valid(b))\), where valid(p) \(\equiv (p \neq NULL)\), in 172s, with 45 test cases.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Active?</th>
<th>Success.</th>
<th>#Test cases</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daikon</td>
<td>no</td>
<td>no</td>
<td>100</td>
<td>0.6s</td>
</tr>
<tr>
<td>PIE</td>
<td>no</td>
<td>no</td>
<td>100</td>
<td>11s</td>
</tr>
<tr>
<td>Gehr et al.</td>
<td>yes</td>
<td>no</td>
<td>177</td>
<td>700s</td>
</tr>
<tr>
<td>P-Gen</td>
<td>(white-box)</td>
<td>(do not apply)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PRECA</td>
<td>yes</td>
<td>yes</td>
<td>45</td>
<td>172s</td>
</tr>
</tbody>
</table>

Table 1: find_first_of results, no source code

4 Precondition Acquisition

Given a function under analysis \(F\), we aim to infer the weakest precondition of \(F\) w.r.t. some postcondition \(Q\) through CA. Note that, as generalization is involved, we are not sure \textit{a priori} to compute a real precondition, hence the wording "likely-precondition" introduced in Daikon [Ernst et al., 2001]. Guarantees are studied in Section 4.3. To our knowledge, this is the first time CA is used for program analysis.

Algorithm 1: PRECA

\[
\begin{array}{l}
\text{In} & : \text{A function } F; \text{ a postcondition } Q; \text{ a bias } B; \text{ variables } X; \\
\text{Out} & : \text{A constraint network over } F \text{ input encoded by } \Omega \text{ consistent with oracle answers or collapse;}
\end{array}
\]

\begin{algorithm}
\begin{algorithmic}[1]
\STATE \(\Omega \leftarrow \top\)
\WHILE{true}
\STATE \(e \leftarrow \text{QueryGeneration}(B, X, \Omega)\)
\IF{\(e = \text{nil}\)}
\IF{\(\Omega \text{ is SAT}\)}
\STATE \(\text{return network}(\Omega)\)
\ELSE \text{return "collapse"}
\ENDIF
\ENDIF
\STATE \(\Omega \leftarrow \Omega \land \bigvee_{c \in \kappa(e)} a(c)\)
\ENDIF
\ENDWHILE
\end{algorithmic}
\end{algorithm}

4.1 Problem at Hand

We show here that precondition inference can be translated to a CA problem. In our context the user is replaced by an oracle, which automatically answers queries—implemented in practice by running the program on given inputs—and the target concept is \(WP(F, Q)\). The set of variables \(X\) equals \(M\) where \(M\) is the initial memory state to run \(F\). \(M\) is a map from symbols—like \(F\) arguments and global variables—to their values and the domain \(D\) of \(M\) is the finite set of all its possible mappings—thus, \(D^X\) equals \(In\), the definition domain of \(F\). The constraint language \(\Gamma\) and the bias \(B\) are sets of constraints over \(M\). We describe precisely \(\Gamma\) and \(B\) in Section 5.1. Finally, a membership query \(e\) is a complete assignment of \(M\) s.t. \(F\) can be executed over \(e\).

4.2 Description of PRECA

We detail here our approach, dubbed PRECA, composed of

1. the oracle; 2. the acquisition module.

Oracle. Given a function \(F\) and a postcondition \(Q\), PRECA queries an oracle to classify membership queries. It takes \(F\), \(Q\) and an input \(e \in In = D^X\) and answers in finite time. The oracle must comply to the following specification:

\[
\text{runOracle}(F, Q, e) = \begin{cases} 
\text{yes or unk} & \text{if } e \models WP(F, Q) \\
\text{no or unk} & \text{otherwise}
\end{cases}
\]

Note that the oracle answers \(unk\) when it cannot classify \(e\), extending the CONACQ framework where the user must answer only by \(yes\) or \(no\). In practice, such oracle runs \(F\) over \(e\) with a timeout. If the execution timeouts it returns (i) \(unk\), otherwise it returns (ii) \(yes\) if \(F(e) = y\) and \(y \models Q\); (iii) \(no\) if \(F(e) = y\) and \(y \not\models Q\) or if execution raises a runtime error.

Acquisition module. PRECA (see Algo.1) starts from an empty theory \(\Omega\) and iteratively expands it by processing examples generated at line 4. PRECA submits these examples to the oracle for a classification (runOracle call at line 9). If the oracle answers \textit{yes}, we must discard all constraints of \(B\) in \(\kappa(e)\), those rejecting \(e\), by expanding \(\Omega\) with negative
4.3 Theoretical Analysis

We show that PRECA terminates and that learned preconditions are sound when PRECA is fed with an expressive enough bias \( B \). Then we show that if \texttt{runOracle} never answers \( \texttt{unk} \), PRECA returns the weakest precondition.

**Proposition 1** (Consistency). Given a function \( F \), a postcondition \( Q \) and a bias \( B \). If PRECA returns a network \( L \), then \( L \) agrees with all positive and negative queries.

*Proof.* (sketch.) \( L \) discards all constraints of \( \kappa(e) \) when \( e \) is a positive and learns at least one constraint from \( \kappa(c) \) when \( e \) is a negative. It follows that the returned network \( L \) agrees with all examples given so far.

**Proposition 2** (Termination). Given a function \( F \), a postcondition \( Q \) and a bias \( B \), PRECA terminates.

*Proof.* (sketch.) Termination of PRECA immediately follows the reduction of \( \Omega \) to a monomial with an atom for each constraint \( c \in B \). As (i) \( \Omega \) involves a finite number of atoms (\( B \) being a finite set of constraints), (ii) \texttt{QueryGeneration} terminates returning an informative example if it exists, \texttt{nil} otherwise (Lemma 2 in [Bessiere et al., 2017]), and (iii) \texttt{runOracle} always responds, we have termination.

**Proposition 3** (Soundness). Given a function \( F \), a postcondition \( Q \) and a bias \( B \) s.t. \( \text{WP}(F, Q) \) is representable by \( B \). If PRECA returns a network \( L \) then \( L \) is a precondition of \( F \) w.r.t. the postcondition \( Q \).

*Proof.* (sketch.) We aim to prove that \( L \vdash \text{WP}(F, Q) \). As \( \text{WP}(F, Q) \subseteq B \) and we returned \( L \), there exists no example \( e \) s.t. \( e \models L \) and \( e \not\models \text{WP}(F, Q) \).

**Theorem 1** (Correctness). Given a function \( F \), a postcondition \( Q \) and a bias \( B \) s.t. \( \text{WP}(F, Q) \) is representable by \( B \). If \texttt{runOracle} never returns \texttt{unk} then PRECA converges to a network \( L \) equivalent to the weakest precondition.

*Proof.* If \( \text{WP}(F, Q) \subseteq B \) and \texttt{runOracle} returns \texttt{yes/no} answers, PRECA is equivalent to CONACQ. CONACQ is correct, terminates and always converges when \( B \) is expressive enough [Bessiere et al., 2017], it follows that PRECA always converges on to a constraint network \( L \) equivalent to \( \text{WP}(F, Q) \) under the assumptions on \( B \) and \texttt{runOracle}.

**Discussion.** These guarantees, while not perfect, are still very pleasant for a black-box approach. Prior black-box learners are much more limited: Daikon [Ernst et al., 2001] does not guarantee consistency (Proposition 1), while [Padhi et al., 2016; Gehr et al., 2015] guarantee consistency but not correctness (Theorem 1). Also, previous black-box methods consider that functions always terminate i.e., no \texttt{unk} answers.

5 PRECA for Memory-oriented Preconditions

We now setup PRECA to the case of memory-related preconditions, which are of paramount importance for the safety and security of low-level languages like C or binary code.

5.1 Constraint Acquisition Settings

**Vocabulary \((X, D)\).** Given a function \( F \), our variables set \( X = \{ p_1, \ldots, p_k, i_1, \ldots, i_l \} \) represents the initial memory state of \( F \). It is composed of all \( F \) arguments and global variables in scope. Here, \( p_j \) are pointers and \( i_j \) are integers (signed or not). \( D^X \) defines possible \( F \) inputs. It compactly represents all cases induced by \( \Gamma \). We note \( r_1, \ldots, r_m \) the address of each global variables in \( X \) and \( a_1, \ldots, a_k \), \( k \) pairwise distinct new valid addresses. Then, \( D(p_i) \) is \( \{ \text{NULL}, r_1, \ldots, r_m, a_1, \ldots, a_k \} \) and \( D(i_j) \) is \( \{ 0, N_U \} \) if \( i_j \) is unsigned and \( \{ -N_l, N_l \} \) otherwise -- \( N_l \) and \( N_U \) are the number of signed and unsigned integers in \( X \).

**Language \( \Gamma \).** PRECA considers the constraint language \( \Gamma \) described in Section 5.1 including well-typed constraints only. Observe that: (i) it does not include conjunctions of constraints as acquisition will infer them; (ii) \( \Gamma \) holds Horn clauses of arbitrary size which is crucial to handle conditional preconditions, e.g., \texttt{find_first_of weakest precondition} in Listing 1 contains the constraint \( m > 0 \Rightarrow \text{valid}(a) \).

<table>
<thead>
<tr>
<th>Grammar</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( C \Rightarrow A \land \sim A )</td>
</tr>
<tr>
<td>( A )</td>
<td>( \text{valid}(p_j) \land \text{alias}(p_j, p_i) \lor \text{deref}(p_j, g) \land i_j = 0 \land i_j &lt; 0 \land i_j = i_i \land i_j &lt; i_i \land i_j \leq i_k )</td>
</tr>
</tbody>
</table>

Table 2: Grammar of constraint language \( \Gamma \)

**Bias \( B \).** The bias \( B \) is a finite set of constraints extracted from \( \Gamma \). A balance must be found here, as a large bias is more expressive but can slow down inference. Given the function \( F \), PRECA considers the following heuristic: “Let \( i \) be the number of \( F \) integer inputs and \( k = \max(i, 1) \). Then PRECA bias includes all Horn clauses of size \( \leq k + 1 \) from \( \Gamma \)”.

5.2 Speeding up PRECA

First, we describe PRECA background knowledge. Secondly, we present a domain-based preprocessing heuristic.
Background knowledge. A background knowledge $K$ to speed up convergence of CA contains known relations over the bias constraints to filter incoherent networks. Table 3 shows a subset of $K$. It contains usual boolean properties, transitivity relations over integers and relations on memory – e.g., if $p_1$ is valid and $p_1$ aliases with $p_2$ then $p_2$ is valid.

$$a(e) \rightarrow \neg a(e), \forall e \in B$$
$$a(c_1) \rightarrow a(c_1 \lor c_2), \forall c_1, c_2 \in B$$
$$a(i = 0) \land a(i = i_2) \rightarrow a(i_2 = 0)$$
$$a(i = i_2) \land a(i_2 = i_3) \rightarrow a(i = i_3)$$
$$a(\neg valid(p_1)) \land a(\neg alias(p_1, p_2)) \rightarrow a(valid(p_2))$$
$$a(valid(p_1)) \land a(alias(p_1, p_2)) \rightarrow a(valid(p_2))$$
$$a(alias(p_1, p_2)) \land a(alias(p_2, p_3)) \rightarrow a(alias(p_1, p_3))$$

Where $p_j$ (resp. $i_j$) are pointer (resp. integer) variables.

Table 3: Background knowledge $K$ (a subset)

Preprocess. Functions rarely raise runtime errors or contradict postconditions over valid and non aliasing pointers (i.e., the easy case that programmers usually handle well). Thus, given a function $F$, we call likely-positive queries assignments of $F$ inputs s.t. at most one $p_j$ is invalid or at most one pair $(p_j, p_k)$ aliases. Over likely-positive queries, the oracle will probably answer yes, which would be really helpful as it would discard all constraints from $\kappa(e)$ (unlike negative ones which introduce non-unit clause in $\Omega$, see Algorithm 1). Thus, PRECA starts by likely-positive queries in the hope to prune the search space before launching the active phase.

6 Experimental Evaluation

We implemented PRECA\(^1\) in JAVA, and rely on the CHOCO constraint solver and MINISAT SAT solver. We evaluate PRECA on the following Research Questions:

RQ1 Can PRECA handle realistic functions? We launch PRECA against our benchmark and check if it indeed infers weakest preconditions;

RQ2 How PRECA components influence results? We compare PRECA with and without background knowledge, preprocess and active learning;

RQ3 Is PRECA competitive with black-box methods? We compare to black-box state-of-the-art methods in terms of correctness and speed.

RQ4 Is PRECA competitive with white-box methods? We compare to the white-box method P-Gen on clean C code, and consider also 3 “hard” scenarios: no source code, obfuscated code, presence of inline assembly.

6.1 Experimental Design

Benchmark. Our benchmark considers 50 real C functions. It contains all functions from string.h, all functions from [Seghir and Kroening, 2013; Sankaranarayanan et al., 2008] (except 2 functions from an old Xen version), functions from the DSA benchmark (https://tinyurl.com/tvzzpvmn), Framac-C WP test suite (https://tinyurl.com/yx3bfj33), Siemens suite [Hutchins et al., 1994] and the book Science of Programming [Gries, 2012]. Functions range from 3 LoC to 250 (mean 59), have up to 9 loops (mean 2.8, 47/50 functions with loops) and 2/50 with recursive calls.

Postconditions. For each function, we study two scenarios: with the implicit true postcondition (dubbed "no postcondition") and with explicit postcondition. In the latter case, we manually choose relevant ones, e.g. $Q = valid(\text{ret})$ for pointers, and $Q = \text{ret} \neq 0$ or $Q = \text{ret} > 0$ for integers. Finally, six functions returns no output and are discarded. In total, our benchmark contains 94 inference tasks, 50 with the implicit postconditions and 44 with explicit postconditions.

Setup. We run PRECA with different time budgets per function (from 1s to 1h) and an oracle timeout of 1min (leading to the unknown answers). Experiments are done on a machine with 6 Intel Xeon E-2176M CPUs and 32 GB of RAM.

6.2 Experimental Results

Results are summarized in Section 6.2.

RQ1. With a time budget of 5min per example and without postcondition, PRECA infers 46/50 weakest preconditions (29/50 for 1s, 38/50 for 5s). Two examples timeout, and two others return a constraint network not equivalent to the weakest precondition – a manual inspection shows our bias is not expressive enough in these cases, still it returns a (correct) precondition for one of them. With postconditions, PRECA infers 18/44 weakest preconditions with $< 5$ min time budget each (11/44 for 1s, 16/44 for 5s) and never timeouts (in 7 other cases it still infers a correct precondition). These results are far better than other state-of-the-art tools (RQ3, RQ4).

PRECA is able to handle real functions precisely (weakest precondition) in a small amount of time. Especially, it is extremely accurate for implicit postconditions.

RQ2. First, we consider PRECA in passive mode, with 100 random queries, in order to see the impact of active learning (denoted Random in Section 6.2). Results are averaged over 10 runs per function. We see a significant drop in performance for time budgets $\geq 5$min (for 5min: 30/50 vs 46/50, 18/44 vs 12/44). Second, we study how the background knowledge and the preprocess impact PRECA results. We see a clear impact only for small time budgets (e.g., 1s and no postcondition: 29 vs 15/19/13). Interestingly, both the background knowledge and the preprocess are necessary to get speedup.

PRECA benefits strongly from its active mode. Background knowledge and preprocess over complex preconditions are useful for small time budgets.

RQ3. We compare now against state-of-the art black-box precondition learners, namely Daikon [Ernst et al., 2001], PIE [Padhi et al., 2016] and Gehr et al. approach [Gehr et al., 2015] – code being unavailable, we reimplemented it. Daikon and PIE performing passive learning, we run them over 100 random queries. As Daikon, PIE and Gehr et al. methods are randomized, we run them 10× and report their average results. We first observe that PRECA performs significantly better than these three competitors for all setups – for 1s and no postcondition: 29 vs 8.0 - 16.0 - 1.4; for 1h and no postcondition: 46 vs 26.1 - 17.7 - 1.6. We tried feeding Daikon, PIE and Gehr et al. with PRECA queries (lines

\(^{1}\)https://github.com/binsec/preca
PRECA significantly outperforms prior black-box methods. Especially, it infers in 5× more weakest preconditions than Daikon, PIE and Gehr et al. in 1h. Moreover, it generates high quality queries that can benefit other methods.

RQ4. We compared to the white-box method P-Gen [Seghir and Kroening, 2013]. We also considered [Kafle et al., 2018] and [Gulwani et al., 2008], but the former requires to manually translate C code to Prolog (no front-end provided) and the latter is not available. First, we consider a favourable setup where the source code of our 94 examples is available (Section 6.2). Surprisingly, PRECA infers slightly more WP with a 5s time budget than P-Gen with 1h (both with and without postcondition). The gap increases for a time budget of 1h and implicit postconditions (46 vs 37). Second, we consider “hard” application scenarios: (i) no source code; (ii) obfuscated code; (iii) inline assembly – our 94 samples are transformed accordingly. As expected for a white-box method, P-Gen infers no precondition for these scenarios (0/94) while PRECA results remain the same. As expected, PRECA significantly outperforms P-Gen on hard application scenarios. Less expected, it performs also better in the case where the source code is fully available.

As expected, PRECA significantly outperforms P-Gen on hard application scenarios. Less expected, it performs also better in the case where the source code is fully available.

7 Related Work

Black-box contracts inference. Daikon [Ernst et al., 2001] dynamically infers preconditions through predefined patterns over the evolution of variable values. The technique is passive and lacks clear foundations. PIE [Padhi et al., 2016] relies on program synthesis for black-box precondition inference. Garg et al. [Garg et al., 2016] and Sankaranarayanan et al. [Sankaranarayanan et al., 2008] infer invariants and preconditions through tree learning algorithms. As invariant inference distinguishes from precondition inference, we did not consider [Garg et al., 2016] in our evaluation. However, even if [Sankaranarayanan et al., 2008] method was not available, we integrated their use-cases and show that we handle them all (except one) while enjoying better theoretical properties. These methods perform passive learning and heavily depend on test cases quality. Gehr et al.'s method [Gehr et al., 2015] relies on black-box active learning. Yet, it relies on program synthesis and performs (type-aware) random sampling, preventing it to enjoy PRECA correctness properties.

White-box dynamic contracts inference. While purely static white-box approaches [Cousot et al., 2013; Calcagno et al., 2009; Gulwani et al., 2008; Kafle et al., 2018] are considered imprecise (too conservative) and hard to get right (loops, memory, etc.), some approaches combine dynamic reasoning together with white-box information. Seghir et al. [Seghir and Kroening, 2013] method must translate the analyzed function into transition constraints being thus highly impacted by code complexity (Section 6.2 RQ4). On the other hand, Astorga et al. [Astorga et al., 2018; Astorga et al., 2019] relies on symbolic execution to retrieve a set of useful inputs and language features, yet the technique is incomplete in the presence of loops and cannot ensure that all interesting test cases were tested.

Constraint acquisition. CA has been applied to different contexts from scheduling [Beldiceanu and Simion, 2012] to robotics [Paulin et al., 2008]. However, this is the first time CA is applied to program analysis and precondition inference. While we rely on CONACQ, other techniques exist [Beldiceanu and Simion, 2012; Lalouet et al., 2010; Tsouras et al., 2020] and could be explored.

Program synthesis. Program synthesis [Gulwani et al., 2017] aims at creating a function meeting a given specification, given either formally, in natural language or as input-output relations. This last case shows some similarities with precondition inference and is used in some prior work on black-box inference [Gehr et al., 2015; Padhi et al., 2016].

8 Conclusion

We propose the first application of Constraint Acquisition to the Precondition Inference problem, a major issue in Program Analysis and Formal Methods. We show how to instantiate the standard framework to the program analysis case, yielding the first black-box active precondition inference method with clear guarantees. Moreover, our experiments for memory-oriented preconditions show that PRECA significantly outperforms prior works, demonstrating the interest of Constraint Acquisition here.

6.3 Discussion

While PRECA shows overall good properties, it also comes with some limitations. First, handling constant values is problematic. Indeed, we should add comparisons to them in the bias. However, in a black-box context, there is no reason to choose one constant value from another and we cannot add all of them as bias would explode. Second, PRECA uses Horn clauses to handle disjunctive specifications. We consider a simple heuristic for size selection (Section 5.1), yet a more principled approach is desirable. Finally, we require the function under analysis to be deterministic (a common assumption in the field). Going further remains open.
Acknowledgments

This work has received funding from (i) the European Union’s Horizon 2020 research and innovation programme under grant agreement No 952215; (ii) the Agence National de la Recherche under grant ANR-20-CE25-0009-TAVA and (iii) the Research Council of Norway in the context of the T-Largo project under grant agreement 274786.

References


