Automated Program Analysis: Revisiting Precondition Inference through Constraint Acquisition

Grégoire Menguy[1] , **Sébastien Bardin**[1] , **Nadjib Lazaar**[2] and **Arnaud Gotlieb**[3]
[1]Université Paris-Saclay, CEA, List, Palaiseau, France
[2]LIRMM, University of Montpellier, CNRS, Montpellier, France
[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 PRECA, the first ever framework based on active constraint acquisition dedicated to infer memory-related preconditions. 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 *et al.*, 2005] or program refactoring [Ernst *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 *et al.*, 2015; Godefroid *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.

Problem. Efforts have been devoted to *automatically infer* preconditions from the code, and contract inference is now a hot topic in Program Analysis and Formal Methods [Cousot *et al.*, 2013; Ernst *et al.*, 2001; Padhi *et al.*, 2016; Astorga *et al.*, 2018; Gehr *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 *white-box approaches* leveraging standard static analysis [Hoare, 1969; Floyd, 1993; Dijkstra, 1968; Cousot *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 *black-box methods*, leveraging test cases to dynamically infer (likely) function contracts [Ernst *et al.*, 2001; Padhi *et al.*, 2016; Gehr *et al.*, 2015], overcome static analysis limitations on complex codes and have drawn attention from the software engineering community [Zhang *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 *et al.*, 2001] (passive learning), or automatically generated during the learning process – but without any clear coupling between sampling and learning [Padhi *et al.*, 2016; Gehr *et al.*, 2015] – and so, show no clear guarantee on the inference process.

Constraint Acquisition. Constraint programming (CP) [Rossi *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, CONACQ is presented in its passive and active versions [Bessiere *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 *et al.*, 2013]. However, even though CONACQ enjoys strong theoretical foundations, such CA systems are hard to put in practice, as they require to submit *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.

Goal and contributions. In this paper, we explore the potential of Constraint Acquisition for *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 PRECA, the first ever (CONACQ-like) framework based on active constraint acquisition and dedicated to infer preconditions (Section 4). We show in Section 4.3 that 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, $F : In \rightarrow Out$ can be seen as a partial mapping from inputs to outputs. Given an input $x \in In$, execution of F over x can: (i) terminate and return $y \in Out$, 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 $\mathcal{WP}(F,Q)$ is the most generic precondition i.e. for all P s.t. $\{P\}F\{Q\}$, $P \Rightarrow \mathcal{WP}(F,Q)$.*

Example 1. *Let* `int div(int a) {return 100/a}` *the function under analysis. Here, $In = Out = [\text{-}2^{n\text{-}1}; 2^{n\text{-}1}\text{-}1]$ with n the size of an* `int`*. Note that it is undefined when $a = 0$. Hence, for postcondition $Q_1 = true$, 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. $\mathcal{WP}(F,Q_1)$, is $a \neq 0$. Now, consider the postcondition $Q_2 =$ "the return value must be ≥ 0". Then $\mathcal{WP}(F,Q_2)$ 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* Γ 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 Γ 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 $f^{-1}(true)$ 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.

```
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 $\approx 700s$ 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* $(m > 0 \Rightarrow valid(a)) \wedge (m > 0 \wedge n > 0 \Rightarrow valid(b))$, where $valid(p) \equiv (p \neq NULL)$, in 172s, with 45 test cases.

Alg.	Active?	Success.	#Test cases	Time
Daikon	no	no	100	0.6s
PIE	no	no	100	11s
Gehr et al.	yes	no	177	700s
P-Gen	(white-box)	(do not apply)	-	-
PRECA	**yes**	**yes**	**45**	**172s**

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 *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

In : A function F; a postcondition Q; a bias B; variables X;
Out : A constraint network over F input encoded by Ω consistent with oracle answers or collapse;

1 **begin**
2 $\quad \Omega \leftarrow \top$
3 $\quad$ **while** $true$ **do**
4 $\quad\quad e \leftarrow$ `QueryGeneration`(B, X, Ω)
5 $\quad\quad$ **if** $e = nil$ **then**
6 $\quad\quad\quad$ **if** Ω *is SAT* **then**
7 $\quad\quad\quad\quad$ **return** `network`(Ω)
8 $\quad\quad\quad$ **else return** "$collapse$"
9 $\quad\quad$ **if** `runOracle`$(F, Q, e) \neq yes$ **then**
10 $\quad\quad\quad \Omega \leftarrow \Omega \wedge (\bigvee_{c \in \kappa(e)} a(c))$
11 $\quad\quad$ **else** $\Omega \leftarrow \Omega \wedge (\bigwedge_{c \in \kappa(e)} \neg a(c))$

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 $\mathcal{WP}(F, Q)$. 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* Γ and the *bias* B are sets of constraints over M. We describe precisely Γ 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:

$$\texttt{runOracle}(F, Q, e) = \begin{cases} yes \text{ or } ukn \text{ if } e \models \mathcal{WP}(F, Q) \\ no \text{ or } ukn \text{ otherwise} \end{cases}$$

Note that the oracle answers ukn 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) ukn, 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 Ω 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 yes, we must discard all constraints of B in $\kappa(e)$, those rejecting e, by expanding Ω with negative unit clauses (line 11). However, if the oracle answers no or ukn, Ω is expanded with a clause consisting of all literals $a(c)$ s.t. $c \in \kappa(e)$ (line 10). Bear in mind that `QueryGeneration` function returns informative examples aiming to reduce Ω to a monomial (conjunction of unit clauses). `QueryGeneration` is used exactly as it appears in [Bessiere *et al.*, 2017]. If there is no example to return, this means that Ω is monomial. Now if Ω is not satisfiable, a *"collapse"* message is returned (line 8). This happen when the concept to learn is not representable by B. Otherwise, we return the constraint network encoded by Ω through the `network` function (line 7).

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 `runOracle` never answers ukn, 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.) PRECA discards all constraints of $\kappa(e)$ when e is a positive and learns at least one constraint from $\kappa(e)$ 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 Ω to a monomial with an atom for each constraint $c \in B$. As (i) Ω involves a finite number of atoms (B being a finite set of constraints), (ii) `QueryGeneration` terminates returning an informative example if it exists, nil otherwise (Lemma 2 in [Bessiere *et al.*, 2017]), and (iii) `runOracle` always responds, we have termination. □

Proposition 3 (Soundness)**.** *Given a function F, a postcondition Q and a bias B s.t. $\mathcal{WP}(F, Q)$ 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 $L \Rightarrow \mathcal{WP}(F, Q)$. As $\mathcal{WP}(F, Q) \subseteq B$ and we returned L, there exists no example e s.t. $e \models L$ and $e \not\models \mathcal{WP}(F, Q)$. □

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

Proof. If $\mathcal{WP}(F, Q) \subseteq B$ and `runOracle` returns 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 $\mathcal{WP}(F, Q)$ under the assumptions on B and `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 ukn 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 $X = \{p_1, \ldots, p_k, i_1, \ldots, i_{k'}\}$ 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 Γ. We note $r_1, ..., r_m$ the address of each global variables in X and $a_1, ..., a_k$, k pairwise distinct new valid addresses. Then, $D(p_i)$ is $\{NULL, r_1, ..., r_m, a_1, ..., a_j\}$ and $D(i_j)$ is $[\,0\,,\,N_U\,]$ if i_j is unsigned and $[\,-N_I\,,\,N_I\,]$ otherwise – N_I and N_U are the number of signed and unsigned integers in X.

Language Γ. PRECA considers the constraint language Γ 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) Γ holds Horn clauses of arbitrary size which is crucial to handle conditional preconditions, e.g., `find_first_of` *weakest precondition* in Listing 1 contains the constraint $m > 0 \Rightarrow valid(a)$.

	Grammar	
P	:=	$C \Rightarrow A \mid A \mid \neg A$
C	:=	$C \wedge C \mid A \mid \neg A$
A	:=	$valid(p_j) \mid alias(p_j, p_l) \mid deref(p_j, g)$
	$\mid$	$i_j = 0 \mid i_j < 0 \mid i_j \leq 0 \mid i_j = i_l \mid i_j < i_l \mid i_j \leq i_l$
	Semantics of constraint over pointers	
$valid(p_j)$	$\equiv$	$p_j \neq NULL$
$alias(p_j, p_l)$	$\equiv$	$p_j = p_l$
$deref(p_j, g)$	$\equiv$	$p_j = \&g$ where $\&g$ is the address of g

p_j (resp. i_j) are pointers (resp. integers) and g is a global variable.

Table 2: Grammar of constraint language Γ

Bias *B*. The bias B is a finite set of constraints extracted from Γ. 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 Γ".* Indeed, from our experience, validity of a pointer is usually conditioned by constraints over integer variables.

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.

$a(c) \longrightarrow \neg a(\bar{c}), \forall c \in B$
$a(c_1) \longrightarrow a(c_1 \vee c_2)\ , \forall c_1, c_2 \in B$
$a(i_1 = 0) \wedge a(i_1 = i_2) \longrightarrow a(i_2 = 0)$
$a(i_1 = i_2) \wedge a(i_2 = i_3) \longrightarrow a(i_1 = i_3)$
$a(\neg valid(p_1)) \wedge a(\neg alias(p_1, p_2)) \longrightarrow a(valid(p_2))$
$a(valid(p_1)) \wedge a(alias(p_1, p_2)) \longrightarrow a(valid(p_2))$
$a(alias(p_1, p_2)) \wedge a(alias(p_2, p_3)) \longrightarrow a(alias(p_1, p_3))$

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_l) 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 Ω, 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/tvzzpvmm), Frama-C WP test suite (https://tinyurl.com/ycxdbjf3), 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.

[1]https://github.com/binsec/preca

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. $Q = valid(ret)$ for pointers, and $Q = ret \neq 0$ or $Q = ret > 0$ 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 ukn 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 $\geq$ 5min (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 $10\times$ 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

↳ PRECA and ↳ Both). All methods except Daikon benefit from it, highlighting the quality of PRECA sample generation mechanism.

PRECA *significantly outperforms prior black-box methods. Especially, it infers in 5s 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.*

	1s		5s		5 mins		1h	
	#WP$_\top$	#WP$_Q$	#WP$_\top$	#WP$_Q$	#WP$_\top$	#WP$_Q$	#WP$_\top$	#WP$_Q$
Daikon	1.4/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44
↳ PRECA	2/50	1/44	2/50	1/44	2/50	1/44	2/50	1/44
↳ Both	3.3/50	0/44	5.7/50	0/44	5.7/50	0/44	5.7/50	0/44
PIE	16.4/50	4.7/44	16.4/50	4.7/44	17.7/50	4.7/44	17.7/50	5.3/44
↳ PRECA	5/50	3/44	5/50	3/44	5/50	3/44	5/50	3/44
↳ Both	25.3/50	11.3/44	25.4/50	11.3/44	26.4/50	11.3/44	28.4/50	11.3/44
Gehr et al.	8.0/50	5.0/44	16.8/50	8.1/44	26.1/50	10.1/44	26.1/50	10.3/44
↳ PRECA	37/50	15/44	43/50	17/44	46/50	18/44	46/50	18/44
PRECA	29/50	11/44	**38/50**	**16/44**	**46/50**	**18/44**	**46/50**	**18/44**
↳ BK	15/50	8/44	38/50	16/44	45/50	18/44	46/50	18/44
↳ Preproc.	19/50	9/44	36/50	16/44	45/50	18/44	46/50	18/44
↳ $\emptyset$	13/50	7/44	35/50	15/44	45/50	18/44	46/50	18/44
↳ Random	29.9/50	12.1/44	29.9/50	12.1/44	30.0/50	12.1/44	30.0/50	12.1/44
P-Gen	**34/50**	**13/44**	37/50	15/44	37/50	15/44	37/50	15/44

#WP$_\top$ (resp. #WP$_Q$) is the number of inferred weakest precondition without (resp. with) a postcondition. We study 3 variations of Daikon and PIE: (i) original one (highlighted) on 100 random examples; (ii) on PRECA examples; (iii) on both random and PRECA examples. We study the original active Gehr et al. method (highlighted) and we feed it with PRECA examples. Finally, we study PRECA with its background knowledge and preprocess (highlited), with background knowledge only (BK), with preprocessing only (Preproc.), without any of them ($\emptyset$) and in passive mode with 100 random queries (Random). P-Gen being a static method, we consider only its original form.

Table 4: Results depending on the time budget

6.3 Discussion

While PRECA shows overall good properties, it also comes with a few 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.

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 Simonis, 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 Simonis, 2012; Lallouet *et al.*, 2010; Tsouros *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.

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

[Astorga *et al.*, 2018] Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. Preinfer: Automatic inference of preconditions via symbolic analysis. In *DSN*. IEEE, 2018.

[Astorga *et al.*, 2019] Angello Astorga, P Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. Learning stateful preconditions modulo a test generator. In *PLDI'19*, 2019.

[Beldiceanu and Simonis, 2012] Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *CP'12*, 2012.

[Bessiere *et al.*, 2013] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *IJCAI*, 2013.

[Bessiere *et al.*, 2017] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 2017.

[Burdy *et al.*, 2005] Lilian Burdy, Yoonsik Cheon, David R Cok, Michael D Ernst, Joseph R Kiniry, Gary T Leavens, K Rustan M Leino, and Erik Poll. An overview of jml tools and applications. *STTT*, 2005.

[Calcagno *et al.*, 2009] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*. ACM, 2009.

[Cousot *et al.*, 2013] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *VMCAI'13*. Springer, 2013.

[Dijkstra, 1968] Edsger W Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 1968.

[Ernst *et al.*, 2001] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *TSE*, 2001.

[Floyd, 1993] Robert W Floyd. Assigning meanings to programs. In *Program Verification*. Springer, 1993.

[Garg *et al.*, 2016] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, 2016.

[Gehr *et al.*, 2015] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. Learning commutativity specifications. In *CAV'15*, 2015.

[Godefroid *et al.*, 2011] Patrice Godefroid, Shuvendu K Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*. Springer, 2011.

[Gries, 2012] David Gries. *The science of programming*. Springer Science & Business Media, 2012.

[Gulwani *et al.*, 2008] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[Gulwani *et al.*, 2017] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 2017.

[Hoare, 1969] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *CACM*, 1969.

[Hutchins *et al.*, 1994] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *ICSE*. IEEE, 1994.

[Kafle *et al.*, 2018] Bishoksan Kafle, John P Gallagher, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. An iterative approach to precondition inference using constrained horn clauses. *Theory and Practice of Logic Programming*, 2018.

[Kirchner *et al.*, 2015] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 2015.

[Lallouet *et al.*, 2010] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *ICTAI*. IEEE, 2010.

[Meyer, 1988] Bertrand Meyer. Eiffel: A language and environment for software engineering. *JSS*, 1988.

[Padhi *et al.*, 2016] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 2016.

[Paulin *et al.*, 2008] Mathias Paulin, Christian Bessiere, and Jean Sallantin. Automatic design of robot behaviors through constraint network acquisition. In *ICTAI*, 2008.

[Rossi *et al.*, 2006] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[Sankaranarayanan *et al.*, 2008] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*. ACM, 2008.

[Seghir and Kroening, 2013] Mohamed Nassim Seghir and Daniel Kroening. Counterexample-guided precondition inference. In *ESOP*. Springer, 2013.

[Tsouros *et al.*, 2020] Dimosthenis C Tsouros, Kostas Stergiou, and Christian Bessiere. Omissions in constraint acquisition. In *CP*. Springer, 2020.

[Zhang *et al.*, 2014] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. Feedback-driven dynamic invariant discovery. In *ISSTA*. ACM, 2014.