Journal of Artificial Intelligence Research 82 (2025) 901-936 Submitted 03/2024; published 02/2025

A Query-Based Constraint Acquisition Approach for Enhanced Precision in Program Precondition Inference

Grégoire Menguy GREGOIRE.MENGUY@CEA.FR
Sébastien Bardin SEBASTIEN.BARDIN@CEA.FR
Université Paris-Saclay, CEA List, France

Arnaud Gotlieb ARNAUD@SIMULA.NO
Simula Research Laboratory, VIAS Dept., Norway

Nadjib Lazaar LAZAAR@LISN.FR
Université Paris-Saclay, CNRS, LISN, France

Abstract

Program annotations in the form of function pre/postconditions play a crucial role in various software engineering and program verification tasks. However, the frequent unavailability of these annotations necessitates manual retrofitting. This paper shows how constraint acquisition, a learning framework derived from constraint programming and version space learning, can be extended for automatically inferring program preconditions. Our approach performs this inference in a black-box manner through automatic query generation and input-output observations of program executions. We introduce PreCA, the first-ever precondition inference framework leveraging query-based constraint acquisition. Notably, we specialize PreCA to handle memory-related preconditions on binary code, which pose significant challenges in data and information management systems. In contrast to prior black-box techniques, PreCA provides well-defined guarantees. Specifically, it employs a sound and complete method to generate preconditions consistent with all the observed input-output relationships of the program. Furthermore, empirical evaluations on our benchmark demonstrate that PreCA outperforms the results of state-of-the-art approaches, delivering comparable or superior results in 5s, as opposed to the 1-hour runtime of existing approaches on identical machines. We also present two successful use cases from the standard `libc` and the `mbedtls` cryptographic library. PreCA notably infers for the former one a more precise precondition than specified in the documentation.

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, Cheon, Cok, Ernst, Kiniry, Leavens, Leino, & Poll, 2005) and program refactoring (Ernst, Cockrell, Griswold, & Notkin, 2001). They can benefit both a human and automated program analyzers, typically in software verification where they enable scalable (modular) analysis (Kirchner, Kosmatov, Prevosto, Signoles, & Yakobowski, 2015; Godefroid, Lahiri, & Rubio-González, 2011). Unfortunately, annotations are rarely available and must be retrofitted by hand into the code, limiting their interest – especially for black-box third-party components.

Precondition Inference. Efforts have been devoted to *automatically infer* preconditions from the code, and contract inference is an important topic in program analysis and formal methods (Cousot, Cousot, Fähndrich, & Logozzo, 2013; Ernst et al., 2001; Padhi, Sharma, & Millstein, 2016; Astorga, Srisakaokul, Xiao, & Xie, 2018; Gehr, Dimitrov, & Vechev, 2015). The C code shown in Listing 1 illustrates how difficult it can be to generate precise preconditions even for simple functions. In this example, determining the precondition requires reasoning over memory layout, handling loops, sub-function calls, and disjunctive behaviors. Since the precondition inference problem is undecidable (Rice, 1953), 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 (e.g., nested loops, dynamic jumps, dynamic data structures) and cannot cope with black-box, third-party, 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, Yang, Rungta, Person, & Khurshid, 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.

```
int find(int* t, int n, int val) {
    for (int i = 0; i < n; i++)
        if (t[i] == val) return i;
    return n;
}

int find_first_of(int* a, int m, int* b, int n) {
    for (int i = 0; i < m; i++)
        if (find(b, n, a[i]) < n) return i;
    return m;
}
```

Listing 1: Example of C code returning the index of the first element in `a`, which is also in `b`. To execute `find_first_of` without runtime errors (i.e., postcondition "*true*"), `a`, `m`, `b`, and `n` must satisfy the precondition: $(m > 0 \Rightarrow \text{valid}(a)) \wedge (m > 0 \wedge n > 0 \Rightarrow \text{valid}(b))$, where $valid(a) \equiv (a \neq NULL)$. For instance, if `a` and `b` are valid addresses, and `m`=-3, `n`=2, `find_first_of` executes without a runtime error. On the other hand, if `a` = `b` = NULL, and `m`=1 and `n`=2, it does not.

Constraint Acquisition. Constraint programming (CP) (Rossi, Van Beek, & Walsh, 2006) has made considerable progress over the last forty years, becoming a powerful paradigm for modeling and solving combinatorial problems. However, modeling a problem as a constraint network remains a challenging task that requires expertise in the field. Among the *constraint acquisition* (CA) systems that have been introduced to support the uptake of constraint technology by non-experts (Bessiere, Carbonnel, Dries, Hebrard, Katsirelos, Lazaar, Narodytska, Quimper, Stergiou, Tsouros, & Walsh, 2023), Conacq acquires a set of constraints that correctly represent as solutions all examples classified as positive by

the user and reject as non-solutions all examples classified as negative (Bessiere, Koriche, Lazaar, & O'Sullivan, 2017). The system is presented both in *passive* and *active* versions. In the former, the examples are given by the user, while in the latter, they are generated by Conacq and classified by the user. Despite the robust theoretical foundations of Conacq, its practical implementation faces challenges. The system may require the submission of *thousands of queries to a user*, making it challenging to apply in real-world scenarios with humans as oracles. In automated program analysis, a substantial number of queries is viable because the program acts as an automated oracle, provided that the program execution terminates promptly.

Contributions. The contributions of this paper can be summarized in three points:

- **PreCA**: We introduce PreCA, the first-ever framework based on CA for inferring preconditions (Section 4). We establish that PreCA exhibits theoretical correctness properties compared to previous black-box approaches. Specifically, when the learning language is expressive enough, and the program execution terminates, PreCA is guaranteed to infer the so-called *"weakest precondition."*

- **Memory-Oriented Application:** In Section 5, we illustrate the application of PreCA in the context of inferring memory-related preconditions. For that, we introduce a specialized constraint language, including memory constraints. Additionally, we present domain-based strategies using background knowledge, including facts over memory constraints, and a preprocessing step based on the intuition that simple input memory layouts generally result in positive classifications. These combined strategies enable us to infer 2.2 times more weakest preconditions in 1 second.

- **Empirical Evaluation:** In Section 6.1, we empirically evaluate the effectiveness of our method on various benchmark functions. The results demonstrate that PreCA outperforms previous precondition learners, whether they are black-box or white-box. Notably, even with a 5-second budget per sample, PreCA outperforms the state-of-the-art approaches, even when they are allocated a 1-hour time budget per sample.

To the best of our knowledge, it is the *first* application of CA to program analysis. Tackling precondition inference problem using CA is advantageous, resulting in favourable theoretical properties and interesting results compared to the existing approaches. Hence, program analysis emerges as a valuable application for CA, bypassing specific limitations and opening the door to new challenges and opportunities for the CA community.

This article extends the content presented in the paper *Automated Program Analysis: Revisiting Precondition Inference Through Constraint Acquisition* (Menguy, Bardin, Lazaar, & Gotlieb, 2022), published at IJCAI-ECAI 2022. It extends the core technical material in five significant ways. Firstly, we provide a revised version of PreCA in Algorithm 2 to enhance its autonomy in constructing the search space and its efficiency in terms of generating test cases and convergence. Consequently, we have also revised all the related propositions and proofs. Secondly, we give a detailed presentation of each component of PreCA. This includes the description of the memory representation used by PreCA and how it impacts its various components, notably the oracle. Thirdly, we extend the constraint language by introducing new constraints related to strings. Fourthly, we explore two more

research questions in our experimental evaluation of PreCA (RQ5 and 6). Finally, we present the results of PreCA over two use-cases from the `libc` and the `mbedtls` libraries to highlight its practical utility.

2. Motivating Example

The focus of this paper is on *memory-related preconditions*, such as predicates specifying the inputs for which a function can execute without causing a memory violation, and this is done in a *black-box manner*. Inferring these preconditions is crucial to ensure the error-free execution of the program. As motivating example, we consider the function `find_first_of` in Listing 1 from the Frama-C code verification platform's test suite (Kirchner et al., 2015). In this case, the expected result is $(m > 0 \Rightarrow \text{valid}(a)) \ \wedge \ (m > 0 \wedge n > 0 \Rightarrow \text{valid}(b))$. The precondition is expressed as a condition involving `a`, `m`, `b`, and `n`, ensuring a successful execution of the function with an output that satisfies the postcondition *true*. Our objective is to acquire preconditions in a black-box manner (i.e., by executing test cases), without analysing the source code.

White-Box vs. Black-Box Methods. When available, using the source code to infer preconditions is interesting in white-box analyses (e.g., P-Gen (Seghir & Kroening, 2013)). Yet, several practical issues prevent the software engineer from fully exploiting white-box analyses. First, having the *whole* source code is often unrealistic (many projects embed third-party binary components). Second, in practice, program analyzers focus on a single programming language where many projects use combinations of them (e.g., inline assembly in C code). Third, despite the progress achieved in recent years, white-box program analysis still fails on complex codes (containing unbounded or input-bounded loops, recursion, dynamically allocated data structures, etc.), possibly leading to serious scalability or precision issues. Fourth, code obfuscation (Nagra & Collberg, 2009) used in some application domains (e.g., video games, android and web applications or military systems) renders white-box analyses impossible or extremely hard to apply (Ollivier, Bardin, Bonichon, & Marion, 2019a, 2019b). To cope with these issues, black-box methods can be employed because they only require the availability of the executable code. Yet, as generalization occurs, black-box methods can return preconditions which are correct only with respect to a set of test cases. In other words, these are true predicates but not necessarily actual preconditions of the considered function.

Black-box analysis through passive learning. 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 for precondition inference. Yet, the specification being unavailable, assuming that the user can provide meaningful test cases is often unrealistic – especially when the source code is also unavailable. Another solution would be to generate test cases randomly, but this is rarely satisfactory. Indeed, it ensures no guarantee and requires running many test cases. For example, on `find_first_of`, Daikon (Ernst et al., 2001) cannot infer the precondition even with $10,000$ random test cases, which already takes more than 1h to monitor. Conversely, PIE (Padhi et al., 2016) only succeeds with $10,000$ test cases, which take 900s to run.

Active learning. Gehr et al. (Gehr et al., 2015) performs active learning, by generating test cases automatically. This approach is more actionable and needs less user expertise. Still, active learning methods developed so far lack theoretical guarantees. Indeed, they cannot ensure that all useful test cases have been considered. The 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 CA (Bessiere et al., 2017). Unlike previous active approaches, PreCA mixes pertinent test case generation and learning, resulting in robust theoretical properties. When generating a test case, PreCA directly observes the function's behavior and updates its search space accordingly. Consequently, given a set of constraint candidates, PreCA generates all pertinent test cases, ensuring convergence. Therefore, if the execution of all test cases terminates, and the set of constraint candidates is expressive enough, PreCA infers the weakest, i.e., optimal, precondition. Regarding our example, PreCA infers the (non-trivial) *weakest precondition* $(m > 0 \Rightarrow valid(a)) \wedge\ (m > 0 \wedge n > 0 \Rightarrow valid(b))$,[1] where $valid(p) \equiv (p \neq NULL)$, in 172s, with 45 test cases.

Alg.	Active?	Success.	#Test cases	Time	Result
Daikon (100)	no	✗	100	39s	$n \neq 0 \wedge m \neq n$
Daikon (1k)	no	✗	1,000	409s	$true$
Daikon (10k)	no	✗	10,000	>1h	$true$
PIE (100)	no	✗	100	34s	$((n \leq 1 \vee valid(b)) \wedge valid(a)) \vee m \leq 1$
PIE (1k)	no	✗	1,000	282s	$m \leq 0 \vee ((n \leq 1 \vee valid(b)) \wedge valid(a))$
PIE (10k)	no	✓	10,000	900s	$(valid(b) \vee m \leq 0 \vee n \neq 1) \wedge (m \leq 0 \vee n \leq 1 \vee valid(b)) \wedge (m \leq 0 \vee valid(a))$
Gehr et al.	yes	✗	177	700s	$m \leq 0 \vee (valid(a) \wedge (n = 0 \vee ((valid(b) \wedge n > 0))))$
PreCA	**yes**	✓	**45**	**172s**	$(m > 0 \Rightarrow valid(a)) \wedge\ (m > 0 \wedge n > 0 \Rightarrow valid(b))$

Daikon and PIE perform both passive learning. Thus, Daikon/PIE (100), (1k) and (10k) stand for 100, 1,000, and 10,000 randomly generated test cases. Note that "Time" includes the test case generation, execution, and inference.

Table 1: `find_first_of` results, no source code (CPUs: 6 Intel Xeon E-2176M CPUs; RAM: 32 GB)

3. Background

This section provides essential background on constraint acquisition and program analysis. We explore the constraint acquisition framework, with a particular focus on Conacq.2 (Bessiere et al., 2017), which forms the basis of PreCA. Following that, we formally define the concept of a precondition using the operational semantics.

3.1 Constraint Acquisition

Active Constraint Acquisition is a process where a CA-learner interacts with an oracle through queries (Bessiere et al., 2017, 2023). To enable communication, both the CA-learner and the user use a shared vocabulary, which consists of a finite set of variables X taking values from a finite domain D. A constraint network is built over this shared vocabulary $\langle X, D \rangle$, with a set C of constraints. Each constraint c in C is denoted as $\langle var(c), rel(c) \rangle$, where $var(c)$ is a tuple of variables from X, known as the constraint scope. The relation $rel(c)$ is defined over $D^{var(c)}$, where tuples in $rel(c)$ indicate valid combinations of value assignments for the variables in $var(c)$. The arity of a constraint c is the size $|var(c)|$ of

[1] We do not consider constraints over array sizes as C does not trigger out-of-bound errors. We focus on validity and alias/overlapping problems.

its scope. Given a vocabulary $\langle X, D\rangle$, an example e is an element of D^X. An example e satisfies a constraint c if the projection of e onto the scope $var(c)$ is in $rel(c)$. An assignment e violates a constraint c, or in other words, the constraint c rejects e, if e does not satisfy c. Therefore, an example e satisfies a constraint network C if it satisfies every constraint c in C. Such an example is called a solution of C. A non-solution of C is an example that violates at least one constraint from C. The set of all solutions of a constraint network C is denoted $sol(C)$. C is satisfiable if $sol(C) \neq \emptyset$ and unsatisfiable otherwise. If $sol(C) \subseteq sol(C')$ for two constraint networks C and C', then C entails C' (i.e., $C \models C'$). Finally, if $sol(C) = sol(C')$, then C and C' are equivalent (i.e., $C \equiv C'$).

Example 1. *Consider the constraint network* $C_1 = \{c_1 : x \times y > 0, c_2 : x + y > 0\}$ *expressed using the vocabulary* $\langle X, D\rangle = \langle\{x, y\}, \{-5, ..., 5\}\rangle$.

- $e_1 = \langle -3, -1\rangle$*, where* $x \leftarrow -3$ *and* $y \leftarrow -1$*, is an example that satisfies* c_1 *and violates* c_2*, that is,* $e_1 \notin sol(C_1)$.

- $e_2 = \langle 2, 1\rangle$ *is an example that satisfies both* c_1 *and* c_2*, that is,* $e_2 \in sol(C_1)$.

The CA-learner uses a *constraint language* Γ, containing relations of bounded arity. The *constraint bias*, denoted as B, is a set of constraint candidates generated from Γ using the vocabulary $\langle X, D\rangle$. The CA-learner uses B to build constraint networks. A constraint network C is said representable by B if $C \subseteq B$.

For a given vocabulary $\langle X, D\rangle$, a *concept* is a Boolean function f over D^X that assigns either $\{true, false\}$ to each example $e \in D^X$. A *representation* of a concept f is a constraint network C for which $f^{-1}(true) = sol(C)$.

A *"target concept"* is the concept f_T intended to be learned through the constraint acquisition process. The *"target network"* is a network T that satisfies $T \subseteq B$ and represents f_T. Thus, we can say that the target concept f_T is representable by B.

A *membership query*, or simply a query, requests the oracle to classify a given example e as either positive or negative. The answer is *"yes"* if e is a solution of the target concept and *"no"* otherwise. An example e is classified as positive if the query answer is *"yes"* (resp., classified as negative if the answer is *"no"*). The oracle is supposed to consistently provide correct answers to queries, without any mistakes.

A constraint network is said to *agree* with a set of examples $E \subset D^X$ if and only if it accepts all positive examples in E and rejects all negative examples in E. For a given example e, $\kappa(e)$ represents the set of all constraints in B that reject e. Given a subset of variables $Y \subset X$, $e[Y]$ is the projection of e on Y.

Learning a constraint network using queries involves finding the shortest query sequence that converges to a constraint network representing the target concept f_T.

Definition 1 (Identification problem)**.** *Given a bias* B *defined on a vocabulary* $\langle X, D\rangle$*, the identification problem for a target-concept* f_T *representable by* B *involves finding a sequence of queries* $\langle q_{i1}, q_{i2}, \ldots, q_{im}\rangle$ *that leads to the discovery of a constraint network* C *representing the target-concept* f_T*.*

Algorithm 1: Conacq.2 (Bessiere et al., 2017)

In : a bias B;
Out : A constraint network or collapse;

begin
  $\Omega \leftarrow \top$
  while *true* **do**
    $e \leftarrow$ queryGeneration(B, Ω) `// generates an informative example`
    if $e = nil$ **then**
      if Ω *is SAT* **then return** `network`(Ω) `// convergence`
      else return "collapse" `// collapse`
    if `Ask`$(e) = no$ **then** $\Omega \leftarrow \Omega \wedge \left(\bigvee_{c \in \kappa(e)} a(c)\right)$ `// expansion`
    else $\Omega \leftarrow \Omega \wedge \left(\bigwedge_{c \in \kappa(e)} \neg a(c)\right)$ `// expansion`

Conacq.2 (Algorithm 1). Conacq.2 (Bessiere et al., 2017) is a CA-learner that employs membership queries to interact with an oracle. Internally, each constraint $c \in B$ is linked to a Boolean atom $a(c)$ that indicates whether c should be included in the learned network. Subsequently, using a sequence S of generated and classified examples, Conacq.2 constructs a compact representation of the learner's search space in the form of a clausal formula Ω:

$$\Omega = \bigwedge_{e \in E^+} \left(\bigwedge_{c \in \kappa(e)} \neg a(c) \right) \wedge \bigwedge_{e \in E^-} \left(\bigvee_{c \in \kappa(e)} a(c) \right)$$

where E^+ (resp., E^-) represents the set of positive (resp., negative) examples generated in the sequence S. At the beginning, both the sequence S and the clausal formula Ω are empty. The **expansion** of Ω is performed iteratively in a step-by-step process. During each iteration, Conacq.2 **generates an example** and presents it to the oracle. The generated example e is carefully chosen to provide valuable information for pruning the learner's search space, regardless of the oracle's response. In other words, e is intended to be informative regarding the examples generated so far in the sequence S.

Definition 2 (Informative Example)**.** *Given a bias B and a sequence of queries S, an example e is informative with respect to S if there exist two constraint networks $C_1, C_2 \subseteq B$ such that both networks agree with S, and $(e \models C_1 \Leftrightarrow e \not\models C_2)$.*

To ensure both termination and correctness, the query generation process must satisfy the following properties:[2]

$$\text{queryGeneration}(B, \Omega) \text{ terminates} \tag{1}$$

$$\big(\exists e \in D^X, e \text{ is informative w.r.t. } S\big) \Rightarrow \text{queryGeneration}(B, \Omega) \neq nil \tag{2}$$

$$\text{queryGeneration}(B, \Omega) \neq nil \Rightarrow \text{queryGeneration}(B, \Omega) \text{ is informative wrt. } S \tag{3}$$

[2]The queryGeneration procedure proposed by (Bessiere et al., 2017) satisfies all these 3 properties.

When no informative example remains, the inference process terminates. If Ω is satisfiable, it indicates **convergence**, and Otherwise, Conacq.2 determines that the target concept cannot be represented by B and returns **collapse**. By generating all informative examples within a finite time, Conacq.2 is correct and terminates (Bessiere et al., 2017).

Example 2. *Let us apply* Conacq.2 *to learn the target network* $T = \{x \times y > 0, x + y > 0\}$ *from Example 1. We have the variable set* $X = \{x, y\}$ *defined over the domain* $\{-5, \ldots, 5\}$, *and a bias* $B = \{x > 0, x \leq 0, y > 0, y \leq 0, x = y, x \neq y\}$. *The execution of* Conacq.2 *is detailed in Table 2, where only informative examples are generated. For instance, after* e_1, *no example violating a subset of* $x \leq 0$, $y \leq 0$, *and* $x \neq y$ *is generated because these constraints are already forbidden by* Ω. *After generating four examples, no other informative example remains, and the resulting theory is* Ω_4, *which simplifies to* $\neg a(x \leq 0) \wedge \neg a(y \leq 0) \wedge \neg a(x \neq y) \wedge \neg a(x = y) \wedge a(y > 0) \wedge a(x > 0)$. *Therefore,* Conacq.2 *has converged to* $\{x > 0, y > 0\}$, *which is equivalent to* T.

Iterations	Membership Queries		Version space (Ω)
	Examples $\langle x, y \rangle$	Classification	
1	$e_1 = \langle 1, 1 \rangle$	yes	$\Omega_1 = \neg a(x \leq 0) \wedge \neg a(y \leq 0) \wedge \neg a(x \neq y)$
2	$e_2 = \langle 2, -1 \rangle$	no	$\Omega_2 = \Omega_1 \wedge (a(x \leq 0) \vee a(y > 0) \vee a(x = y))$
3	$e_3 = \langle 1, 3 \rangle$	yes	$\Omega_3 = \Omega_2 \wedge \neg a(x = y)$
4	$e_4 = \langle -1, 3 \rangle$	no	$\Omega_4 = \Omega_3 \wedge (a(x > 0) \vee a(y \leq 0) \vee a(x = y))$

Table 2: Example of Conacq.2 execution.

3.2 Program Analysis: Operational Semantics

To reason about program executions, it is essential to establish a formal framework for defining programs and their execution. For this purpose, various formalisms, such as denotational and axiomatic semantics (Schmidt, 1986; Winskel, 1993; Nepomniaschy, Anureev, & Promskii, 2003), have been proposed. Among such formalisms, we use the operational semantics description (Plotkin, 2004), which characterizes program execution as a sequence of elementary steps. This standard formalism fits well our context presented in Section 4. Indeed, our approach runs binary code (i.e., sequence of assembly instructions), which has a direct interpretation in operational semantics (Bardin & Herrmann, 2011). Hence, this formalism simplifies the presentation of CA for precondition inference.

In operational semantics, a *program* p is a finite sequence $i_1; ...; i_n$ of instructions operating on a *memory state* s. The special instruction *skip* signifies the computation termination without modifying the memory state. The combination of a program p and a memory state s is termed a *configuration*, noted p/s, which represents the current memory state and the remaining instructions to be executed. To further clarify the operational semantics, we formalize an execution step, which corresponds to the execution of a single instruction.

Definition 3. ***(Operational semantics)*** *An operational semantics is a transition function "*$\rightsquigarrow$*" specifying transitions between pairs of configurations.*

Next, we define the reflexive closure $\rightsquigarrow^*$, which captures the successive execution steps. This closure allows us to describe the complete execution of a program and analyze reachability properties such as *end*, *diverge*, or *stuck*.

Definition 4. ***(End/diverge/stuck)*** *A configuration p/s:*

- *ends if and only if it exists s' s.t. $p/s \rightsquigarrow^* skip/s'$;*
- *diverges if and only if it can be derived infinitely, noted $p/s \rightsquigarrow^*$;*
- *is stuck if and only if there is no p'/s' s.t. $p/s \rightsquigarrow p'/s'$.*

Intuitively, these three cases describe the possible behaviors of a program. The *end* case means the program was successfully executed (execution terminated without crashing). The *diverge* case means the execution never terminates (i.e., infinite loop or infinite recursion).[3] Finally, the *stuck* state enables the description of ***runtime errors*** (RTE). Especially, a configuration p/s leads to an RTE if and only if $p/s \rightsquigarrow^* p'/s'$ where p'/s' is stuck. In the following, we are only interested in the high-level behaviors of the code (ends, diverges, is stuck) and rely only on the $\rightsquigarrow^*$ operator. Still, for the sake of completeness, Appendix B presents an example of a full operational semantic for a simple imperative language.

3.3 Program Analysis: Preconditions and Weakest Preconditions

Given a program function F and a predicate Q over F output called a postcondition, Hoare logic (Hoare, 1969) defines the precondition (a predicate over F inputs) of F w.r.t. Q.

Definition 5 (Precondition)**.** *Given a function F and a postcondition Q, P is a precondition of F with respect to Q if, for all states s such that $s \models P$, it holds that $F/s \rightsquigarrow^* skip/s'$ and $s' \models Q$. Thus, we denote $\{P\}F\{Q\}$.*

A function F may have multiple preconditions for a given postcondition Q. The *weakest precondition* holds special importance in software engineering and formal verification (Hoare, 1969; Floyd, 1993). The challenge of automatically computing the *weakest precondition* of F w.r.t. Q has been extensively investigated since the 1970s. However, due to the undecidability of the problem (Rice, 1953), conventional approaches are limited to manual annotations or approximations.

Definition 6 (Weakest precondition)**.** *Let a function F and a postcondition Q. The weakest precondition of F with respect to 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 3. *Consider the function under analysis F :* `int foo(int a) { return a + (1/a); }`*. Note that F is undefined when $a = 0$. Hence, for a postcondition ($Q_1 = True$), possible preconditions could be ($P_1 : a = 5$), ($P_2 : a > 10$), or ($P_3 : a < 0$). However, such preconditions are overly restrictive as they discard a significant set of values for* `a` *where the execution of F terminates and the output satisfies Q_1. The less restrictive precondition, i.e., the weakest precondition, is the one that discards only inputs for which the execution of F does not terminate, crashes, or in case of termination, the output does not satisfy the postcondition Q_1. In this example, $\mathcal{WP}(F, Q_1)$ is $a \neq 0$. Now, let's consider a second postcondition Q_2 as "the return value must be ≥ 0." In this case, $\mathcal{WP}(F, Q_2)$ is $a > 0$. For a detailed example of* WP calculus *over a simple imperative language, refer to Appendix C.*

[3] Deciding program termination is undecidable in general.

4. Precondition Inference Through Constraint Acquisition

We aim to infer the *weakest precondition* of a function F under analysis, for a given postcondition Q using query-based CA. To the best of our knowledge, this represents the first instance of employing CA for program analysis.

4.1 Precondition Acquisition

In this section, we illustrate how precondition inference can be formulated as a CA problem, as presented in Fig. 1. Here, the traditional user role is replaced by an *automated oracle*, able to automatically classifying examples by executing the program on specified inputs and checking if the output satisfies the postcondition Q. The target concept is the weakest precondition $\mathcal{WP}(F, Q)$. The *variable* set X corresponds to M, where M represents the initial memory state required to run F. M is a mapping from symbols, such as F arguments and global variables, to their respective values. The *domain* D of M represents the finite set of all possible mappings, indicating that D^M defines the domain of F. The *constraint language* Γ and the *bias* B are sets of constraints over M. A detailed description of Γ and B is provided in Section 5.1. Lastly, a *membership query* $e \in D^M$ is a test case, representing a comprehensive assignment of M that enables the execution of F over e.

Constraint Acquisition		Precondition Inference
Target network	$\rightarrow$	Weakest precondition
Variables	$\rightarrow$	Program memory
User	$\rightarrow$	Automated oracle
Constraint language	$\rightarrow$	Constraints over memory
Membership query	$\rightarrow$	Test case

Figure 1: From Constraint Acquisition to Precondition Inference

4.2 Description of PreCA

We depict in Fig. 2 the full PreCA framework. In the following, we provide a detailed presentation of PreCA, which primarily consists of the *oracle* and the *acquisition module*.

Oracle. A membership query involves presenting an example e to the oracle for classification as positive or negative. In other words, considering a function F and a postcondition Q,[4] the oracle, implemented by runOracle, must determine within a finite timeframe whether $e \models \mathcal{WP}(F, Q)$ – that is, whether $F/e \rightsquigarrow^* skip/s$ and $s \models Q$. However, the execution termination problem is generally undecidable (Rice, 1953), making our classification problem similarly undecidable. Acknowledging this, expecting the oracle to classify all examples may be unrealistic. Therefore, we allow the oracle to respond with *ukn* (i.e., "unknown") when it cannot verify whether $e \models \mathcal{WP}(F, Q)$ or not (typically due to an ex-

[4] The postcondition can be any decidable property over the output memory state. Decidability is required to be able to check in finite time if the output verifies Q. This is not a restrictive assumption as most used postconditions are decidable.

ecution timeout).[5] This necessitates a revision of the Conacq framework since Conacq can only handle *yes* and *no* classifications.

In summary, runOracle must adhere to the following specifications:

$$\begin{aligned} &\bullet \ \text{runOracle terminates and returns } yes, no \text{ or } ukn \\ &\bullet \ \Big(\text{runOracle}(F, Q, e) = yes\Big) \Rightarrow e \models \mathcal{WP}(F, Q) \\ &\bullet \ \Big(\text{runOracle}(F, Q, e) = no\Big) \Rightarrow e \not\models \mathcal{WP}(F, Q) \end{aligned} \tag{4}$$

Note that an oracle always answering *ukn* would fit our definition. However, if the oracle provides only *ukn* response, PreCA might infer the unhelpful empty concept (i.e., $false$). In practice, the oracle should minimize the use of *ukn* responses to yield more insightful preconditions. A detailed implementation of the oracle to handle memory-related preconditions is given in Section 5.

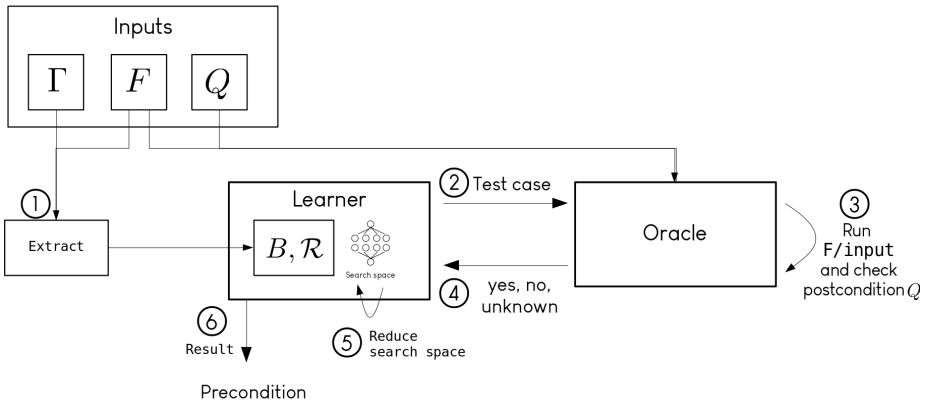

Figure 2: PreCA framework: Firstly, it extracts the bias B from the constraint language Γ and the function F. Then, it generates a test case from the bias and queries the oracle for classification. The oracle runs F over the test case and evaluates its behavior w.r.t., the postcondition Q. Based on the classification, the learner updates its search space. On convergence, PreCA returns a precondition.

Acquisition Module. The PreCA (Algorithm 2) acquisition module first invokes the Extract function to obtain the vocabulary $\langle X, D\rangle$ and the bias B from the function F and the constraint language Γ at line 2. The Extract function also returns a set of constraints $\mathcal{R} \not\subset B$, which represents a set of mandatory constraints in program analysis. These constraints must be satisfied during the query generation process to guarantee that the returned example is a feasible test case. An example of such constraints is the one ensuring that aliasing pointers cannot reference strings of distinct sizes.

Then, the GetHints function at line 3 collects the *"hint"* constraints from B. This is a subset of B where each constraint c is likely inconsistent with $\mathcal{WP}(F, Q)$ (i.e., if $e \models c$, runOracle(F, Q, e) likely returns no). Its choice does not affect correctness, but we observe in Section 5.3 that hint constraints can accelerate the acquisition process through the

[5] If F always terminates, the classification problem becomes decidable, and the timeout can be disregarded, eliminating the need for a *ukn* answer.

Algorithm 2: PreCA

In : A function F; a postcondition Q; a constraint language Γ;
Out : A constraint network over F input or collapse;

1 **begin**
2 $(\langle X, D\rangle, B, \mathcal{R}) \leftarrow$ Extract(F, Γ)
3 $\mathcal{H} \leftarrow$GetHints(B)
4 $\Omega \leftarrow \bigwedge_{c\in\mathcal{R}} a(c)$
5 $\Theta \leftarrow \emptyset$
6 **while** *true* **do**
7 $e \leftarrow$ GenTestcase$(B, X, \Omega, \mathcal{H}, \Theta)$
8 **if** $e = nil$ **then**
9 **if** Ω *is SAT* **then return** $\texttt{network}(\Omega) \setminus \mathcal{R}$
10 **else return** "*collapse*"
11 **if** runOracle$(F, Q, e) \neq yes$ **then** $\Omega \leftarrow \Omega \wedge (\bigvee_{c\in\kappa(e)} a(c))$
12 **else** $\Omega \leftarrow \Omega \wedge (\bigwedge_{c\in\kappa(e)} \neg a(c))$

Algorithm 3: GenTestcase

In : A bias B; a variables set X; a theory Ω; finite sets of constraints $\mathcal{H}$, Θ;
Out : A query

1 **begin**
2 $C \leftarrow \{c \mid a(c) \in \Omega\}$
3 $\Delta \leftarrow C \cup \Theta \cup \{b_c \Leftrightarrow c \mid c \in \mathcal{H}\} \cup \sum_{c\in\mathcal{H}} b_c \leq 1$;
4 **if** $sol(\Delta) \neq \emptyset$ **then**
5 pick $e \in sol(\Delta)$;
6 **if** $e \not\models \bigvee_{c\in\mathcal{H}} c$ **then** $\Theta \leftarrow \Theta \cup \{\bigvee_{c\in\mathcal{H}} b_c\}$
7 **else** $\Theta \leftarrow \Theta \cup \{\overline{b_c} \mid e[b_c] = 1\}$
8 **return** $e[X]$
9 **return** queryGeneration(B, Ω)

generation of examples with a high probability of being classified positively. The implementation details of Extract and GetHints functions depend on the application context, and Section 5 gives a detailed presentation on how to implement them to infer memory-related preconditions at the binary level.

PreCA initializes Ω to a theory enforcing the mandatory constraints in $\mathcal{R}$ and iteratively expands it by processing examples generated at line 7. PreCA submits these examples to the oracle for classification (runOracle call at line 11). If the oracle answers *yes*, we discard all constraints in $\kappa(e)$ from B, those rejecting e, by expanding Ω with negative unit clauses (line 12). 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 11). Throughout the acquisition, Ω represents the set of candidate solutions, i.e., the possible constraint networks from the bias that agree with observed queries. If there is no example to return, indicating no informative examples

remaining, and Ω is not satisfiable, a *"collapse"* message is returned at line 10—we say that PreCA collapsed. This may occur when the concept to learn is not representable by B or if an example classified as *ukn* conflicts with another example classified as *yes*. Otherwise, we return the constraint network encoded by Ω through the `network` function, and we remove the mandatory constraints in $\mathcal{R}$, which are implicit in program analysis and are unnecessary to be part of a precondition (line 9). It is important to note that the oracle clearly distinguishes between a negative response and an unknown response, whereas the PreCA algorithm chooses to handle them the same way to ensure the correctness.

The GenTestcase function, as presented in Algorithm 3, extends the queryGeneration function from (Bessiere et al., 2017) with additional steps at lines 2-8. The purpose of this extension is to generate examples that accept at most one constraint from $\mathcal{H}$, anticipating that these examples might be classified as positive. For that, the function needs to generate an example that satisfies C, the set of all the already learned constraints, along with the mandatory constraints in $\mathcal{R}$ (line 2). A constraint network Δ is constructed to produce an example satisfying C, differing from all previous examples (encoded in Θ), and guaranteeing the acceptance of at most one hint constraint in $\mathcal{H}$ (line 3). To do so, each constraint c from $\mathcal{H}$ is reified with a boolean variable b_c, used to compel the number of constraints from $\mathcal{H}$ that are verified. After handling the set $\mathcal{H}$, the query generation process follows the standard process of the queryGeneration function. The specific use and rationale behind $\mathcal{H}$ and Θ are explained in Section 5.3. It is worth noting that, unlike queryGeneration, GenTestcase may generate redundant (non-informative) examples at lines 2-8. Such non-informative examples may lead the acquisition process to a collapse state if, for instance, an example e_1 is classified as *"yes,"* and a second example e_2, now non-informative, is classified as *ukn*. However, our experimental evaluation indicates a significant speed-up due to the examples generated through the heuristic based on hint constraints in GenTestcase, and zero collapse occurrences due to conflicting answers.

4.3 Theoretical Analysis

In this section, we discuss key properties of PreCA, including consistency, termination, soundness, and correctness. Consistency, a fundamental property, ensures that the acquired constraint networks agree with the generated and classified examples. Soundness and correctness naturally follow. Soundness guarantees that PreCA obtains a precondition when the weakest precondition is representable by the bias. The correctness property holds under the additional assumption that the oracle never returns *ukn* during the acquisition process. In such cases, PreCA returns the expected weakest precondition. Throughout this section, we assume the termination of the Extract, GetHints and runOracle functions.

Proposition 1 (Consistency)**.** *Given a function F, a postcondition Q, and a constraint language Γ, if* PreCA *does not collapse, then the learned network L accepts all positive examples and rejects all negative and unknown examples.*

Proof. Let L be the learned network returned by PreCA, and let S be the sequence of all examples generated and classified as positive, negative, or unknown during the acquisition process. For each positive example $e \in S$, PreCA adds all $\neg a(c)$ at line 12 where $e \not\models c$. Given that L is a model of Ω where $a(c)$ encodes the presence of c in L, L accepts all positive queries in S. Moreover, for each example classified as negative or unknown $e \in S$,

PreCA adds $\bigvee_{c \in \kappa(e)} a(c)$ to Ω at line 11. Considering that L is a model of Ω where $a(c)$ encodes the presence of c in L, L contains at least one constraint from $\kappa(e)$. Therefore, L rejects e as a solution. □

Proposition 2 (Termination)**.** *Let F be a function, Q a postcondition, and Γ a constraint language. The* PreCA *algorithm terminates.*

Proof. Since the Extract, GetHints, and runOracle functions terminate, proving the termination of PreCA involves proving two key points: (i) the termination of GenTestcase and (ii) showing that a given call of GenTestcase eventually returns *nil*.
(i) Generating an example using GenTestcase is a terminating process, reducing to finding a solution to a constraint network within a finite set of variables and values.
(ii) At line 8, GenTestcase produces a finite set of examples as $\mathcal{H}$ is finite. Subsequently, the next GenTestcase calls are equivalent to queryGeneration calls, and returning only informative examples (Bessiere et al., 2017). Within any sequence of examples S generated by queryGeneration, each example $e \in S$ has a unique $\kappa(e)$. However, as B is finite, the number of possible $\kappa(e)$ is also finite. Therefore, queryGeneration explores all possibilities in the worst case and eventually returns *nil*. □

Proposition 3 (Soundness)**.** *Let F be a function, Q a postcondition, and Γ a constraint language such that $\mathcal{WP}(F, Q)$ is representable by the bias generated through $Extract(F, \Gamma)$. If* PreCA *does not collapse, then the resulting constraint network L is a precondition of F with respect to the postcondition Q.*

Proof. Let B be the bias generated through $\textsc{Extract}(F, \Gamma)$, L the resulting network of PreCA, and S the sequence of generated examples. We aim to prove that $L \Rightarrow \mathcal{WP}(F, Q)$ (Definition 6). Suppose that there exists $e \in sol(L)$ such that $e \not\models \mathcal{WP}(F, Q)$. As $\mathcal{WP}(F, Q)$ is representable by B, there is $c \in B$ such that $e \not\models c$. However, we know that $c \notin L$ (as $e \models L$). So, there is a positive example $e^* \in S$ such that $e^* \not\models c$, which added $\neg a(c)$ to Ω at line 12. But e^* is positive, so $e^* \models \mathcal{WP}(F, Q)$ and $e^* \models c$, leading to a contradiction. Therefore, $\mathcal{WP}(F, Q)$ cannot reject an example accepted by L, which proves that $L \Rightarrow \mathcal{WP}(F, Q)$. □

Theorem 1 (Correctness)**.** *Let F be a function, Q a postcondition, Γ a constraint language such that $\mathcal{WP}(F, Q)$ is representable by the bias generated through $\textsc{Extract}(F, \Gamma)$, and $S \subset D^X$ the sequence of generated examples. If there is no example in S classified as ukn by the oracle, then* PreCA *converges to a network $L \equiv \mathcal{WP}(F, Q)$.*

Proof. PreCA terminates (Proposition 2) and is sound (Proposition 3). To prove the correctness of PreCA, which can only be done under the assumption that no examples are classified as *ukn* during the acquisition process, we need to prove the completeness of PreCA under that assumption (i.e., $\mathcal{WP}(F, Q) \Rightarrow L$). Let us assume there exists $e \models \mathcal{WP}(F, Q)$ and $e \notin sol(L)$. Therefore, there exists at least one constraint $c \in L$ rejecting e and learned by PreCA. Since no example is classified as *ukn*, the only place where c can be added as a candidate to learn is at line 11 after classifying the example e as a negative example. Moreover, this implies that from $\kappa(e)$, c is one of the constraints that PreCA decides to include in the returned precondition L. It's important to note that if e is classified negatively by the runOracle function, this indicates that executing F with e

results in an output violating the postcondition Q. Therefore, $e \not\models \mathcal{WP}(F, Q)$, leading to a contradiction. Therefore, adding a constraint to a precondition L cannot reject an example accepted by $\mathcal{WP}(F, Q)$, which proves that $\mathcal{WP}(F, Q) \Rightarrow L$.

□

Discussion. The correctness of PreCA depends on the expressiveness of the bias B as well as the termination of function execution. While we cannot formally prove that B represents the weakest precondition to learn, we consider such a correctness proposition valuable in the black-box context. From a theoretical standpoint, previous black-box methods lack some guarantees: Daikon (Ernst et al., 2001) doesn't guarantee consistency (Proposition 1), and methods like (Padhi et al., 2016; Gehr et al., 2015) ensure consistency but not correctness (Theorem 1). From a user's perspective, providing a bias might be more intuitive than offering test cases, especially in scenarios where the source code is unavailable. Moreover, enhancing the expressiveness of B contributes to increased confidence in the obtained result. It's worth noting that previous black-box methods restrict themselves to analyzing functions that always terminate, whereas PreCA handles such cases with *ukn* classifications, adding flexibility to its applicability. Table 3 summarizes PreCA properties and compares it to other state-of-the-art black-box methods.

Alg.	Mode	Dependencies	Consistency	Soundness	Correctness	Guarantee if timeout
Daikon	passive	testcases + language	no	no	no	no
PIE	passive	testcases + language	yes	no	no	no
Gehr et al.	active	language	yes	no	no	no
PreCA	active	language	**yes**	**yes**	**yes**	**yes**

The *Guarantee if timeout* column specifies if the inference methods show guarantees even if the oracle answer *ukn*, i.e., if the execution of the code under analysis does not always terminate within the time budget.

Table 3: Summary of the black-box methods properties

5. PreCA for Memory-Related Preconditions

We now configure PreCA for the case of *memory-related preconditions*, which are crucial for the safety and security of low-level languages like C or binary code. First, we abstract program memory to manipulate only relevant variables. For that, we introduce a new constraint language specifically designed for preconditions over memory and detail how to extract a finite bias from it. Second, we provide a comprehensive description of the oracle. Third, we introduce two optimizations aimed at accelerating the acquisition process.

5.1 Constraint Acquisition Settings

Representing memory. Applying CA to precondition inference requires mapping the initial memory state of a function to a set of acquisition variables. However, creating a variable for each memory cell is impractical, as many would remain unconstrained. Therefore, our approach involves abstracting each function input (arguments or global variables) with a concise representation that summarizes all and only the necessary data. Specifically, in the context of a function F under analysis, each input is represented as a `Cell` (defined in Listing 2). A `Cell` can be either a `GlobalCell` (linked to a global variable) or an `ArgCell`

(linked to a function argument). The `ref` field of global cells represents the address where the associated global variable is stored. Each cell also has a `stored` field of type `Store`, representing the data in the cell. This stored data can be of type `Ptr` (a pointer), `SInt` (a signed integer), or `UInt` (an unsigned integer). Pointers can be general pointers (`VoidPtr`) or null-terminated strings (`StrPtr`). Both types have a `val` field representing the address pointed to by the pointer. The `StrPtr` type extends the `VoidPtr` type with a `len` field representing the size of the string.

Vocabulary *(X, D)*. With this formalization, we can systematically map each field (`ref`, `val`, `len`) to a meaningful acquisition variable and define its domain. For instance, consider a string global variable named `str`; we would create three acquisition variables: `str.ref`, `str.val`, and `str.len`. Each of these variables is then associated with a specific domain.

Let $r_1, ..., r_m$ represent the addresses of each global variable in scope, where $a_1 < ... < a_k$ denote k new valid addresses, ensuring that for all $j < i$, $a_i - a_j > 1$.[6] Let N_I (resp. N_U) be the number of signed (resp. unsigned) integer inputs (arguments or global variables) used as inputs for the function under analysis (e.g., in Listing 1, the function `find_first_of` takes two signed integers as arguments, so $N_I = 2$) The operator $dom(x)$, when applied to a variable x, returns as follows:

- For a reference (`Ptr.ref`), its domain is a singleton subset of $\{r_i\}_{1 \leq i \leq m}$, ensuring that two distinct global variables have disjoint reference domains (i.e., `G1.ref` $\neq$ `G2.ref`).
- For `Ptr.val`, the domain is $\{NULL, r_1, ..., r_m, a_1, ..., a_k\}$.
- For `StrPtr.len`, the domain is $\{0, 1\} \cup \{a_i - a_j \mid j < i\}$.
- For `SInt.val` and `UInt.val`, their domains respectively are $[\,-N_I\,,\,N_I\,]$ and $[\,0\,,\,N_U\,]$.

```
type Cell = GlobalCell | ArgCell

type GlobalCell = {ref: address; stored: Store }
type ArgCell = {stored: Store}

type Store = Ptr | SInt | UInt
type Ptr = VoidPtr | StrPtr

type VoidPtr = {val: address}
type StrPtr = {val: address; len: unsigned int}

type SInt = {val: int}
type UInt = {val: unsigned int}

type address = unsigned int
```

Listing 2: Cell type definition

[6] The constraint $a_i - a_j > 1$ is necessary to enable strings of non-null size.

Language Γ. Over this vocabulary, PreCA operates within the constraint language Γ, detailed in Fig. 3. Note that: (i) Conjunction of constraints is not listed, as the acquisition process infers it. (ii) Γ includes disjunctions of arbitrary size, crucial for learning conditional preconditions. For instance, the `find_first_of` weakest precondition in Listing 1 involves the constraint ¬`sle0(n)` ⇒ `valid(a)`.

Grammar of Γ			
C	:=	$C \vee C \mid A \mid \neg A$	
A	:=	`valid(p)` | `alias(p, q)` | `deref(p, g)`	
	|	`strleneq0(p)` | `stroverlap(p, q)` | `ptrgt(p, q)`	
	|	`seq0(i)` | `ueq0(i)` | `slt0(i)` | `sle0(i)`	
	|	`seq(i, j)` | `ueq(i, j)` | `slt(i, j)` | `ult(i, j)`	
Semantics of constraint over pointer cells			
`valid (p: Ptr)`	≡	`p.val` $\neq NULL$	p points to valid memory
`alias (p: Ptr) (q: Ptr)`	≡	`p.val = q.val`	p, q point to the same memory
`deref (p: Ptr) (g: GlobalCell)`	≡	`p.val = g.ref`	p points to g
`strleneq0 (p: StrPtr)`	≡	`p.len` = 0	the string has a null size
`stroverlap (p: StrPtr) (q: StrPtr)`	≡	`p.val` ≤ `q.val` ≤ `p.val + p.len` ∨ `q.val` ≤ `p.val` ≤ `q.val + q.len`	the strings p and q overlap
`ptrgt (p: Ptr) (q: Ptr)`	≡	`p.val > q.val`	p refers to a higher address than q
Semantics of constraint over integer cells			
`seq0 (i: SInt)`	≡	`i.val` = 0	the signed integer equals 0
`ueq0 (i: UInt)`	≡	`i.val` = 0	the unsigned integer equals 0
`slt0 (i: SInt)`	≡	`i.val` $<_s$ 0	the signed integer is lower than 0
`sle0 (i: SInt)`	≡	`i.val` $\leq_s$ 0	the signed int is lower or equal to 0
`seq (i: SInt) (j: SInt)`	≡	`i.val = j.val`	the two signed integers are equal
`ueq (i: UInt) (j: UInt)`	≡	`i.val = j.val`	the two unsigned integers are equal
`slt (i: SInt) (j: SInt)`	≡	`i.val` $<_s$ `j.val`	the signed integer i is lower than j
`ult (i: UInt) (j: UInt)`	≡	`i.val` $<_u$ `j.val`	the unsigned integer i is lower than j

Each constraint, being an arithmetic constraint over integers, is negated by negating the arithmetic constraint itself. For instance, ¬`valid(p)` is represented as `p.val` = `NULL`.

Figure 3: Grammar of constraint language Γ

Bias *B*. The bias B is a finite set of constraints extracted from Γ. Balancing the size of B is crucial, as a larger bias is more expressive but can potentially slow down the acquisition process. To optimize the bias size, we should especially take care of the disjunctions that are included in the bias i.e., decide which constraints of the form $c_1 \vee ... \vee c_n$, with c_i being atomic, must be considered. With Γ_n being the restriction of Γ to disjunctions of size up to n,[7] PreCA employs the heuristic: *"Let i be the number of integer inputs in F, and $size = \max(i, 1) + 1$. PreCA includes all constraints from Γ_{size} in its bias."* This heuristic is based on the observation that the validity of a pointer is often conditioned by constraints over integer variables. The Extract function in Algorithm 4, called in Algorithm 2 at line 2, implements this heuristic. The function Extract initializes Δ as the set of all inputs (arguments and global variables) of the function under analysis and categorizes them into three disjoint sets: Δ_I, Δ_P, and Δ_S, depending on their input types. It also populates Δ_G with the inputs representing global variables such that $\Delta_G \subset (\Delta_I \cup \Delta_P \cup \Delta_S)$. From line 8 to line 14, it adds to X the acquisition variables associated with each input. $\mathcal{R}$ is then extended with mandatory constraints over these inputs (i.e., constraints that must hold to run a function). At line 15, it determines the maximum disjunction size to include in the

[7] Γ_n being the maximal subset of constraints from Γ with disjunctions of size $\leq n$, it is unique

Algorithm 4: Extract

In : A function F; a constraint language Γ;
Out : Vocabulary $\langle X, D\rangle$; Bias B; Mandatory constraints $\mathcal{R}$;

1 **begin**
2 $X \leftarrow \emptyset$, $D \leftarrow \emptyset$, $\mathcal{R} \leftarrow \emptyset$
3 $\Delta \leftarrow F$ inputs;
4 $\Delta_I \leftarrow \{i \in \Delta \ s.t.\ i$ is a `Integer`$\}$
5 $\Delta_P \leftarrow \{i \in \Delta \ s.t.\ i$ is a `VoidPtr`$\}$
6 $\Delta_S \leftarrow \{i \in \Delta \ s.t.\ i$ is a `StrPtr`$\}$
7 $\Delta_G \leftarrow \{i \in \Delta \ s.t.\ i$ is a `GlobalCell`$\}$
8 **foreach** $i \in \Delta_I \cup \Delta_P$ **do** $X \leftarrow X \cup \{i.val\}$
9 **foreach** $i \in \Delta_S$ **do**
10 $X \leftarrow X \cup \{i.val, i.len\}$
11 $\mathcal{R} \leftarrow \mathcal{R} \cup \{\neg$`valid`$(i) \Rightarrow$`strleneq0`$(i)\}$
12 **foreach** $i \in \Delta_G$ **do** $X \leftarrow X \cup \{i.ref\}$
13 **foreach** $(i, j) \in (\Delta_S)^2 \ s.t.\ i \neq j$ **do**
14 $\mathcal{R} \leftarrow \mathcal{R} \cup \{$`stroverlap`$(i, j) \Rightarrow i.val + i.len = j.val + j.len\}$
15 $size \leftarrow \textsc{Max}(|\Delta_I|, 1) + 1$
16 $D \leftarrow \prod_{x_i \in X} dom(x_i)$
17 $B \leftarrow$Extension$(\Gamma_{size}, \langle X, D\rangle)$
18 **return** $(\langle X, D\rangle, B, \mathcal{R})$

bias. The domain for each variable in X is computed at line 16. Finally, it computes the extension of Γ_{size} on $\langle X, D\rangle$, representing all constraints c for which $var(c)$ is a tuple of variables of X, and there exists a relation $r \in \Gamma_{size}$ such that $rel(c) = r \cap D^{|var(c)|}$ (line 17).

Mandatory constraints. Traditional CA considers all variable assignments as meaningful. However, in program analysis and using PreCA, we must ensure that these assignments align with the execution of the function under analysis. To do this, the Extract function (Algorithm 4) returns a set of mandatory constraints denoted as $\mathcal{R}$. These constraints will be enforced during query generation. In the context of our memory representation, $\mathcal{R}$ includes constraints for all pairs of strings (p, q), such as:

1. `¬valid(p)` ⇒ `strleneq0(p)`: This constraint ensures that if `p` is not a valid string, its length should be zero.
2. `stroverlap(p, q)` ∧ `ptrgt(p, q)` ⇒ `p.val` − `q.val` = `p.len` − `q.len`: This constraint enforces that if PreCA generates overlapping strings (`stroverlap(p, q)`) with `p` a pointer greater than `q` (`ptrgt(p, q)`), as strings are null-terminated, their end address must be equal: it is the address of the first 0 encountered.

5.2 Oracle for Memory-Related Preconditions

In this section, we present the process of defining an oracle to ensure that PreCA accurately infers memory-related preconditions.

Example classification. To verify if an example e satisfies the weakest precondition to be learned, our oracle executes the function F under analysis with e as a test case. The classification is determined based on the execution behavior by evaluating the output w.r.t. the postcondition Q. Now, as the termination problem is undecidable (Rice, 1953), the oracle can not classify an example leading to a diverge state ($F/e \rightsquigarrow^*$) in finite time. To ensure runOracle termination, the runOracle function is called with a time limit. If the function execution reaches the predefined time limit, it is artificially halted to guarantee termination, and the oracle responds with *ukn*. Choosing an appropriate execution time limit is essential. If the time limit is too restrictive, all examples might be classified as *ukn*, while an important time limit could slow down the acquisition process. If the execution terminates within the time limit, the real function behavior is observed, and a precise classification is returned. Specifically, if $F/e \rightsquigarrow^* skip/s$ and $s \models Q$, the oracle returns *yes*. It responds with *no* if $F/e \rightsquigarrow^* skip/s$ and $s \not\models Q$, or if $F/e \rightsquigarrow^* p/s$ where p/s is stuck, meaning the execution triggers a runtime error.

Handling the examples. An example is expected to provide assignments for all function inputs. However, providing a complete assignment for the entire initial memory is impractical. To tackle this, each function input is abstracted as a `Cell` (Section 5.1), capturing only the relevant, i.e., constrained, part of the function input. The oracle acts as an emulator, initializing all cells represented by variables in X using the example values (the test case). The other memory cells, left unspecified and not in X, can be randomly initialized. In practice, if the weakest precondition is representable by the bias B (a fundamental assumption of Proposition 3 and Theorem 1), the value of a non-represented cell in X does not impact the observed execution behavior. Otherwise, it would be represented by a variable in X and constrained in B. For example, when a function takes a string `s` as input, only its value `s.val` and the referenced string size `s.len` are sent to the oracle. As no acquisition variable in X represents the content of the referenced buffer, it can not be specified. Upon receiving `s.val` and `s.len`, the oracle generates a random string of size `s.len` and writes it at address `s.val`. During execution, if a non-represented memory cell in X is read, it is assigned a random value.

The operational semantics. The definition of (weakest) preconditions relies on the provided operational semantics, which outlines the potential behaviors of configurations (termination, divergence, being stuck). The runOracle specification is also attached to this operational semantics. In our context, the binary is executed solely for precondition inference. We can easily identify access to invalid memory regions or division by zero errors as they trigger signals. However, at the binary level, there is no information about variables and their sizes. Consequently, a buffer overflow, often considered undefined behavior in the C standard, typically does not result in a runtime error. Hence, it may not lead to a negative answer. In practice, we adopt a straightforward operational semantics that only detects errors in cases of division by zero and when accessing an address outside the readable and writable address range.

5.3 Speeding up PreCA

We describe now two heuristics to speed up PreCA:

Leveraging background knowledge to accelerate CA convergence. A set of inference rules, often simply called rules, over Γ forms a background knowledge K (Bessiere et al., 2017). To illustrate this concept, a subset of these rules, focusing on pointer and integer variable usage, is presented in Fig. 4. This subset includes typical boolean properties, transitivity relations over integers, and relationships over memory. For instance, if p_1 is valid and p_1 aliases with p_2, then p_2 is also valid.

Preprocessing. Understanding code behavior when pointers alias, overlap, or dereference undesired variables is recognized as a challenging task. In such cases, functions are presumed to be more likely to trigger runtime errors or violate postconditions, resulting in *no* or *ukn* answers. However, examples classified as negative or unknown prune a smaller part of the search space compared to positive ones. While a positive example e eliminates all constraints in $\kappa(e)$, an example classified as negative or unknown e' only indicates that at least one constraint from $\kappa(e')$ can be part of the precondition. The lines 2-8 are introduced to tackle this in GenTestcase (Algorithm 3), which utilizes two sets of constraints, $\mathcal{H}$ and Θ, to generate likely positive examples. The $\mathcal{H}$ set is populated with constraints likely inconsistent with the precondition, i.e., constraints representing challenging input states. In Algorithm 3, line 3 ensures that the generated example verifies at most one constraint from $\mathcal{H}$. When an example verifying a combination of hint constraints is generated, Θ is updated at lines 6 and 7, preventing the generation of examples that verify exactly the same hint constraints. This mechanism facilitates the generation of examples satisfying at most one hint constraint. We believe such examples are less likely to be classified as negative or unknown because they address at most one hard-to-comprehend case. In practice, the set of hint constraints is generated by GetHints and should be tailored to the application scenario. In our context, the set of hint constraints includes $\neg$`valid`, `alias`, `overlap`, and `deref` constraints.

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

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

Figure 4: Background knowledge K (a subset): the two first rules are general logical rules, then there are two rules over integer variables, and finally, rules over pointers.

5.4 Example

We run PreCA over the `sum` function (Listing 3) and postcondition $Q = true$. For simplicity, we restrict Γ's atomic relations to `valid(l)`, $\neg$`valid(l)`, `ueq0(n)`, $\neg$`ueq0(n)` and consider an empty background knowledge. The `l` input is mapped to type `VoidPtr`. The function `sum` takes one integer input `n`, so the bias integrates disjunctions of size ≤ 2, i.e., $B = \{$`valid(l)`, $\neg$`valid(l)`, `ueq0(n)`, $\neg$`ueq0(n)`, $\neg$`ueq0(n)` $\Rightarrow$ `valid(l)`, `ueq0(n)`

```
int sum(int* l, uint n) {
    int res = 0;
    for (unsigned int i = 0; i<n; i++) res = res + l[i];
    return res;
}
```

Listing 3: Sum function

$\Rightarrow$ `valid(l)`, $\neg$`ueq0(n)` $\Rightarrow$ $\neg$`valid(l)`, `ueq0(n)` $\Rightarrow$ $\neg$`valid(l)`$\}$. Note that B shows no redundancies. For example, the constraint "`valid(l)` $\Rightarrow$ $\neg$`ueq0(n)`" is not in B as it would be redundant with "`ueq0(n)` $\Rightarrow$ $\neg$`valid(l)`". Moreover, `l.val` and `n.val` take values in $\{NULL, a_1\}$ and $[0, 1]$ respectively.

The examples generated by PreCA are summarized in Table 4. At first, PreCA relies on the preprocessing to generate the two first examples. The first one is classified positively as $sum(a_1, 1)$ terminates without raising a runtime error. The second one, however, is classified negatively. Indeed, `sum(NULL, 1)` $\rightsquigarrow^*$ (`res += l[i]`)$; p/s[$ `l` $\leftarrow$ `NULL`, `i` $\leftarrow 0]$ which is a stuck configuration – dereferencing a NULL pointer raises a runtime error. Finally, the third one is classified positively as the loop is never executed and the NULL pointer is so never dereferenced. Finally, the solution extracted from T_3 is $\{\neg$`ueq0(n)` $\Rightarrow$ `valid(l)`$\}$.

Queries		Version Space (Ω)
$\langle l.val, n.val\rangle$	Answer	
$\langle a_1, 1\rangle$	yes	$\Omega_1 = \neg a(\neg valid(l)) \wedge \neg a(ueq0(n)) \wedge \neg a(\neg ueq0(n)) \Rightarrow \neg valid(l)$
$\langle NULL, 1\rangle$	no	$\Omega_2 = \Omega_1 \wedge (a(valid(l)) \vee a(ueq0(n)) \vee a(\neg ueq0(n) \Rightarrow valid(l)))$
$\langle NULL, 0\rangle$	yes	$\Omega_3 = \Omega_2 \wedge \neg a(valid(l)) \wedge \neg a(\neg ueq0(n)) \wedge \neg a(ueq0(n) \Rightarrow valid(l))$

Table 4: PreCA over the `sum` function

6. Experimental Evaluation

We implemented PreCA in Java, making use of the Choco constraint solver (Prud'homme, Fages, & Lorca, 2014) and the MiniSat SAT solver (Eén, 2006). PreCA is fully available as an open-source project at the following address: https://github.com/binsec/preca. To evaluate PreCA, we performed an experimental evaluation to answer the following Research Questions:

RQ1 *Can* PreCA *handle realistic functions?* We evaluate PreCA capability to infer *weakest preconditions* on our benchmark, including real C function from prior work and standard libraries;

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 also consider 3 "hard" scenarios: no source code, obfuscated code, and presence of inline assembly.

RQ5 *How does* PreCA *behave with the bias size?* We evaluate PreCA over our dataset with three biases ordered by size.

RQ6 *How does* PreCA *behave with the disjunctions size?* We evaluate PreCA over our dataset with increasingly big disjunctions.

6.1 Experimental Design

Benchmark. Our benchmark considers 50 real C functions. It contains all functions from string.h, all functions from (Seghir & Kroening, 2013; Sankaranarayanan, Chaudhuri, Ivančić, & Gupta, 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, Foster, Goradia, & Ostrand, 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. $Q = valid(ret)$ for pointers, and $Q = ret \neq 0$ or $Q = ret > 0$ for integers. Finally, six functions return no output and are discarded from the explicit postcondition setting. In total, our benchmark contains 94 inference tasks, 50 with implicit postconditions and 44 with explicit postconditions.

Bias. To answer **RQ1**, **RQ2**, **RQ3**, and **RQ4**, we consider the bias presented in Section 5.1. We include all the atomic constraints, except the ones on strings (`stroverlap`, `strleneq`), which are used later to extend the evaluation of PreCA. Indeed, to answer **RQ5**, we consider the different bias configurations presented in Table 5. The *B1* configuration considers biases with only constraints and variables requested to express the inferred preconditions. The *B2* configuration considers biases with only requested constraints but applied to all combinations of variables. The *B3* configuration considers all possible constraints from Γ – except the strings constraints – applied to all combinations of variables. Finally, the *B4* configuration considers all possible constraints from Γ – including the strings constraints – applied to all combinations of variables. Thus, given a function under analysis, we have that $B1 \subset B2 \subset B3 \subset B4$.

	min size	max size	mean size
B1	2	14	4.6
B2	2	16	5.3
B3	2	42	11.1
B4	4	64	18.6

Table 5: Statisitics of the biases (in terms of atomic constraints) used for precondition inference

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). As passive learning approaches highly depend on the given examples, we ran them 10 times, each time with 100 new random

examples. 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 Tables 6 and 7.

RQ1. We study now if PreCA can handle the realistic functions from our benchmark. To evaluate it, we apply it to our dataset of real-world C functions and recorded the results for different timeouts. We observe in Table 6 that 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**).

Conclusion: 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. We perform an ablation study to understand the impact of the different PreCA components, especially, the background knowledge and the preprocess. First, we consider PreCA in passive mode, with 100 random queries, to see the impact of active learning (denoted ↳ Random in Table 6). 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 speed up.

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

RQ3. We evaluate if PreCA is competitive with state-of-the-art black-box approaches, namely Daikon (Ernst et al., 2001), PIE (Padhi et al., 2016)[8] and Gehr et al. approach (Gehr et al., 2015) – 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 the queries PreCA generated in active mode (lines ↳ PreCA and ↳ Both). All methods except Daikon benefit from it, highlighting the quality of PreCA sample generation mechanism.

Conclusion: 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.*

[8] PIE is used as publicly released, with its generic (expressive enough) grammar

RQ4. We evaluate if PreCA is competitive with white-box approaches. To do so, we compared to the white-box method P-Gen (Seghir & Kroening, 2013). We also considered (Kafle, Gallagher, Gange, Schachte, Søndergaard, & Stuckey, 2018) and (Gulwani, Srivastava, & Venkatesan, 2008), but the former requires manually translating 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 (Table 6). 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.

Conclusion: As expected, PreCA *significantly outperforms P-Gen on hard application scenarios. Less expected, it also performs better when 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) the 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 (highlighted), 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 6: PreCA against the state-of-the-art depending on the time budget

RQ5. We now evaluate how the bias choice impacts PreCA. Table 7 presents the results of PreCA over the four bias sizes presented in Table 5. We restrict the analysis to the implicit postcondition case because this is the context where we have the most granularity. Interestingly, for the 1h timeout, we observe that PreCA's results (Heur. column) are sim-

ilar over the four cases, inferring between 45/50 and 46/50 weakest preconditions. However, for smaller timeouts (lower than 5 minutes), we observe that the bias size has an impact. Especially, PreCA infers in 1s 33/50, 31/50, 24/50, and 8/50 weakest preconditions for B1, B2, B3 and B4, respectively. Such behavior can also be observed on the different variations of PreCA with fixed disjunction sizes, except in the "No disj" case where only the most simple functions can be handled. On all these configurations, for the 1s second timeout, PreCA can infer up to 37/50 weakest precondition over the minimal bias, while it cannot infer more than 16/50 over the maximal bias. Also, observe that PreCA (Heur. column) infers in 1h more preconditions over B4 than B3. Indeed, B4 is expressive enough to handle one new function, namely `strcat` (see Section 7).

Conclusion: Giving more knowledge to PreCA*, by reducing its input bias, enables it to speed up its inference, leading to substantial improvement for timeouts lower than 5 minutes. For bigger timeouts (e.g. 1h),* PreCA *is efficient enough to handle the maximal bias, as well as the smaller ones.*

RQ6. We compare how PreCA behaves when integrating in the bias fixed size disjunctions and compare it to our heuristic given in Section 5.1. From Table 7, we observe that the proposed heuristic leads to the best results for the 1h timeout – over each bias size. This is expected because the heuristic includes different disjunction sizes depending on the target function. Each acquisition task has, as such, a tailored bias. We also observe that PreCA infers less weakest preconditions over B3 and B4 when including disjunctions of sizes up to 7 and 10. Indeed, over B3 it infers with disjunctions of size up to 3, 45/50 weakest preconditions against 35/50 for disjunctions of size up to 7 and 10. This is expected as the bias size explodes, impacting the acquisition efficiency. However, we do not observe such behaviors over the smaller biases as they stay small enough to be manageable by PreCA.

Conclusion: PreCA *is impacted by the disjunction sizes that have to be carefuly selected. Still, the proposed heuristic which includes in the bias the maximum size of disjunctions is efficient and leads to better results than specifying the maximal boundary.*

7. Use Cases

We now show that PreCA can be used to understand non-trivial function behaviors. We consider two use-cases: `strcat` from the libc and `mbedtls_md_setup` from the mbedtls library[9].

7.1 The `strcat` Usecase

The `strcat` function is a standard function that manipulates strings. It takes two null-terminated strings as input (`dst` and `src`) and appends the `src` string to `dst`. To do so, it copies the ith character of `src`, at address `dst + strlen(dst) + i` – this removes the `'\0'` from `dst`. We want to know, which inputs lead to non-crashing and terminating behaviors, i.e., $Q = true$. We chose this example as it is simple and broadly used but still presents unexpected behaviors that may be misunderstood by developers. Indeed, the manual states that *"The strings may not overlap"*. Thus, we could expect the weakest precondition to be `valid(dst)` $\wedge$ `valid(src)` $\wedge\neg$`stroverlap(dst, src)`.

Interestingly, this is too conservative, and PreCA *is able to find a better precondition.*

[9]https://github.com/Mbed-TLS/mbedtls

		Heur.	No disj	$\|disj\| \leq 2$	$\|disj\| \leq 3$	$\|disj\| \leq 4$	$\|disj\| \leq 7$	$\|disj\| \leq 10$
B1	1s	33/50	21/50	**37/50**	30/50	30/50	30/50	30/50
	5s	**45/50**	21/50	43/50	43/50	42/50	42/50	43/50
	5 mins	**46/50**	21/50	44/50	**46/50**	**46/50**	45/50	**46/50**
	1h	**46/50**	21/50	44/50	**46/50**	**46/50**	45/50	**46/50**
B2	1s	31/50	21/50	**34/50**	26/50	26/50	27/50	29/50
	5s	**44/50**	21/50	42/50	42/50	41/50	41/50	42/50
	5 mins	**46/50**	21/50	44/50	**46/50**	45/50	44/50	45/50
	1h	**46/50**	21/50	44/50	**46/50**	**46/50**	44/50	45/50
B3	1s	**24/50**	20/50	22/50	18/50	18/50	18/50	17/50
	5s	36/50	21/50	**38/50**	31/50	29/50	28/50	27/50
	5 mins	**44/50**	21/50	**44/50**	42/50	35/50	35/50	35/50
	1h	**45/50**	21/50	44/50	44/50	40/50	35/50	35/50
B4	1s	8/50	**16/50**	10/50	5/50	5/50	5/50	5/50
	5s	**28/50**	20/50	**28/50**	17/50	15/50	14/50	14/50
	5 mins	42/50	20/50	**44/50**	39/50	27/50	19/50	19/50
	1h	**46/50**	20/50	45/50	44/50	32/50	19/50	19/50

Table 7: PreCA depending on the time budget, the disjunction sizes and the bias used.

Indeed, PreCA synthesizes the "real" weakest precondition, which highlights that the two strings can overlap when `src` is the empty string. As `strcat` has no integer inputs, the bias integrates disjunctions of size up to 2. This led to a bias with 72 constraints. Over it, PreCA generates 13 queries (4 from the preprocess) and returns the learned concept in 126s. On average, PreCA takes 98ms to generate a query. While most queries are answered by the oracle in less than 1s, two queries take a lot more time as they reach the given timeout (60s). Indeed, over these queries, the non-empty strings `dst` and `src` overlap. Thus, by appending the `src` characters to `dst`, it also appends the characters to the `src` string, and execution never terminates. This is coherent with the manual. However, there is also a query where the two strings overlap but where execution succeeds. Indeed, when `src` is the empty string, only the `'\0'` character is appended to `dst`, and nothing happens. So finally, PreCA infers the weakest precondition: `valid(dst)` $\wedge$ `valid(src)` $\wedge$ (`strleneq0(src)` $\vee$ $\neg$`stroverlap(dst, src)`).

Conclusion: PreCA *can infer subtle weakest preconditions over strings, which can even be more precise than human-given documentation.*

7.2 The `mbedtls_md_setup` Usecase

The mbedtls library implements cryptographic functions in C for embedded systems. We especially focus on function `mbedtls_md_setup`, which sets all internal structures to apply *message disgest* (i.e., cryptographic hash) algorithms and returns 0 if everything goes well. We want to know over which inputs this function returns 0. `mbedtls_md_setup` takes `ctx`, `md_info`, and `hmac` as arguments. The `ctx` argument is a pointer to a complex structure called `mbedtls_md_context_t`; the `md_info` is also a pointer to a structure called `mbedtls_md_info_t`; and the `hmac` is a signed integer. Currently, PreCA cannot automatically handle structures, and we need to specify which fields will be converted to acquisition variables. By quickly

reading the code, we observe that only a few portions of the structure fields are in fact manipulated. Thus, we restrict our set of acquisition variables to these 5 ones. One of these fields is `md_info->type`. This is an enumeration specifying the 8 different types of message digest algorithms that can be used (e.g., MD5, SHA1 and more). Thus, we add to the bias the comparison of `md_info->type` to the bound values of the enumeration i.e., `ueq8(md_info ->type)`, `ugt8(md_info->type)` (defined as `ueq0(i)`, `ugt0(i)`, but for the constant value 8) and their negations.

From such a setting, PreCA can infer a meaningful precondition. Because there is only one integer argument in the function prototype (`hmac`), the bias includes disjunctions of size up to 2. This led to a bias with 722 constraints. Over it, PreCA generates 426 queries (4 from the preprocess) and returns the learned concept in 9min 46s. On average, PreCA takes 488ms to generate a query. Finally, it returns the precondition: `valid(ctx)` $\wedge$ `valid (md_info)` $\wedge$ $\neg$`alias(ctx, md_info)` $\wedge$ `ugt0(md_info->type)` $\wedge$ `ult8(md_info->type)` $\wedge$ (`seq0(hmac)` $\vee$ `ugt0(md_info->blocksize)`). Note that the 4th and 5th constraints state that `md_info->type` should be between 1 and 7. This is coherent as the enumeration can take value in 0..7, and the 0 case stands for `MBEDTLS_MD_NONE`. This case means that no message digest algorithm was specified and led to an error return value.

Conclusion: PreCA *can handle complex functions from cryptographic code. Still, there are two challenges: 1. the high number of acquisition variables; 2. the handling of constant values. A simple but effective solution involves filtering unused variables and searching for used constant values, e.g., enumerations. This could be performed automatically with basic static analysis*

8. Discussion

We now explain how PreCA can be adapted and extended to handle new contexts. We then discuss its limitations and present promising research directions.

PreCA on new programming languages. The PreCA approach can be applied to different precondition inference tasks. In this paper, we show that it can infer preconditions of binary code functions. Still, it can be applied to high-level languages such as JAVA or Python. The main step to applying PreCA on a programming language is to create a new oracle for this language. It should initialize the input memory state of the function under analysis based on a given query and monitor errors and execution time. Still, building a new oracle may not be enough. The constraint language might also have to be extended with new constructs tailored for the new programming language to analyze. Eventually, the mandatory constraints could also need adaptation as the memory model axioms depend on the programming language and types considered.

PreCA with new constraints. The constraint language used by PreCA establishes the type of precondition it can infer. In Fig. 3, the given constraint language aims to infer preconditions about memory for binary code. Thus, when applied to other contexts (e.g., high-level languages or other types of precondition), it might contain too low-level constraints or miss some others. For example, the `stroverlap` constraint may not be necessary in JAVA as two different string objects cannot overlap. On the other hand, constraints over lists or sets could be meaningful. To extend the constraint language, the memory

representation (`Cell`, `Ptr`, etc) should also be adapted accordingly to include all the necessary data asked by the constraints. Moreover, as PreCA needs to reason about the constraints from Γ, the user has also to provide a checker for the constraint and a solver for CNFs of the given constraints – we have to handle CNFs because of disjunctions generated in the bias. Finally, the Extract and GetHints procedures have to be adapted for the new constraints.

Beyond finite domains. In this paper, the domain of each variable is finite. This is the usual CA setting, and it easily ensures the feasibility and termination of the method, by making the underlying theory decidable. All our proofs lift to the infinite domain cases as long as the satisfiability and the model generation problems are decidable over the underlying theory. If so, we can prove all the properties from Section 4.3. Indeed, the real crucial assumption is to have a finite bias. It is even possible to use PreCA on an undecidable theory. In such a case, we lose the termination property as the solver may never terminate. Still, if the acquisition terminates we keep the guarantees from Propositions 1 and 3. Systematic evaluation of constraint acquisition and PreCA over infinite domains like streams or natural numbers is left as a future work.

Limitations. While PreCA shows overall good properties, it also comes with a few limitations. First, handling constant values is problematic. Indeed, constraints holding over these constant values are not necessarily present in the bias and might thus remain undiscoverable. However, in a black-box context, there is no reason to select one constant instead of another. Adding all constraints corresponding to all values in a finite domain may rapidly lead to an unbearable combinatorial explosion. 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. Third, requiring a user-defined constraint grammar is both a strength and a weakness. On the one hand, it renders PreCA easy to extend and applicable to a broader context. On the other hand, it lets the user define the necessary language to handle each specific use case. Finally, we require the function under analysis to be deterministic (a common assumption in the field). Going further remains open.

Future work. As presented previously, a major limitation of fully black-box approaches is the poor handling of constant values. Still, in different use cases (e.g., cryptographic functions) handling them is needed. Thus, it would be beneficial to design methods to integrate useful constant values in arithmetic constraints. This could be achieved naively in the grey-box scenario, which combines black- and white-box analysis. By parsing the code, grey-box methods could extract all constant values. Still, more subtle approaches could also be considered relying on symbolic execution, for example. Such a grey-box scenario could also enable PreCA to get the most from its input constraint grammar Γ. Indeed, while this very flexible grammar can be easily defined by the user, in practice, PreCA works over a subset of it. Extracting such a subset of constraints can be error-prone and may depend on the context. Studying how such an extraction procedure could be automized could be beneficial to make good use of Γ and simplify PreCA usage. From a more general point of view, both program analysis and constraint acquisition could benefit from each other. Indeed, program analysis often deals with disjunctive behaviors, asks for an under/over-approximation of the target property, or relies on both static and dynamic

analysis. This could justify extensions of constraint acquisition to handle such disjunctive problems or approximations of them, possibly with a more powerful oracle, able to answer more complex queries. On the other side, different notions of constraint acquisition could be applied to program analysis. Especially other kinds of queries, like partial queries (Bessiere, Coletta, Hebrard, Katsirelos, Lazaar, Narodytska, Quimper, & Walsh, 2013), could be used by extending the oracle. Omissions in constraint acquisition (Tsouros, Stergiou, & Bessiere, 2020) could also be applied to handle our *ukn* answers.

9. 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, only uses positive test cases, and lacks clear formal guarantees (like soundness or completeness). PIE (Padhi et al., 2016) relies on program synthesis for black-box precondition inference. Garg et al. (Garg, Neider, Madhusudan, & Roth, 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 case quality. Gehr et al. (Gehr et al., 2015) perform black-box active learning. Yet, they rely on program synthesis and perform (*type-aware*) random sampling, preventing them from enjoying PreCA guarantees.

White-box dynamic contracts inference. While purely static white-box approaches (Cousot et al., 2013; Calcagno, Distefano, O'Hearn, & Yang, 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 with white-box information. Seghir et al. (Seghir & 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, Madhusudan, Saha, Wang, & Xie, 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 & Simonis, 2012) to robotics (Paulin, Bessiere, & Sallantin, 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 & Simonis, 2012; Lallouet, Lopez, Martin, & Vrain, 2010; Tsouros et al., 2020; Tsouros, Berden, & Guns, 2024) and could be explored. First, different kinds of queries exist (Bessiere, Coletta, O'Sullivan, & Paulin, 2007; Belaid, Belmecheri, Gotlieb, Lazaar, & Spieker, 2022). Especially, partial qeries (Bessiere et al., 2007) could be used. The oracle could be extended to handle these queries that do not specify all the program inputs. On the other hand, Tsouros et al. (Tsouros et al., 2020) proposed the first handling of unknown answers in CA. They extended the Quacq framework to infer the target and the unknown concepts. PreCA also needs to handle

these unknown answers, but our work distinguishes in 2 ways. First, Quacq uses partial queries, while PreCA only relies on membership queries. This impacts the underlying learning process as partial queries carry more information than membership ones. Second, Tsouros et al. do not handle *ukn* classifications as PreCA: they include them in the target concept while PreCA exclude them to ensure a precondition is found (Proposition 1).

Program synthesis. Program synthesis (Gulwani, Polozov, & Singh, 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). However, giving a representative a set of examples to the synthesizer is hard, especially when the code is unknown. Moreover, the inference result highly depends on the given example, which prevents synthesis from enjoying clear correctness guarantees. Especially, the examples selection problem, combined with complex grammar, often leads to overfitting in program synthesis (Padhi, Millstein, Nori, & Sharma, 2019) – unlike PreCA, which will generate all the needed queries through active learning.

10. Conclusion

We propose PreCA, 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. Indeed, PreCA terminates and returns a result consistent with the generated test cases and, under some assumption, returns even the weakest precondition. These are better guarantees than prior state-of-the-art precondition inference methods. Moreover, PreCA automatically and smartly generates the test cases, removing the burden from the user to give them. Our experiments for memory-oriented preconditions show that PreCA significantly outperforms prior works – including some white-box methods, demonstrating the interest of Constraint Acquisition here. Finally, we show that PreCA can infer interesting preconditions over two use cases: strcat from the libc and a function from the mbedtls library. The preconditions found are more precise than human intuition and developer documentation. Finally, CA and program analysis seem to be two cross-fertilizing domains. Considering the precondition inference problem as a CA task proves beneficial, yielding favorable theoretical properties and superior outcomes compared to previous approaches. In turn, program analysis emerges as a valuable application for CA, addressing certain limitations and introducing new challenges and opportunities to the CA community.

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; (iii) the European Union's Horizon Europe AI4CCAM project (Trustworthy AI for Connected, Cooperative Automated Mobility) under grant agreement No 10107691 and (iv) Plan France 2030 under references ANR-22-PECY-0007 and DOS0233319/00.

Appendix A. Notations

Symbol	Description	Definition section
c	a constraint	Section 3.1
$rel(c)$	the relation of the constraint c	Section 3.1
C, L, T	constraint networks i.e., sets of constraints (T stands for the target network)	Section 3.1
$sol(C)$	the set of C solutions	Section 3.1
$e \models C$	$e \in sol(C)$	Section 3.1
Γ	a constraint language	Section 3.1
Γ_{size}	restriction of the language Γ to disjunctions of size up to $size$	Section 5.1
$\neg b, \neg c, \overline{b}, \overline{c}$	negation of the boolean b and the constraint c	Section 3.1
B	bias extracted from the constraint language Γ to perform acquisition	Section 3.1
X	set of acquisition variables	Section 3.1
D^X	the product domain of the variables in X	Section 3.1
$var(c)$	the set of variables constrained by c	Section 3.1
Ω	clausal formula encoding the search space in Conacq and PreCA	Sections 3.1 and 4.2
f_T	concept encoded by the constraint network T	Section 3.1
e	an example i.e., a element of D^X	Section 3.1
$e[X]$	restriction of e to the variables in X	Section 3.1
E	a set of examples	Section 3.1
E^+	the set of examples classified as positive (subset of E)	Section 3.1
E^-	the set of examples classified as negative (subset of E)	Section 3.1
S	the sequence of queries generated by Conacq or PreCA	Section 3.1
$\kappa(e)$	the set of constraint c from B s.t., $e \not\models c$	Section 3.1
yes, no, ukn	the possible classifications of queries in PreCA	Sections 3.1 and 4.2
"$collapse$"	the result of Conacq and PreCA if no solution can be found	Section 3.1
`Ask`	the procedure to query the user in Conacq	Section 3.1
`network`	a function returning a constraint network represented by Ω	Section 4.2
$a(c)$	a boolean stating if c is in the solution	Section 3.1
$a(c_1) \longrightarrow a(c_2)$	a rule stating that is c_1 is in solution then c_2 also	Section 5.3
p/s	the application of the program p to the memory state s	Section 3.2
$\rightsquigarrow, \rightsquigarrow^*$	the operational semantics (sigle step) and its reflexive closure	Section 3.2
$p_1; p_2$	a sequence of instructions to execute	Section 3.2
$skip$	the special instruction representing the end of execution	Section 3.2
F	a program function under analysis	Section 3.3
Q	a postconditon	Section 3.3
$\mathcal{WP}(F, Q)$	the weakest precondition of F w.r.t., Q	Section 3.3
$\lvert disj\rvert \leq k$	states that PreCA includes all disjunctions of size up to k in its bias	Section 6.2

Appendix B. Operational Semantics

Example 4. *Fig. 6 describes a simple operational semantics for IMP, a simple imperative language with conditionals (`if then else`) and loops (`while`). IMP manipulates integer values and enables to check internal states at run-time through the `assert` primitive. Each code construct is associated with semantics specifying code behavior. For example, when executing $(x := a)/s$, the first rule applies and updates the memory state s by setting the variable x to the value of the arithmetic expression a (noted $s[x \leftarrow [\![a]\!]]$). Observe that there is no rule to evaluate a division by 0 or an `assert(false)`. This is used to describe stuck states. From this language and operational semantics, we can define programs describing each behavior:*

1. ***end.*** $(x := 0; (\ while\ x < 10\ do\ x := x + 1); skip)/s \rightsquigarrow^* skip/s[x \leftarrow 10]$
2. ***diverge.*** $(x := 0; (\ while\ true\ do\ x := x + 1); skip)/s \rightsquigarrow^*$
3. ***stuck.*** $(x := 0; assert(x \neq 0); skip)/s \rightsquigarrow^* (assert(x \neq 0); skip)/s[x \leftarrow\ 0]$

Appendix C. Weakest Precondition Calculus

Weakest precondition calculus. Automatically computing *weakest preconditions* has been a strong drive for program analysis since the 70's. Given an instruction, weakest precondition calculus specifies how to compute the weakest precondition through rules given in Fig. 5. Yet, as the whole problem is undecidable, standard approaches must rely on manual annotations, renouncing to full automation. Indeed, the rule handling loops in Fig. 5 needs a loop invariant I to prove specification and a loop variant ν to prove termination. Such annotations must be given by the user, which is a hard task.

Figure 5: Weakest precondition calculus deduction rules

$\mathcal{WP}(skip, Q)$	$\equiv$	Q	$\mathcal{WP}(x := e, Q)$	$\equiv$	$Q[x \leftarrow e]$
$\mathcal{WP}(assert(b), Q)$	$\equiv$	$Q \wedge b$	$\mathcal{WP}(p_1; p_2, Q)$	$\equiv$	$\mathcal{WP}(p_1, \mathcal{WP}(p_2, Q))$
$\mathcal{WP}(\text{if } b \text{ then } p_1 \text{ else } p_2, Q)$	$\equiv$	$(b \Rightarrow \mathcal{WP}(p_1, Q)) \wedge (\neg b \Rightarrow \mathcal{WP}(p_2, Q))$			
$\mathcal{WP}(\text{while } b \text{ invariant } I \text{ variant } \nu, \prec \text{ do } p, Q)$	$\equiv$	$I \wedge \forall x_1, ..., x_k, \xi, (I \wedge b \wedge \xi = \nu \Rightarrow \mathcal{WP}(e, I \wedge \xi \prec \nu))$ $\wedge\ (I \wedge \neg b \Rightarrow Q)$			

Where x_j are references modified in the loop body.

```
x := a ;  i := 0 ;
while  i < 10  do
      x := x + 1;
      i := i + 1;
```

Listing 4: Program example

Example 5. Consider the postcondition $Q : x = 10$ for the program c in Listing 4. To handle the loop, WP calculus needs the user to give the loop invariant $I : x = i \wedge i \leq 10$ and the variant $\nu : 10 - i$. Then, the rule over loops infers that $\mathcal{WP}(\text{while } i < 10 \text{ do } ...) \equiv x = i \wedge i \leq 10$. Then we can simply compute that $\mathcal{WP}(i := 0, x = i \wedge i \leq 10) \equiv (x = 0)$ and then $\mathcal{WP}(x := a, x = 0) \equiv (a = 0)$. Finally, by using deduction rules of weakest precondition calculus and giving the loop annotations by hand, we show that $\mathcal{WP}(c, Q) = (a = 0)$.

Figure 6: IMP language syntax and evaluation rules

Grammar

$$
\begin{array}{lcl}
a & := & x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \mid a_1 \div a_2 \\
b & := & \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 < a_2 \mid a_1 \leq a_2 \\
p & := & \text{skip} \mid \text{x := a} \mid p_1; p_2 \mid \text{if } b \text{ then } p_1 \text{ else } p_2 \mid \text{while } b \text{ do } p \mid \text{assert(b)}
\end{array}
$$

Evaluation rules

Programs:

$$\frac{}{(x := a)/s \rightsquigarrow skip/s[x \leftarrow [\![a]\!]]} \qquad \frac{}{(skip; p)/s \rightsquigarrow p/s} \qquad \frac{p_1/s \rightsquigarrow p_1'/s'}{(p_1; p_2)/s \rightsquigarrow (p_1'; p_2)/s'}$$

$$\frac{[\![b]\!]s = true}{(\text{if } b \text{ then } p_1 \text{ else } p_2)/s \rightsquigarrow p_1/s} \qquad \frac{[\![b]\!]s = false}{(\text{if } b \text{ then } p_1 \text{ else } p_2)/s \rightsquigarrow p_2/s}$$

$$\frac{[\![b]\!]s = true}{(\text{while } b \text{ do } p)/s \rightsquigarrow (p; \text{while } b \text{ do } p)/s} \qquad \frac{[\![b]\!]s = false}{(\text{while } b \text{ do } p)/s \rightsquigarrow skip/s} \qquad \frac{[\![b]\!]s = true}{assert(b)/s \rightsquigarrow skip/s}$$

Boolean expressions:

$$\frac{a_1/s \rightsquigarrow a_1'/s}{(a_1 \bullet a_2)/s \rightsquigarrow (a_1' \bullet a_2)/s} \qquad \frac{a_2/s \rightsquigarrow a_2'/s}{(n \bullet a_2)/s \rightsquigarrow (n \bullet a_2')/s} \qquad \frac{n \bullet m}{(n \bullet m)/s \rightsquigarrow true/s} \qquad \frac{\neg(n \bullet m)}{(n \bullet m)/s \rightsquigarrow false/s}$$

Arithmetic expressions:

$$\frac{[\![x]\!]s = n}{x/s \rightsquigarrow n/s} \qquad \frac{a_1/s \rightsquigarrow a_1'/s}{(a_1 \diamond a_2)/s \rightsquigarrow (a_1' \diamond a_2)/s} \qquad \frac{a_2/s \rightsquigarrow a_2'/s}{(n \diamond a_2)/s \rightsquigarrow (n \diamond a_2')/s} \qquad \frac{r = n \diamond m \qquad \diamond \neq \div}{(n \diamond m)/s \rightsquigarrow r/s}$$

$$\frac{m \neq 0 \qquad r = n \div m}{(n \div m)/s \rightsquigarrow r/s}$$

x is a variable, p a program, a an arithmetic expression, b a Boolean expression, n, m, r constant values, $\bullet \in \{=, \neq, <, \leq\}$, $\diamond \in \{+, \times, \div\}$ and $[\![expr]\!]$ is the evaluation of the expression $expr$.

References

Astorga, A., Madhusudan, P., Saha, S., Wang, S., & Xie, T. (2019). Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*.

Astorga, A., Srisakaokul, S., Xiao, X., & Xie, T. (2018). Preinfer: Automatic inference of preconditions via symbolic analysis. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*. IEEE.

Bardin, S., & Herrmann, P. (2011). Osmose: automatic structural testing of executables. *Software Testing, Verification and Reliability*, *21*(1), 29–54.

Belaid, M.-B., Belmecheri, N., Gotlieb, A., Lazaar, N., & Spieker, H. (2022). Geqca: Generic qualitative constraint acquisition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36, pp. 3690–3697.

Beldiceanu, N., & Simonis, H. (2012). A model seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming (CP'12)*, pp. 141–157. Springer.

Bessiere, C., Carbonnel, C., Dries, A., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C., Stergiou, K., Tsouros, D. C., & Walsh, T. (2023). Learning constraints through partial queries. *Artificial Intelligence Journal*, *319*, 103896.

Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C.-G., & Walsh, T. (2013). Constraint acquisition via partial queries. In *Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI-13)*.

Bessiere, C., Coletta, R., O'Sullivan, B., & Paulin, M. (2007). Query-driven constraint acquisition. In *International Joint Conference on Artificial Intelligence*, Vol. 7, pp. 50–55.

Bessiere, C., Koriche, F., Lazaar, N., & O'Sullivan, B. (2017). Constraint acquisition. *Artificial Intelligence Journal*, *244*, 315–342.

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., & Poll, E. (2005). An overview of JML tools and applications. *Int. Journal on Software Tools and Technology Transfer (JSTTT)*, *7*(3), 212–232.

Calcagno, C., Distefano, D., O'Hearn, P., & Yang, H. (2009). Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'09)*, pp. 289–300. ACM.

Cousot, P., Cousot, R., Fähndrich, M., & Logozzo, F. (2013). Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*. Springer.

Dijkstra, E. W. (1968). A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, *8*(3), 174–186.

Eén, N. (2006). The minisat reference pages. `http://minisat.se/`.

Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, *27*(2), 99–123.

Floyd, R. W. (1993). Assigning meanings to programs. In *Program Verification*. Springer.

Garg, P., Neider, D., Madhusudan, P., & Roth, D. (2016). Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, *51*(1), 499–512.

Gehr, T., Dimitrov, D., & Vechev, M. (2015). Learning commutativity specifications. In *International Conference on Computer Aided Verification (CAV'15)*.

Godefroid, P., Lahiri, S. K., & Rubio-González, C. (2011). Statically validating must summaries for incremental compositional dynamic test generation. In *International Static Analysis Symposium (SAS'11)*, pp. 112–128. Springer.

Gries, D. (2012). *The science of programming*. Springer Science & Business Media.

Gulwani, S., Polozov, O., & Singh, R. (2017). Program synthesis. *Found. Trends Program. Lang.*, *4*(1-2), 1–119.

Gulwani, S., Srivastava, S., & Venkatesan, R. (2008). Program analysis as constraint solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pp. 281–292.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, *12*(10), 576–580.

Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering (ICSE'94)*. IEEE.

Kafle, B., Gallagher, J. P., Gange, G., Schachte, P., Søndergaard, H., & Stuckey, P. J. (2018). An iterative approach to precondition inference using constrained horn clauses. *Theory Pract. Log. Program.*, *18*(3-4), 553–570.

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., & Yakobowski, B. (2015). Frama-c: A software analysis perspective. *Formal Aspects Comput.*, *27*(3), 573–609.

Lallouet, A., Lopez, M., Martin, L., & Vrain, C. (2010). On learning constraint problems. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'10)*, Vol. 1, pp. 45–52. IEEE.

Menguy, G., Bardin, S., Lazaar, N., & Gotlieb, A. (2022). Automated program analysis: Revisiting precondition inference through constraint acquisition. In Raedt, L. D. (Ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pp. 1873–1879. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Meyer, B. (1988). Eiffel: A language and environment for software engineering. *Journal of Systems and Software (JSS)*, *8*(3), 199–246.

Nagra, J., & Collberg, C. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.

Nepomniaschy, V. A., Anureev, I. S., & Promskii, A. (2003). Towards verification of c programs: Axiomatic semantics of the c-kernel language. *Programming and Computer Software*, *29*(6), 338–350.

Ollivier, M., Bardin, S., Bonichon, R., & Marion, J.-Y. (2019a). How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 177–189.

Ollivier, M., Bardin, S., Bonichon, R., & Marion, J.-Y. (2019b). Obfuscation: where are we in anti-dse protections?(a first attempt). In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, pp. 1–8.

Padhi, S., Millstein, T. D., Nori, A. V., & Sharma, R. (2019). Overfitting in synthesis: Theory and practice. In Dillig, I., & Tasiran, S. (Eds.), *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, Vol. 11561 of *Lecture Notes in Computer Science*, pp. 315–334. Springer.

Padhi, S., Sharma, R., & Millstein, T. (2016). Data-driven precondition inference with learned features. *SIGPLAN Not.*, *51*(6), 42–56.

Paulin, M., Bessiere, C., & Sallantin, J. (2008). Automatic design of robot behaviors through constraint network acquisition. In *2008 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'08)*, Vol. 1, pp. 275–282. IEEE.

Plotkin, G. D. (2004). A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, *60-61*, 17–139.

Prud'homme, C., Fages, J.-G., & Lorca, X. (2014). Choco documentation. Inria's TASC project-team.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, *74*(2), 358–366.

Rossi, F., Van Beek, P., & Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

Sankaranarayanan, S., Chaudhuri, S., Ivančić, F., & Gupta, A. (2008). Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA'08)*. ACM.

Schmidt, D. A. (1986). *Denotational semantics: a methodology for language development*. William C. Brown Publishers.

Seghir, M. N., & Kroening, D. (2013). Counterexample-guided precondition inference. In *European Symposium on Programming (ESOP'13)*, pp. 451–471. Springer.

Tsouros, D., Berden, S., & Guns, T. (2024). Learning to learn in interactive constraint acquisition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38, pp. 8154–8162.

Tsouros, D. C., Stergiou, K., & Bessiere, C. (2020). Omissions in constraint acquisition. In Simonis, H. (Ed.), *Principles and Practice of Constraint Programming (CP'20)*, Vol. 12333 of *Lecture Notes in Computer Science*, pp. 935–951. Springer.

Winskel, G. (1993). *The formal semantics of programming languages: an introduction*. MIT press.

Zhang, L., Yang, G., Rungta, N., Person, S., & Khurshid, S. (2014). Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM.