

Precondition Constraint Acquisition Through Explanatory Queries

Sarra Djebour^{1,2}, Sébastien Bardin¹, Arnaud Gotlieb³, Nadjib Lazaar² and Grégoire Menguy¹

¹Université Paris-Saclay, CEA, List, Palaiseau, France

²Université Paris-Saclay, LISN, Gif-sur-Yvette, France

³Simula Research Laboratory, Oslo, Norway

Abstract

The ability to automatically infer preconditions for safe program execution is a critical challenge in software verification. Recently, symbolic AI learners such as PreCA and DCA have addressed this by generating membership queries (i.e. test inputs) for an oracle to classify. Although these methods provide strong correctness guaranties, their exclusive reliance on binary feedback limits their scalability. Specifically, it creates an unavoidable trade-off between the speed of query generation (DCA) and the total number of queries required to converge (PreCA). In this paper, we extend PreCA by introducing a novel class of interactions: *explanatory queries*. Under this model, the oracle provides not only a binary classification but also a structured explanation that generalizes its decision. After formalizing this new query type and defining the requirements for compatible automated oracles, we present XAcq. XAcq uses this explanatory feedback to simultaneously reduce interaction counts and query generation time, while maintaining a strict guaranty of soundness and completeness. We validate our approach using *path predicates* as automated explanations. The results show that XAcq is on par with existing constraint-acquisition methods in simple cases but scales significantly better in harder programs, solving previously intractable functions.

Keywords

Constraint Acquisition, Precondition Inference, Path Condition

1. Introduction

Constraint acquisition (CA) is a technique from the constraint programming community originally designed to model and solve combinatorial problems [1]. By adapting CA for precondition inference, a well-known program verification problem [2], Precondition Constraint Acquisition methods PreCA [3, 4] and DCA [5] automatically extracts the precise conditions under which a program runs safely. This ensures that valid executions terminate smoothly and have no errors. Active CA automates the acquisition process through an interaction between a learner, which generates queries corresponding to full instantiations of the constraint problem’s variables (a.k.a., membership queries) [6, 1], and an oracle, which classifies each as a solution or non-solution. In the context of PreCA and DCA, the problem’s variables correspond to the program’s input variables, and the oracle’s classification translates depending on the program’s execution result into three states: safe, unsafe, or unknown. The unknown state corresponds to executions that do not terminate within a given timeout and are therefore considered unsafe by default to avoid false positives. From these classifications, the learner infers the corresponding precondition for a given program. Interestingly, delegating the query classification task to an automated oracle can advantageously classify thousands of queries in seconds, enabling faster acquisitions.

Goal and Challenges.

RuleML+RR’26: Companion Proceedings of the 10th International Joint Conference on Rules and Reasoning, August 24–26, 2026, Vilnius, Lithuania

✉ sarra.djebour@cea.fr (S. Djebour); sebastien.bardin@cea.fr (S. Bardin); arnaud@simula.no (A. Gotlieb); lazaa@lisn.fr (N. Lazaar); gregoire.menguy@cea.fr (G. Menguy)

🆔 0009-0007-1013-2188 (S. Djebour); 0000-0002-6509-3506 (S. Bardin); 0000-0002-8980-7585 (A. Gotlieb); 0000-0003-2524-9462 (N. Lazaar); 0000-0002-8776-8770 (G. Menguy)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

While the PreCA and DCA oracle is efficient, query generation remains a bottleneck. Both methods rely solely on membership queries reporting execution success or failure, requiring numerous executions to be generated and processed which hinder scalability. Automated program analysis [7] can overcome this limitation by extracting deeper context. Developing more informative, analysis-driven queries and extending CA to support them is a crucial next step.

Contributions. In this paper, we explore how to enhance the scalability of PreCA by introducing explanatory queries and by proposing XAcq, the first gray-box, data-driven precondition inference algorithm that actively leverages these queries while providing strict formal guarantees for correctness and termination. More precisely, our contribution is threefold:

- We introduce a novel class of queries designed for oracles capable of providing structured feedback beyond mere classification. An *explanatory query* is a query that not only provides the classification of a given input (i.e., safe/unsafe/unknown) but also provides an explanation for its classification. In order to benefit from this extended answer, this explanation is expected to generalize the response and to allow us to reduce the number of further queries to ask. We formalize the assumptions required for an explanatory query to be informative and for an oracle to support such queries;
- We propose the XAcq algorithm and its implementation, extending PreCA to actively utilize explanatory queries. We show how XAcq can leverage the notion of *path predicate* in symbolic execution [8], a classical static analysis approach, to create informative explanations. Under reasonable hypotheses, we formalize and prove the termination and correctness properties for XAcq, establishing that this novel approach is sound and capable of efficiently processing structured feedback;
- Finally, we conducted a detailed experimental evaluation comparing XAcq with existing CA-based methods (PreCA and DCA) and PIE, a state-of-the-art baseline for precondition inference. Across two shared precondition inference benchmarks, our empirical results demonstrate that integrating explanatory queries substantially boosts the performance and scalability of CA-based methods. Furthermore, XAcq yields higher-quality preconditions than PIE in terms of soundness guarantees, while maintaining reasonable computational overhead.

The XAcq implementation, benchmarks, and results are publicly available on GitHub for reproducibility: <https://github.com/sdjebour/XACQ>.

2. Motivation

Consider the following C code example from [3] benchmark, which swaps elements of an array over a range of indices.

```
#define mswap(n, m) { int tmp = n; n = m; m = tmp; }

void mvswap(uint* ptr, int x1, int x2, int x3) {
    int y1 = (x1); int y2 = (x2); int y3 = (x3);
    while (y3 > 0) {mswap(ptr[y1], ptr[y2]); y1++; y2++; y3--;}
}
```

Precondition inference aims to infer on which inputs a function safely executes. Recently, constraint acquisition (CA) has been used to automatically infer preconditions. It turns the learning process into a dialogue: a *learner* repeatedly generates test inputs and submits them to an automated *oracle*, which executes the program and answers with a binary verdict: *safe* or *unsafe*. From this stream of verdicts, the learner progressively reconstructs the conditions under which the program runs safely. The precondition of `mvswap` equals $(x_3 > 0 \Rightarrow \text{valid}(ptr))$.

Core bottleneck. While CA-based approaches, like PreCA [3], are sound and fully automated, they suffer from a fundamental bottleneck: each oracle response carries only one bit of information. The learner therefore needs a large number of test inputs before it can distinguish the conditions that truly matter from those that do not. In practice, generating each new informative test input requires solving hard combinatorial problems, and this cost compounds rapidly as the program’s parameter space grows. Especially, PreCA does not converge on `mvswap` within one hour. The culprit is not the oracle’s speed — it can evaluate thousands of inputs per second — but the impoverished feedback: each response only narrows down the space of candidate preconditions by a tiny amount.

Exploiting program execution. The key observation driving this work is that a program execution carries *far more information* than a single safe/unsafe verdict. When a program runs on a concrete input, every branch taken, every loop entered or skipped, every pointer dereferenced — all of these decisions are governed by conditions on the input values. These conditions, taken together, describe not just whether *this specific input* is safe, but an entire *region* of the input space that would lead to the exact same execution. PreCA and DCA discard this structural knowledge entirely, using only the final verdict. *Our core idea is to exploit it, leading to our new XAcq proposal.* Especially, on `mvswap` XAcq infers the correct precondition in 2min 42s.

Example 2.1. Table 1 contrasts what PreCA and XAcq learn from these same two inputs over `mvswap`. For conciseness purposes, it considers a small set of constraints that could be used to express the target precondition: $B = \{x_i \leq 0, \text{valid}(\text{ptr}), \neg\text{valid}(\text{ptr}), x_i \leq 0 \vee \text{valid}(\text{ptr}), x_i \leq 0 \vee \neg\text{valid}(\text{ptr}) \mid i \in \{1, 2, 3\}\}$. The learner aims to identify which subset of B forms the actual precondition.

- *Impact on negative queries.* With PreCA, the unsafe verdict of the negative query e^- only tells the learner that *at least one* constraint from a certain set is violated, but not which one. By contrast, XAcq observes that the crash happened precisely because $x_3 > 0$ (the loop was entered) and `ptr` was invalid. This structural observation immediately identifies the constraint $x_3 \leq 0 \vee \text{valid}(\text{ptr})$ as part of the precondition in a single step, from a single execution;
- *Impact on positive queries.* The safe verdict of positive query e^+ tells a similar story. PreCA eliminates the constraints violated by this particular input. XAcq goes further: since the loop was *entirely skipped*, any constraint involving pointer validity is irrelevant to this execution, and can be ruled out wholesale and regardless of whether this specific input happens to violate it.

Information extracted from the execution therefore prune large portions of the hypothesis space in a single step, instead of removing candidates one by one.

3. Background

We review in this section the background on CA, precondition inference, and the program-analysis concepts exploited by the oracle.

Constraint Acquisition. CA [1] is a learning-based process that can be seen as an *interplay* between a CA-Learner and a source of information, referred to as an *oracle*. The learner and the oracle share a common *vocabulary* consisting of a finite set of variables X with values in a finite domain D . A *constraint* c over a subset of variables in X is a relation specifying the allowed value combinations, and a *constraint network* is a set of constraints. An example $e \in D^{|X|}$ satisfies a constraint c if its projection on the scope of c (i.e., the variables featured in c) belongs to its set of solutions, denoted by $\text{sol}(c)$. Furthermore, e is a solution to a constraint network C if it satisfies all constraints in C . Thus, a network represents a conjunction of constraints. The CA-Learner is provided with a constraint *language* Γ of bounded-arity relations, from which constraints can be instantiated over the vocabulary. The *constraint basis* B is the set of all constraints built from Γ on (X, D) . A constraint network C is representable if $C \subseteq B$. A *concept* is a Boolean function over $D^{|X|}$. A constraint network C represents a concept

Table 1

Information gained per query on mvswap: PreCA vs. XAcq.

	Query e^-	Query e^+
Input:	$(ptr = \text{NULL}, x_1 = x_2 = x_3 = 1)$	$(ptr = \text{NULL}, x_1 = x_2 = 1, x_3 = 0)$
Behavior:	the loop is entered, <code>ptr</code> is dereferenced $\rightarrow$ crash [<i>unsafe</i>]	the loop condition is false, the body is never reached $\rightarrow$ returns normally [<i>safe</i>]
PreCA:	The constraints violated by e^- are $\{x_i \leq 0, \text{valid}(ptr), x_i \leq 0 \vee \text{valid}(ptr) \mid i \in \{1, 2, 3\}\}$. At least one of them must belong to the precondition, but which ones remain unknown	The constraints violated by e^+ are $\{x_i \leq 0, \text{valid}(ptr), x_i \leq 0 \vee \text{valid}(ptr) \mid i \in \{1, 2\}\}$. All of them are eliminated, as none can belong to the precondition without incorrectly rejecting a safe input
XAcq:	Path predicate: $\pi(e^-) = (x_3 > 0) \wedge \neg \text{valid}(ptr)$ Since every input satisfying $\pi(e^-) = (x_3 > 0) \wedge \neg \text{valid}(ptr)$ crashes, any constraint whose negation is implied by $\pi(e^-)$ must belong to the precondition. XAcq thus infers $x_3 \leq 0 \vee \text{valid}(ptr)$.	Path predicate: $\pi(e^+) = (x_3 \leq 0)$ Beyond the constraints violated by e^+ , also $\{\neg \text{valid}(ptr), x_i \leq 0 \vee \neg \text{valid}(ptr) \mid i \in \{1, 2\}\}$ are eliminated, since any constraint violated by some input satisfying $\pi(e^+)$ cannot belong to the precondition

f if its solution set coincides with $f^{-1}(\text{true})$. Learning proceeds via *membership queries*, where the learner submits an example e to the oracle, which classifies it as positive (e^+) if e is a solution of the target concept, or negative (e^-) otherwise. For any example e , let $\kappa(e) = \{c \in B \mid e \not\models c\}$ denote the set of constraints in B rejecting e . Given a set E of labeled examples, a concept C *agrees* with E if it accepts all positive examples E^+ and rejects all negative ones E^- . The learning process has *converged* to $L \subseteq B$ if L agrees with E and all networks $L' \subseteq B$ agreeing with E are solution-equivalent to L . Given a basis B , the version space of E is the set of all constraint networks $C \subseteq B$ that are consistent with E .

Conacq [1] is a CA-Learner that relies on membership queries and represents the learner’s version space as a clausal theory. Each constraint $c \in B$ is associated with a Boolean variable $a(c)$ indicating whether c belongs to the learned network. Starting from an empty theory, Conacq iteratively generates informative queries, i.e., examples that reduce the version space regardless of the oracle’s answer. When no informative query remains, convergence is reached and the learned network is returned. This network is guaranteed to represent the target concept f_T if the basis is sufficiently expressive i.e., it contains at least one representation of the target concept.

DCA [5] is another CA-Learner, which also relies on membership queries. However, it is specifically designed to learn disjunctive target concepts: it can express the target concept as a conjunction of disjunctions of constraints from the basis. The core idea is to partition the input space into minimal regions using maximally satisfiable subsets (MSS), which are the largest subsets of constraints that can be constructed from the basis. Each MSS yields a candidate query submitted to the oracle. Upon a negative response, the learned formula is refined by conjoining it with the negation of the corresponding MSS, progressively eliminating regions of the input space classified as unsafe by the oracle.

Precondition Inference using CA. Recent works [3, 5, 4] adapt CA to the precondition inference problem. It aims to identify the conditions P on a function’s inputs under which its execution terminates successfully and satisfies a given postcondition Q [9]. Two frameworks have been proposed: PreCA and DCA. The former revises Conacq while the latter uses DCA for precondition inference. Both operate by replacing the human user with an automated oracle. In both PreCA and DCA, the learner’s queries are interpreted as concrete inputs to the function under analysis. The oracle executes the function on these inputs and classifies the outcome as *yes* if the execution terminates without crashing, as *no* if it crashes,

and as *ukn* if the execution does not terminate within a given timeout. At convergence, the learner returns a constraint network representing the inferred precondition. Existing work currently relies exclusively on membership queries, as automating richer query types has remained an open challenge.

Logical Theories [10]. A theory $\mathcal{T}$ is specified by a signature Σ , which defines its syntactic vocabulary: a set of variables X together with symbols for operations known as function and predicate symbols, each with fixed arities. From this signature, well-formed formulas (wffs) –syntactically correct formulas– of $\mathcal{T}$ are constructed. An interpretation $\mathcal{I}$ specifies the semantics by providing a domain to each variable and concrete meanings to the function and predicate symbols of Σ , allowing wffs to be evaluated as true or false. Given a wff w , we denote by $\text{sol}(w)$ the set of all solutions (models) that satisfy w . Constraints and constraint networks are viewed as well-formed formulas (wffs). Following standard CA notation, a set of wffs is identified with their conjunction. Conversely, a conjunction of wffs is treated as the corresponding set of formulas.

Symbolic Execution and Path Predicates. Program analysis provides multiple techniques for approximating program behavior during execution [7, 11, 12], among which symbolic execution [8] is one of the most prominent. It is particularly useful for testing program properties, for example determining whether an edge case may lead to a crash. Given a program, symbolic execution explores the different paths the program may take under varying inputs. Each path is represented by a path predicate π , a formula constructed by tracking the conditions encountered during execution [13] (e.g., branch conditions, loop guards, and error-handling checks). When a complete input assignment e is given, symbolic execution performs no exploration as only one path can be executed. Still, the *path predicate* can be constructed by gathering all the conditions taken during the execution of the program on e . Extracting path predicates is a common task in program analysis and can be done efficiently [14].

Example 3.1. Consider the `mvswap` function from Section 2, executed on the test input $e^- = (\text{ptr} = \text{NULL}, x_1 = x_2 = x_3 = 1)$. The path predicate π is initially $\top$. Lines 5–9 execute without branching, so π remains unchanged. At line 10, the condition $y_3 > 0$ holds, updating π to $\pi = (x_3 > 0)$. At line 11, `ptr` is dereferenced; since $\text{ptr} = \text{NULL}$, the execution crashes, yielding the final path predicate $\pi(e^-) = (x_3 > 0) \wedge \neg \text{valid}(\text{ptr})$.

4. Acquisition with Explanatory Queries

We introduce *explanatory queries*, a strict generalization of standard membership queries. Beyond returning the classification of an example e , an explanatory query provides a structured justification of this decision. We formalize this notion and show how it can be exploited to improve the efficiency of CA.

Assumptions. We consider a theory $\mathcal{T}$ over a finite set of variables X with domains $D^{|X|}$. The theory $\mathcal{T}$ defines the formal language used for interactions between the learner and the oracle: both manipulate well-formed formulas (wffs) of $\mathcal{T}$ over the same variables and domains. However, the learner and the oracle may rely on different subsets of function and predicate symbols within $\mathcal{T}$. We assume the availability of (i) a solver $\text{sol}v : \text{Wff} \rightarrow \{0, 1\}$ deciding satisfiability in $\mathcal{T}$, and (ii) a checker $\text{check} : D^{|X|} \times \text{Wff} \rightarrow \{0, 1\}$ deciding whether an assignment satisfies a formula. Since $\mathcal{T}$ is decidable, $\text{sol}v$ and check terminate. Finally, we assume that the oracle answers all queries correctly.

4.1. Inference from Explanatory Queries

Before formally defining explanatory queries, we note that in some contexts the oracle may be unable to provide a definite classification. For instance, in PreCA and precondition inference, executing the program on a given input e may not terminate or may be interrupted due to a timeout. In such cases, the oracle returns an *unknown* classification, denoted *ukn*, and no explanatory information is provided. Formally, the explanation I is considered empty for such queries.

Algorithm 1: XAcq

Input: A basis B **Output:** A clausal theory T encoding the target network

```
1 begin
2    $T \leftarrow \top$ ;
3   while true do
4      $e \leftarrow \text{QueryGen}(B, T)$ ;
5     if  $e = \text{nil}$  then
6       return  $T$ ;
7     else
8        $(\phi(e), I) \leftarrow \text{Ask}_X(e)$ ;
9       if  $\phi(e) = \text{yes}$  then  $T \leftarrow T \wedge \bigwedge_{c \in \mathcal{K}(I)} \neg a(c)$ ;
10      else  $T \leftarrow T \wedge \bigvee_{c \in \kappa(e)} a(c)$ ;
11      if  $\phi(e) = \text{no}$  then
12         $C_L \leftarrow \{c \in B \mid a(c) \in T\}$ ;
13        foreach  $c \in (B \setminus C_L)$ 
14          if  $C_L \wedge \neg c \models I$  then  $T \leftarrow T \wedge a(c) \wedge \neg a(\neg c)$ ;
```

Definition 4.1 (Explanatory Query). An *explanatory query* is a complete assignment e over X submitted to the oracle via an Ask_X question. The oracle returns a classification $\phi(e) \in \{\text{yes}, \text{no}, \text{ukn}\}$, and an explanation $I \in \mathcal{T}$ s.t.,:

$$\begin{cases} e \models I \text{ and } I \models C_T & \text{if } \phi(e) = \text{yes}, \\ e \models I \text{ and } I \models \neg C_T & \text{if } \phi(e) = \text{no}, \\ I = \emptyset & \text{if } \phi(e) = \text{ukn}. \end{cases}$$

We say that I *explains* $\phi(e)$ only when $\phi(e) \in \{\text{yes}, \text{no}\}$.

Positive Explanations. For positive queries, the explanation I^+ allows the learner to generalize the classification to a region of the input space. Formally, the set of constraints excluded by I^+ is $\mathcal{K}(I^+) = \{c \in B \mid I^+ \not\models c\}$, or equivalently $\mathcal{K}(I^+) = \bigcup_{e \in \text{sol}(I^+)} \kappa(e)$, where $\kappa(e) = \{c \in B \mid e \not\models c\}$. Hence, a single positive explanation can prune many candidate constraints at once.

Negative Explanations. Negative explanations are more challenging to exploit completely. We focus on a useful case where a negative explanation I^- can directly fix a constraint: for each unfixed $c \in B \setminus C_L$, if $\neg c \models I^-$, then $C_T \models c$, allowing us to fix c in the learned network. This test can be strengthened by considering the current learned network C_L : $C_L \wedge \neg c \models I^-$.

Unknown Queries. For queries e classified as *unknown* ($\phi(e) = \text{ukn}$), we do not receive any explanation I . In this case, we treat the query conservatively as a negative example for the purpose of version space updates.

4.2. Description of XAcq Algorithm

We give ourselves a basis B , which is a finite set of well-formed formulas from $\mathcal{T}$, i.e., $B \subset \text{Wff}$. As per classical CA, we assume that B is expressive enough to represent f_T i.e., there is $C_T \subset B$ s.t., $\text{sol}(C_T) = f_T^{-1}(\text{true})$. We then aim to infer a constraint network equivalent to C_T .

XAcq follows the main structure of PreCA: queries are generated using QueryGen, a procedure introduced in Conacq [1] and reused in PreCA [4, 3], which ensures that only informative queries are

produced. The algorithm iterates until no informative query remains, at which point the clausal theory T representing the target concept C_T is returned.

The key distinction between PreCA and XAcq lies in the Ask_X call and the way T is updated (starting at line 8). For each example e , the oracle is asked to provide a classification $\phi(e)$ and a corresponding explanation I .

- **Positive queries:** If the query is classified as *yes*, T is updated (Line 9) by rejecting any constraint that is violated by at least one solution of I . This allows aggregation over all solutions of I , generalizing the update.
- **Negative or unknown queries:** T is first updated as in PreCA (Line 10). If $\phi(e) = no$, for each unfixed constraint c , we check whether $C_L \wedge \neg c \models I$ (Line 13). When the implication holds, T is updated with $a(c)$ and $\neg a(\neg c)$, followed by unit propagation. If $\phi(e) = unk$, no explanation is available ($I = \emptyset$) and the query contributes as a conservative negative example.

By leveraging the explanatory information in I , XAcq generalizes each query update to a region of the input space, allowing the algorithm to converge faster while preserving soundness and completeness.

4.3. Theoretical Analysis

We show that XAcq terminates and returns a correct result under a sufficiently expressive basis.

Theorem 4.1 (Termination). XAcq terminates.

Sketch. Termination follows from three observations: (i) the underlying theory $\mathcal{T}$ is decidable; (ii) QueryGen terminates and only generates informative queries [1]; (iii) the basis B is finite, so there is a finite number of informative queries to process. $\square$

Theorem 4.2 (Correctness). Let B be a basis and $C_T \subseteq B$ the target constraint network. XAcq returns a solution-equivalent $C_L \equiv C_T$.

Sketch. Assume a sufficiently expressive basis B and an oracle that correctly classifies queries while providing sound explanations. A positive explanation I^+ allows us to aggregate the sets of constraints $\kappa(e)$ over all $e \in sol(I^+)$, effectively capturing the same information as processing each positive example individually. This ensures that the version space is updated consistently with the standard PreCA procedure. For negatives I^- , processing a negative explanation fixes a constraint c iff $C_L \wedge \neg c \models I^-$. Indeed, since $I^- \models \neg C_T$, we have $C_L \wedge \neg c \wedge C_T \models \perp$. As $C_L \subseteq C_T$, this simplifies to $\neg c \wedge C_T \models \perp$, i.e., $C_T \models c$, so c can be added to the learned network safely. Moreover, the negative example is processed as in PreCA, ensuring that the version space is updated consistently. For unknown queries ($\phi(e) = unk$), no explanation is available ($I = \emptyset$). These queries are treated conservatively as negative examples in the version space, without fixing any constraints, which preserves correctness by ensuring termination and guaranteeing no false positives. Combining these observations, each update—whether from a positive, negative, or unknown explanatory query—preserves soundness. Therefore, XAcq terminates with $C_L \equiv C_T$. $\square$

Discussion. XAcq only requires the theory $\mathcal{T}$ to be decidable; it does not require finite domains. This generalizes the classical presentation of CA, which often assumes finite domains. Indeed, neither Conacq, PreCA, nor XAcq depend on domain finiteness for correctness.

4.4. Precondition Inference with XAcq

As introduced in Section 4.1, an explanation generalizes an example. Path predicates [14] are natural candidates for such explanations. A path predicate is a conjunction of constraints characterizing the conditions on program inputs required to follow a specific execution path. Given a test input e , the path

predicate π can be constructed by monitoring execution and collecting branch conditions along the path. Explanations are thus built during query execution by storing encountered checks and conditional branches, and **come at no additional cost**: the explanation overhead is already accounted for by the execution time of the query on the function. By construction, the original test input e satisfies π , and all inputs satisfying π follow the same execution path. Hence, they receive the same classification, making *path predicates valid explanations*.

5. Experimental Evaluation

We address the following Research Questions (RQs):

- RQ1** *Is XAcq efficient for precondition inference?* We evaluate XAcq on the precondition inference task of CA, comparing its performance to PreCA, DCA and PIE on a dataset of real-world functions.
- RQ2** *How does basis size affect XAcq?* We experiment with three basis sizes and measure the impact on XAcq compared to PreCA, DCA and PIE.
- RQ3** *How do explanatory queries influence query generation?* We evaluate the effect of explanatory queries on the bottleneck of CA: query generation.
- RQ4** *How informative are path predicates as explanations?* We quantify how much path predicates help to determine whether constraints in the basis belong to the target concept or not.

5.1. Experimental Protocol

To answer these research questions, we implemented XAcq in Java¹(22.0.2) as an extension of the PreCA framework [4, 3]. For constraint reasoning, we use the Bitwuzla(0.4.0) [15] SMT solver over the Bitvector theory to check satisfiability, and the MINISAT SAT solver(2.3.5) [16] to generate informative queries.² The oracle is implemented with Binsec (0.9.1) [17, 18, 19], a binary analysis tool that runs the program and extracts path predicates, providing the explanations. Both Binsec and Bitwuzla operate over the quantifier-free bitvector theory with arrays, $\mathcal{T}_{\text{QF_ABV}}$, which thus serves as the shared theory between XAcq and the oracle.

Competitors and configurations. We evaluate XAcq against (i) PreCA and DCA (0.1.0), the state-of-the-art CA-based precondition inference framework; and (ii) the PIE [20] precondition inference framework from the programming language community. For each tool, we consider its default configuration.³ Still, all need a user given set of constraints. To evaluate the impact of the basis size (**RQ2**), we evaluate all competitors on three basis levels, ordered by increasing complexity: (i) **Level 1 (ideal)** includes only the necessary constraints; (ii) **Level 2 (default)** extends Level 1 by incorporating all constraints from [4], which include inequalities and basic pointer related constraints; (iii) **Level 3 (extended)** further extends Level 2 by adding the binary constraints $Y = -X$ and $Y = 2X$ for all integer variables.

Benchmark. Our evaluation relies on two complementary benchmark suites.

- **CoreBench** gathers 49 real C functions drawn from the active constraint acquisition literature. It includes all functions from [4, 2, 21], the DSA benchmark (<https://tinyurl.com/tvzzpvm>), the Frama-C WP test suite (<https://tinyurl.com/ycxdbjf3>), the Siemens suite [22], and examples from Gries’ *Science of Programming* [23].⁴

¹Each benchmark instance was executed in a fresh JVM instance

²Both solvers were used in their default configurations, ensuring fully deterministic behavior.

³Each tool is evaluated over a single run. While active methods use a solver with a fixed random seed, PIE (<https://github.com/SaswatPadhi/LoopInvGen>) performs passive learning and is evaluated using 1,000 random inputs.

⁴We discarded the BZ2_hbAssignCodes function because it requires inferring quantifiers, which is beyond the capabilities of PreCA, DCA, PIE, and XAcq.

- **LargeBench** consists of 7 functions from the TinyUSB project.⁵ These functions were selected because of their high number of parameters, which significantly enlarges the hypothesis space and makes constraint acquisition more challenging. This suite allows us to evaluate XAcq under large-basis and highly combinatorial conditions.

Table 2 reports statistics for both benchmark suites and the corresponding basis sizes. Functions in LargeBench involve more inputs than those in CoreBench, leading to larger bases, with up to 123,008 candidate constraints.

Table 2
Benchmark statistics

		CoreBench	LargeBench
Number of functions		49	7
Avg. number of variables (min - max)		2.4 (1-4)	10.3 (5-19)
Average number of atomic constr.	Lvl 1	5	10
	Lvl 2	15	153
	Lvl 3	16	220
Average basis size	Lvl 1	20	2,474
	Lvl 2	461	17,748
	Lvl 3	1,275	38,118

Table 3
Performance comparison across time budget (basis level 2)

		CoreBench				LargeBench			
		1s	5s	5 mins	1h	1s	5s	5 mins	1h
PIE	Succ.	5/49	18/49	18/49	18/49	0/7	4/7	4/7	4/7
	E	1000	1000	1000	1000	1000	1000	1000	1000
PreCA	Succ.	0/49	12/49	45/49	48/49	0/7	0/7	0/7	0/7
	E	-	8.1	13.1	22.5	-	-	-	-
DCA	Succ.	5/49	24/49	49/49	49/49	0/7	0/7	1/7	1/7
	E	3.4	4.6	32.1	32.1	-	-	540	540
XAcq	Succ.	1/49	17/49	48/49	49/49	0/7	0/7	4/7	7/7
	E	2	3.7	5.2	8.5	-	-	4	3.4

Table 4
Performance comparison across basis levels (TO=1h)

		CoreBench			LargeBench		
		Lvl 1	Lvl 2	Lvl 3	Lvl 1	Lvl 2	Lvl 3
PIE	Succ.	18/49	18/49	18/49	5/7	4/7	4/7
	E	1000	1000	1000	1000	1000	1000
PreCA	Succ.	49/49	48/49	45/49	5/7	0/7	0/7
	E	6.3	22.5	21.9	14.2	-	-
DCA	Succ.	49/49	49/49	48/49	7/7	1/7	1/7
	E	6.8	32.1	30.1	668.9	540	810
XAcq	Succ.	49/49	49/49	49/49	7/7	7/7	6/7
	E	3.6	8.5	8.5	2.4	3.4	3.5

Metrics. We evaluate XAcq using the following metrics. To assess efficiency relative to the state-of-the-art, we report: (i) *successes*, the number of runs that converge to the correct solution; (ii) the mean acquisition time (timeouts excluded); and (iii) the total number of queries generated for problems where the acquisition converged. To assess explanation quality (**RQ4**), we use two additional metrics:

$$\Delta^+(I, e) = \frac{|\mathcal{K}(I) \setminus \kappa(e)|}{|B \setminus C_T^*|} \quad \Delta^-(I) = \frac{|\{c \in B \mid \neg c \models I\}|}{|C_T^*|}$$

where $C_T^* = \{c \in B \mid C_T \models c\}$ is the closure of C_T , the constraint network representing target concept f_T . $\Delta^+(I, e)$ measures the ability of the explanation I to remove more constraints than the membership query e . On the other hand, $\Delta^-(I)$ measures the number of constraints that an explanation I can learn directly. In both cases, a value close to 1 indicates a highly informative explanation, while a

⁵<https://docs.tinyusb.org/en/latest/>

value close to 0 indicates that few constraints are fixed. **Setup.** We run PreCA, DCA, PIE and XAcq with varying per-function time budgets ⁶ (1s, 5s, 5min, 1h) and set the oracle timeout to 5s, which may produce unknown answers. Since PIE [20] is a passive black-box approach with no query-generation strategy, we evaluate it using 1000 randomly generated queries. All experiments were performed on a reserved Ubuntu 24.04 LTS server equipped with an Intel[®] Xeon[®] Silver 4112 CPU @ 2.60GHz and 62GB RAM. The postcondition is fixed to the implicit true postcondition for all runs.

5.2. Results

We now present results of XAcq over our benchmark.

RQ1. Table 3 compares XAcq, PreCA, DCA and PIE on both CoreBench and LargeBench, using the default basis across different per-function time budgets (1s, 5s, 5min, 1h). XAcq shows very rapid convergence thanks to explanatory queries. On CoreBench, with a 1s budget it infers 1 out of 49 preconditions, 17/49 after 5s, 48/49 after 5min, and the last precondition (the 49th) is inferred in about 13min. Hence, XAcq outperforms PreCA, which requires 1h to infer 48/49 preconditions from CoreBench, and does not converge on the last precondition within 1h. XAcq is also more efficient than PIE. Indeed, PIE is a passive learner, and since no effort is made to cover informative regions of the input space, the resulting preconditions are often incorrect. Hence, PIE only recovers 18/49 preconditions. However, on CoreBench, DCA is more competitive, being slightly faster than XAcq. This is expected as CoreBench deals with relatively simple functions. However, XAcq is the only method that handles LargeBench, which is much more challenging. XAcq can infer 4/7 preconditions in 5mins and 7/7 in 1h, while PreCA, DCA, and PIE reach a bottleneck, inferring in 1h at most 1 precondition for CA-based approaches and 4 for PIE. In terms of query efficiency and program executions, XAcq also triggers on average three times fewer executions (i.e., queries) than PreCA and four times fewer than DCA, thanks to the additional information provided by explanatory queries. These results show that explanatory queries enhance scalability, allowing XAcq to outpace PreCA and DCA on challenging benchmarks with drastically fewer queries.

Conclusion. XAcq alone handles all benchmarks efficiently. It consistently outperforms PreCA and PIE, scales better to challenging problems than DCA, and significantly reduces required queries.

RQ2. We evaluate the scalability of XAcq, PreCA, DCA and PIE with respect to the basis size under a 1h time budget per function. Results are summarized in Table 4. Overall, the results show that increasing the basis size affects the different approaches very differently. While PIE remains largely insensitive to richer bases, with only a slight increase in learning time and little variation in the number of correctly inferred preconditions, both PreCA and DCA suffer from scalability issues as the basis becomes more expressive. On CoreBench, all methods behave comparably under the Level 1 basis. However, for Levels 2 and 3, PreCA exhibits a sharp increase in convergence time, timing out on three functions at Level 3 that were previously solvable within the 1-hour budget. DCA also experiences degradation, with a noticeable increase in the number of required queries. In contrast, XAcq remains comparatively stable across all basis levels, consistently converging while requiring significantly fewer queries than DCA. The differences become even more pronounced on LargeBench. Under the Level 1 basis, XAcq is already about 9 $\times$ faster than PreCA. For richer bases, scalability issues become critical for both PreCA and DCA: PreCA fails to infer any precondition within the 1-hour budget at Levels 2 and 3, while DCA requires nearly twice as many queries (810 compared to 540) and becomes approximately 3.4 $\times$ slower. In contrast, XAcq successfully infers all target preconditions at Level 2 and 6 out of 7 at Level 3, demonstrating a substantially better ability to cope with the combinatorial growth induced by richer bases.

Conclusion. Explanatory queries allow XAcq to scale to large and expressive bases much more effectively than PreCA and DCA. This is particularly important in practice, as correctness guarantees in active

⁶The time budget accounts strictly for the acquisition process, which includes solver initialization and, for XAcq, path-predicate extraction. It excludes process startup (notably bias creation) and binary compilation.

constraint acquisition rely on expressive bases. While PreCA and DCA struggle to converge within reasonable time limits or require an excessive number of queries as the basis grows, XAcq remains able to efficiently exploit the additional expressiveness. PIE is not directly impacted by basis expressiveness due to its passive nature.

RQ3. We investigate how explanatory queries influence the query generation process. Table 5 reports the execution time of the QueryGen procedure. As expected, QueryGen is the main bottleneck: over the CoreBench dataset, PreCA spends on average 10.9 s in QueryGen, while on LargeBench this skyrockets to 1 078s. For DCA, while query generation is fast on CoreBench, it needs to generate a lot more queries than XAcq. Hence, DCA ultimately spends a total of 2,429.6s generating queries on LargeBench. This demonstrates that PreCA and DCA spend most of their time generating new queries, particularly when the basis is large. In contrast, XAcq significantly reduces QueryGen time thanks to explanatory queries: on average 0.7 s for CoreBench and 1.1 s for LargeBench. This shows that explanatory queries simplify the clausal theory by pruning irrelevant constraints and adding unit clauses, making QueryGen more efficient.

Conclusion. Explanatory queries substantially improve the efficiency of QueryGen by reducing both the number and the complexity of generated queries. By mitigating the current bottleneck in active CA, they enable faster convergence and scalable acquisition even on large and combinatorially challenging bases.

Table 5
Statistics on query generation time (TO=1h, basis level 2)

	CoreBench			LargeBench		
	PreCA	DCA	XAcq	PreCA	DCA	XAcq
Min	0.05s	0.01s	0.06s	0.1s	0.01s	0.06s
Max	95s	2s	4.7s	2,634s	271.6s	4.6s
Avg	10.9s	0.1s	0.7s	1,078.5s	2.2s	1.1s
Total	185s	6.6s	10s	3,599s	2429.6s	5.7s

Reported means cover all target functions, including runs that timed out considering only queries generated before exceeding the time budget. The *Total* row reports the average cumulative time spent on query generation across all functions of the specified benchmark.

Table 6
Explanation Quality (TO=1h, basis level 2)

		Δ^+	Δ^-
Corebench	Min	0%	0%
	Max	73.6%	100%
	Avg	27.8%	8.9%
	Std_dv	14.9%	21.7%
Tinybench	Min	43.7%	2%
	Max	74.4%	54.6%
	Avg	63.8%	32.7%
	Std_dv	9.1%	23.9%

RQ4. We evaluate our explanations on a 1-hour run with the Level 2 basis. Results report the percentage of constraints fixed relative to the basis B or C_T^* , depending on the metric, in the min-, max-, and average, and standard deviation across all runs. On CoreBench, positive explanations fix up to 73.6% of the basis (28% on average), while negative explanations can fully determine C_T^* in some cases (100%), with about 9% of C_T^* fixed on average. On LargeBench, which involves more complex functions, the quality of explanations is higher: positive explanations fix between 43.7% and 74.4% of the basis (63.8% on average), while negative explanations fix up to 54.6% of C_T^* (32.7% on average).

Conclusion. Explanatory queries consistently fix a large number of previously unset constraints, providing significantly more information than membership queries. This directly leads to fewer interactions and faster convergence.

6. Related Work

Constraint Acquisition. Constraint acquisition is a learning-based process that aims to model constraint problems by interacting with a non-expert user. Since the seminal paper [1] that focused on membership-query-based inference, many new types of queries have been proposed to speed up acquisition. These new types of queries include *Partial queries* [24], *Recommendation queries* [25] or

Generalization queries [26]. All these queries have been proposed in a context where the oracle is a human user. Hence, these cannot be answered automatically.

Closer to our work, [27] propose enriching each query with additional information, called *arguments*, intended for experts capable of explaining why examples should be accepted or rejected. Interestingly, our positive explanations (resp. negative ones) are related to their arg^S (resp. arg^-) arguments. Also, XAcq processes positive explanations similarly to arg^S , but adopts a different strategy for negative ones better suited to precondition inference. Moreover, arguments consist of sets of constraints from the basis B extended with all possible value assignments for each variable in its domain. Such restrictions are not conceivable in program verification due to the diversity of codes. Hence, our explanations I are not subject to this restriction and can be any well-formed formula within a shared theory, providing the oracle with greater flexibility.

Precondition Constraint Acquisition. Recently, new works proposed to use constraint acquisition to tackle program analysis tasks [4, 3]. This brings a new use case for constraint acquisition, removing its core limitation: the limited possible user interactions. However, precondition acquisition is still in its infancy. In this work we propose an extension of the PreCA framework, through *explanatory queries*. We have experimentally shown that these new queries enable faster acquisition, pruning a large portion of the search with only a few queries.

7. Conclusion

We introduced a new type of query, called *explanatory queries*, designed to maximize the information extracted from the oracle during the constraint acquisition process. Explanatory queries are particularly suitable in settings where the oracle is powerful and the goal is to accelerate learning, rather than minimizing human effort. We formalized how explanatory queries can be integrated into constraint acquisition, resulting in the XAcq algorithm, and demonstrated how this framework can be instantiated for code analysis to infer function preconditions. Extensive experiments on a dataset of complex, real-world software functions show that XAcq consistently outperforms the state-of-the-art while maintaining scalability. These results highlight that explanatory queries effectively address the main bottleneck of automated constraint acquisition: the limited informational content of traditional membership queries. By transforming program executions into structured explanations, XAcq significantly accelerates convergence, reduces the number of queries, and enables handling highly combinatorial bases that were previously intractable.

Declaration on Generative AI

During the preparation of this work, one of the the authors used Gemini in order to: Improve writing style, Grammar and spelling check. After using this tool/service, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

References

- [1] C. Bessiere, F. Koriche, N. Lazaar, B. O’Sullivan, Constraint acquisition, *Artif. Intell.* 244 (2017) 315–342. doi:10.1016/J.ARTINT.2015.08.001.
- [2] M. N. Seghir, D. Kroening, Counterexample-guided precondition inference, in: M. Felleisen, P. Gardner (Eds.), *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 451–471. URL: https://doi.org/10.1007/978-3-642-37036-6_25. doi:10.1007/978-3-642-37036-6_25.
- [3] G. Menguy, S. Bardin, N. Lazaar, A. Gotlieb, Automated program analysis: Revisiting precondition inference through constraint acquisition, in: L. D. Raedt (Ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, ijcai.org, 2022, pp. 1873–1879. doi:10.24963/IJCAI.2022/260.
- [4] G. Menguy, S. Bardin, N. Lazaar, A. Gotlieb, A query-based constraint acquisition approach for enhanced precision in program precondition inference, *J. Artif. Intell. Res.* 82 (2025) 901–936. doi:10.1613/JAIR.1.16206.
- [5] G. Menguy, S. Bardin, N. Lazaar, A. Gotlieb, Active disjunctive constraint acquisition, in: P. Marquis, T. C. Son, G. Kern-Isberner (Eds.), *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, 2023, pp. 512–520. doi:10.24963/KR.2023/50.
- [6] D. Angluin, Queries and concept learning, *Mach. Learn.* 2 (1987) 319–342. doi:10.1007/BF00116828.
- [7] F. Nielson, H. R. Nielson, C. Hankin, *Principles of program analysis*, Springer, 1999. doi:10.1007/978-3-662-03811-6.
- [8] C. Cadar, K. Sen, Symbolic execution for software testing: three decades later, *Commun. ACM* 56 (2013) 82–90. doi:10.1145/2408776.2408795.
- [9] C. A. R. Hoare, An axiomatic basis for computer programming, *Commun. ACM* 12 (1969) 576–580. doi:10.1145/363235.363259.
- [10] T. Scheurer, *Foundations of computing - system development with set theory and logic*, Addison-Wesley, 1994.
- [11] P. Cousot, *Principles of Abstract Interpretation*, MIT Press, 2021. URL: <https://mitpress.mit.edu/9780262044905/principles-of-abstract-interpretation/>.
- [12] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem (Eds.), *Handbook of Model Checking*, Springer, 2018. doi:10.1007/978-3-319-10575-8.
- [13] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, *ACM Comput. Surv.* 51 (2018) 50:1–50:39. doi:10.1145/3182657.
- [14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, EXE: automatically generating inputs of death, in: A. Juels, R. N. Wright, S. D. C. di Vimercati (Eds.), *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, ACM, 2006, pp. 322–335. doi:10.1145/1180405.1180445.
- [15] A. Niemetz, M. Preiner, Bitwuzla, in: C. Enea, A. Lal (Eds.), *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023. Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 3–17. doi:10.1007/978-3-031-37703-7_1.
- [16] N. Een, The minisat page, <http://minisat.se/>, 2006. Accessed: 2026-02-10.
- [17] A. Djoudi, S. Bardin, BINSEC: binary code analysis with low-level regions, in: C. Baier, C. Tinelli (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 212–217. URL: https://doi.org/10.1007/978-3-662-46681-0_17. doi:10.1007/978-3-662-46681-0_17.

- [18] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, J. Marion, BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis, in: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, IEEE Computer Society, 2016, pp. 653–656. doi:10.1109/SANER.2016.43.
- [19] B. Farinier, R. David, S. Bardin, M. Lemerre, Arrays made simpler: An efficient, scalable and thorough preprocessing, in: G. Barthe, G. Sutcliffe, M. Veanes (Eds.), LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018, volume 57 of *EPiC Series in Computing*, EasyChair, 2018, pp. 363–380. doi:10.29007/DC9B.
- [20] S. Padhi, R. Sharma, T. D. Millstein, Data-driven precondition inference with learned features, in: C. Krintz, E. D. Berger (Eds.), Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016, ACM, 2016, pp. 42–56. doi:10.1145/2908080.2908099.
- [21] S. Sankaranarayanan, S. Chaudhuri, F. Ivancic, A. Gupta, Dynamic inference of likely data preconditions over predicates by tree learning, in: B. G. Ryder, A. Zeller (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, ACM, 2008, pp. 295–306. URL: <https://doi.org/10.1145/1390630.1390666>. doi:10.1145/1390630.1390666.
- [22] M. Hutchins, H. Foster, T. Goradia, T. J. Ostrand, Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria, in: B. Fadini, L. J. Osterweil, A. van Lamsweerde (Eds.), Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994, IEEE Computer Society / ACM Press, 1994, pp. 191–200. URL: <http://portal.acm.org/citation.cfm?id=257734.257766>.
- [23] D. Gries, The science of programming, Springer Science & Business Media, 2012.
- [24] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C. Quimper, T. Walsh, Constraint acquisition via partial queries, in: F. Rossi (Ed.), IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013, IJCAI/AAAI, 2013, pp. 475–481. URL: <http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6659>.
- [25] A. Daoudi, Y. Mechqrane, C. Bessiere, N. Lazaar, E. Bouyakhf, Constraint acquisition with recommendation queries, in: S. Kambhampati (Ed.), Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016, IJCAI/AAAI Press, 2016, pp. 720–726.
- [26] C. Bessiere, R. Coletta, A. Daoudi, N. Lazaar, Y. Mechqrane, E. Bouyakhf, Boosting constraint acquisition via generalization queries, in: T. Schaub, G. Friedrich, B. O’Sullivan (Eds.), ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014), volume 263 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2014, pp. 99–104. doi:10.3233/978-1-61499-419-0-99.
- [27] K. M. Shchekotykhin, G. Friedrich, Argumentation based constraint acquisition, in: W. Wang, H. Kargupta, S. Ranka, P. S. Yu, X. Wu (Eds.), ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009, IEEE Computer Society, 2009, pp. 476–482. doi:10.1109/ICDM.2009.62.