Monadic Predicate Abstraction

Jiening Siow, Hanrui Zuo, Hanyun Jiang, Weiqi Wang, Tingting Lin, and Peisen Yao

The State Key Laboratory of Blockchain and Data Security, Zhejiang University
{jiening,zarin,jhanyun,wqwang1009,polariso,pyaoaa}@zju.edu.cn

Abstract. Many program analyses repeatedly invoke a common subroutine: checking whether a fixed formula remains satisfiable when conjoined with each predicate in a given set. We formalize this task as monadic predicate abstraction (MPA). We examine two baseline strategies and propose a new feedback-guided algorithm that leverages unsatisfiable cores to partition the predicate set via conjunctive under-approximation. We evaluate all approaches on benchmarks derived from symbolic abstraction, characterizing their performance across a range of structural and semantic dimensions. Our work establishes MPA as a fundamental abstraction primitive and provides both a theoretical framework and practical insights for its efficient implementation.

Keywords: Program analysis, constraint solving, program verification

1 Introduction

Predicate abstraction [1–4] is a foundational technique for constructing finite-state models of systems with unbounded or infinite state spaces. As an instance of abstract interpretation [5, 6], it approximates concrete program semantics using Boolean combinations over a finite set of logical predicates. These abstractions are amenable to automated reasoning via SAT and SMT solvers, and have been successfully applied to software [3], hardware [7], and protocol verification [8].

At the core of predicate abstraction lies the following operation: Given a formula φ over program variables in a background theory T, and a finite set of predicates $P = p_1, \ldots, p_n$, compute the strongest Boolean combination over P that soundly over-approximates φ. In practice, this operation is often realized by a battery of satisfiability checks that test a shared context against many related predicates, reusing solver reasoning wherever possible [2].

In this work, we focus on a variant but recurring problem that arises across a range of program analyses: determining the satisfiability of a fixed symbolic context φ conjoined with each predicate in a set. This arises, for instance, when verifying multiple properties under a shared path condition, or when checking a family of assertions along a symbolic execution path:

– *K-Induction*: In verifying multiple safety properties of transition systems, a common transition formula—typically an unrolling of the system semantics to a bounded depth—is conjoined with distinct safety predicates. Industrial systems often specify tens to hundreds of such properties, yielding a large number of satisfiability queries over a shared transition context.

- *Value-Flow Analysis* [9, 10]: The analysis encodes realizable value-flow paths as a symbolic formula capturing how values propagate through the program. The same encoding is queried against location-specific predicates to verify safety conditions at different program points (e.g., pointer dereference sites).
- *Symbolic Abstraction* [11, 12]: For template linear domains, the best abstract transformers can be computed by encoding program semantics as a fixed formula and solving for parameters that yield over-approximations. OMT solvers explore candidate parameters via iterative refinement, issuing satisfiability queries that vary the predicate while reusing the shared semantics.

We isolate and formalize this standard structure as the problem of *monadic predicate abstraction* (MPA):

Given a formula φ and a set of predicates $P = \{p_1, p_2, \ldots, p_n\}$, determine for each predicate p_i whether the conjunction $\varphi \land p_i$ is satisfiable.

This formulation can be viewed as a query-answering problem. A trivial abstraction answers unknown for all predicates, yielding a sound but maximally imprecise result. More precise abstractions distinguish between satisfiable and unsatisfiable cases, enabling clients to reason definitively about the feasibility of their solutions. Our goal is to compute the most precise abstraction possible, classifying each predicate as satisfiable, unsatisfiable, or unknown, while maintaining scalability.

There have been several related techniques to accelerate similar classes of problems—for example, expression caching in symbolic execution [13–15], unsat-core reuse in verification [8], and incremental solving inside SMT solvers. However, previous techniques are typically designed for broad reuse across heterogeneous queries, such as those arising from different paths, time frames, or analysis stages. For example, symbolic execution frameworks employ unified caching layers that store and reuse solver results across queries. While offering general-purpose acceleration, they do not exploit the structural regularities in MPA.

We argue that monadic predicate abstraction should be treated as a first-class operation, rather than as a byproduct of other analyses. This perspective enables new opportunities for efficiency and precision. First, reasoning over the entire batch exposes semantic relationships among predicates—implications, mutual exclusion, and dominance—that can be exploited to prune the search space or discharge queries without invoking the solver. Second, a dedicated bulk-checking interface enables amortization of solving effort across the batch. Shared theory lemmas, learned clauses, and branching heuristics can be reused across all queries, reducing redundant computation and improving throughput.

As a first step toward principled support for MPA, we investigate two existing algorithmic strategies: linear scan and disjunctive over-approximation(§ 3.1). Additionally, we introduce a new feedback-guided algorithm that leverages unsatisfiable core extraction and predicate partitioning (§ 3.2). All the methods are theory-agnostic but can incorporate theory-specific enhancements when applicable. We present a detailed analysis of the algorithms and discuss practical optimization heuristics (§ 3.3). To provide insight into their practical behavior, we evaluate them using queries from symbolic abstraction (§ 4). Our results show that each algorithm exhibits distinct performance trade-offs. We also discuss future directions for enhancing monadic predicate abstraction (§ 5). In summary, we make the following main contributions:

- A precise formalization of the monadic predicate abstraction problem.
- A new feedback-guided algorithm for monadic predicate abstraction, which exploits unsatisfiable cores to dynamically partition the predicate space.
- A systematic analysis and evaluation of three algorithmic approaches with respect to their theoretical properties and practical performance. The implementation is available at `https://tinyurl.com/3mwwr9xc`.

2 Motivation

Program analyses rely critically on abstraction to enable automated reasoning over infinite or intractable state spaces. A recurring computational pattern arises in this context: given a fixed symbolic formula φ—typically encoding a control-flow slice, symbolic execution path, or transition relation—and a finite set of predicates $P = \{p_1, \ldots, p_n\}$, determine the satisfiability of $\varphi_i \wedge p$ for each $p_i \in P$. Since each query conjoins φ with exactly one predicate from P, we refer to this pattern as *monadic predicate abstraction* (MPA). In this section, we illustrate its role in representative analysis tasks.

K-Induction for Multiple Property Verification. K-induction [16–19] extends classical mathematical induction to verify temporal properties of transition systems by considering execution traces of bounded length k. When verifying multiple safety properties simultaneously, both the base and inductive steps share a common formula representing the system's transition semantics.

Let a transition system be specified by the pair $(I(\boldsymbol{x}), T(\boldsymbol{x}, \boldsymbol{x}'))$ over state variables $\boldsymbol{x}$, and let $P_1, \ldots, P_n$ be the safety requirements to prove invariance. For a single property P_j the two k-induction proof obligations are

$$\textbf{Base:} \quad I(\boldsymbol{x}_0) \;\wedge\; \bigwedge_{i=0}^{k-2} T(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \;\models\; P_j(\boldsymbol{x}_{k-1}),$$

$$\textbf{Step:} \quad \big(P_j(\boldsymbol{x}_0) \wedge \cdots \wedge P_j(\boldsymbol{x}_{k-1})\big) \;\wedge\; \bigwedge_{i=0}^{k-1} T(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \;\models\; P_j(\boldsymbol{x}_k).$$

Crucially, the handling of different properties shares the same unrolling of the transition relation: $\Phi_k \;=\; I(\boldsymbol{x}_0) \;\wedge\; \bigwedge_{i=0}^{k-1} T(\boldsymbol{x}_i, \boldsymbol{x}_{i+1})$. Hence, each property P_j boils down to solving

$$\Phi_k \;\wedge\; \Big(\underbrace{P_j(\boldsymbol{x}_0) \wedge \cdots \wedge P_j(\boldsymbol{x}_{k-1})}_{\text{induction hypothesis}}\Big) \;\wedge\; \neg P_j(\boldsymbol{x}_k).$$

The verifier, therefore, invokes the SAT/SMT solver on the *same* core formula Phi_k with a *different* predicate attached each time, which exactly matches the monadic predicate abstraction pattern.

Path-Sensitive Sparse Value-flow Analysis. Value-flow analysis [20, 10, 9, 21, 22] tracks data dependencies through program execution paths to detect safety violations such as null-pointer dereferences, use-after-free, and taint issues. Even for a single property type, the analysis exhibits the monadic predicate abstraction pattern when checking multiple program locations.

Consider null-pointer dereference detection. The analysis constructs a *shared formula* φ capturing all realizable value-flow paths from potential null sources:

$$\varphi(\boldsymbol{x}) = \bigvee_{\pi \in \text{ValueFlowPaths}} \left(\bigwedge_{e \in \pi} \text{guard}(e) \right),$$

where guard(e) represents branch conditions along path edges, and realizability ensures context-sensitivity through call-return matching constraints.

Each dereference site ℓ_i induces a predicate $P_i(\boldsymbol{x}) \equiv (\text{ptr}_i \neq \texttt{null})$. Since dereferences are frequent, the number of such queries is large. The analysis thus repeatedly evaluates a shared formula φ against many predicates P_i.

Synthesizing Best Abstract Transformers. Given a Galois connections between the concrete domain $(\mathcal{C}, \leq_{\mathcal{C}})$ and abstract domains $(\mathcal{A}, \leq_{\mathcal{A}})$. with abstraction and concretization functions $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{C}$. The *best abstract transformer* for a concrete transformer $f : \mathcal{C} \to \mathcal{C}$ is defined as $f^{\alpha} = \alpha \circ f \circ \gamma$. However, this definition is non-constructive: it does not provide a method for computing a representation of f^{α}, nor for applying it algorithmically.

Symbolic abstraction [12, 23, 11, 24] addresses this by encoding program semantics as a formula φ and computing the least abstract element $a \in A$ such that $[\![\varphi]\!] \subseteq \gamma(a)$. In template linear domains, abstract elements are conjunctions of inequalities with fixed linear forms and variable parameters. The abstraction problem reduces to solving optimization modulo theories (OMT) queries: max $g_1, \ldots, g_n$ s.t. φ, where each objective g_i is maximized independently. OMT solvers typically employ iterative refinement. Each iteration evaluates candidate parameter vectors $\boldsymbol{c}$ and $\boldsymbol{c}'$ by issuing satisfiability queries of the form:

$$SAT\left(\varphi(\boldsymbol{x}, \boldsymbol{x}') \wedge p_{\boldsymbol{c}}(\boldsymbol{x}) \wedge \neg p_{\boldsymbol{c}'}(\boldsymbol{x}')\right),$$

where φ encodes the transition relation, and $p_{\boldsymbol{c}}$ and $p_{\boldsymbol{c}'}$ are predicates instantiated with candidate parameters. Each batch of queries fixes φ and varies the predicates, conforming to the MPA pattern. Multiple optimization objectives may be handled across batches, with some objectives converging early while others are resolved in later stages.

3 Algorithms

In the applications described above, a fixed formula φ is evaluated repeatedly under varying sets of predicates. Moreover, analyzing a single program typically

Algorithm 1: Predicate-by-predicate check

```
**Input:** An SMT formula $\varphi$ and a set of predicates $P = \{p_1, \ldots, p_n\}$
**Output:** Whether $\varphi \wedge p_i$ is satisfiable for each $p_i \in P$
1  foreach $p \in P$ do
2      if $\varphi \wedge p$ is satisfiable then
3          mark $p$ as satisfiable;
4      else
5          mark $p$ as unsatisfiable;
6  return marked_results;
```

Algorithm 2: Compact check via over-approximation

```
**Input:** An SMT formula $\varphi$ and a set of predicates $P = \{p_1, \ldots, p_n\}$
**Output:** Whether $\varphi \wedge p_i$ is satisfiable for each $p_i \in P$
1  while $P \neq \emptyset$ do
2      $\Psi \leftarrow \bigvee_{p \in P} p$; // Create disjunction of remaining predicates
3      if $\varphi \wedge \Psi$ is unsatisfiable then
           // All remaining predicates are unsatisfiable
4          mark every $p \in P$ as unsatisfiable;
5          return;
6      else
           // At least one predicate is satisfiable
7          $M \leftarrow$ a model of $\varphi \wedge \Psi$; // Get satisfying assignment
8          foreach $p_i \in P$ do
9              if $M \models p_i$ then
10                 mark $p_i$ as satisfiable;
11                 remove $p_i$ from $P$;// Remove from consideration
12 return marked_results;
```

involves solving a large number of such queries. This recurring structure admits significant optimization: specialized algorithms can exploit the shared computational patterns to improve performance. This section presents three algorithms for such problems.

3.1 Existing Approaches

Linear Scan Algorithm. A straightforward and the most commonly-used approach is to individually check whether each predicate $p \in P$ is satisfiable together with φ. Despite its simplicity, the number of solver calls grows proportionally with the number of predicates. In § 3.3, we will discuss the optimization of the algorithm, such as caching and incremental solving.

Disjunctive Over-Approximation Algorithm To reduce the number of solver queries, Algorithm 2 in [25] applies a disjunctive over-approximation strategy. The main idea is to group unresolved predicates into a single disjunctive formula and analyze them collectively, thereby amortizing the cost of solver invocations.

Given a formula φ and a set of predicates $P = \{p_1, \ldots, p_n\}$, the algorithm proceeds iteratively, refining P until all satisfiability results are determined.

1. *Predicate Aggregation (Line 2):* At each iteration, the algorithm constructs an over-approximated formula $\Psi = \bigvee_{p_i \in P} p$ by taking the disjunction of all predicates in the current set P.
2. *Satisfiability Check (Lines 3–11):* The algorithm then checks whether the conjunction $\varphi \wedge \Psi$ is satisfiable.
   - If $\varphi \wedge \Psi$ is *unsatisfiable*, this implies that no predicate in P can be satisfied alongside φ. In this case, each conjunction $\varphi \wedge p_i$ is marked as unsatisfiable, and this round of the algorithm terminates (Line 5). There is no need for explicit separation of SMT calls for each predicate in P.
   - If $\varphi \wedge \Psi$ is satisfiable, the SMT solver returns a model M. The algorithm iterates through each predicate $p_i \in P$ and evaluates whether $M \models p_i$. If a predicate p_i is satisfied by M, then $\varphi \wedge p_i$ is marked as satisfiable and removed from P (Line 11).
3. The process repeats, with P shrinking after each iteration. The algorithm terminates when all the predicates are marked.

Theorem 1. *Let k be the number of predicates in P for which $\varphi \wedge p_i$ is satisfiable. Algorithm 2 requires at most* $\min(k + 1, n)$ *SMT solver calls [25].*

Example 1. Consider $\varphi \equiv (x \geq 1 \wedge x \leq 5)$ and predicates $P = \{p_1 : x = 2, p_2 : x > 3, p_3 : x < 0, p_4 : x = 4, p_5 : x \leq 5\}$. The algorithm finishes using three iterations. First, with $P = \{p_1, p_2, p_3, p_4, p_5\}$, the algorithm constructs disjunction $\Psi = (x = 2) \vee (x > 3) \vee (x < 0) \vee (x = 4) \vee (x \leq 5)$. The solver returns model $M_1 = \{x = 2\}$ for $\varphi \wedge \Psi$, which satisfies p_1 and p_5. These predicates are marked as satisfiable. Second, with $P = \{p_2, p_3, p_4\}$, the new disjunction yields model $M_2 = \{x = 4\}$, which satisfies p_2 and p_4. These are marked satisfiable and removed. Finally, only $p_3 : x < 0$ remains. The formula $\varphi \wedge (x < 0)$ is unsatisfiable since φ requires $x \geq 1$. The algorithm marks p_3 as unsatisfiable and terminates. This algorithm completes the task in just 3 SMT solver calls—a 40% reduction compared to the naive linear scan approach.

3.2 Conjunctive Under-Approximation Algorithm

The over-approximation algorithm groups them disjunctively and attempts to eliminate multiple predicates in a single pass. In this subsection, we propose a conjunctive under-approximation strategy, which reasons in the opposite direction: rather than asking whether there exists at least one satisfiable predicate, it asks whether multiple predicates can be satisfied *together*. Intuitively, if a group of predicates is internally consistent, we can classify the entire group with a single solver call, thereby reducing the need for many redundant queries.

Consider predicates $P = x = 0, ; x \geq 0, ; x \geq -1$. All three can hold simultaneously with a background formula $\varphi \equiv (x \geq -1)$. Instead of testing each predicate separately (three solver calls), we may check $\varphi \wedge (x = 0 \wedge x \geq 0 \wedge x \geq -1)$ once. If this query is satisfiable, then all three predicates are satisfiable in a

Algorithm 3: Compact check via lazy under-approximation

```
Input: An SMT formula $\varphi$ and a set of predicates $P = \{p_1, \ldots, p_n\}$
Output: Whether $\varphi \wedge p_i$ is satisfiable for each $p_i \in P$
1  $W \leftarrow \{P\}$; // work-list that stores blocks still to be analysed
2  $S_{\text{fallback}} \leftarrow \emptyset$; // predicates delegated to fallback stage
3  while $W \neq \emptyset$ do
4      $s \leftarrow \text{pop}(W)$, $\Psi \leftarrow \bigwedge_{p \in s} p$;
5      if $\Psi$ is satisfiable then
           // no conflicts within the predicates, no need to split
6          while True do
7              $\Phi \leftarrow \varphi \wedge \bigwedge_{p \in s} p$;
8              if $\Phi$ is satisfiable then
                   // all remaining predicates are satisfiable
9                  mark every $p \in s$ as satisfiable;
10                 break
11             else
12                 $C \leftarrow \text{UnsatCore}(\Phi)$;
13                 $s \leftarrow s \setminus C$; // remove unsatisfiable part from this block
14                 $S_{\text{fallback}} \leftarrow S_{\text{fallback}} \cup C$; // add unsatisfiable part to
                       fallback list for later processing
15                 if $s = \emptyset$ then
16                     break
17     else
           // conflicts within the predicates, need to split
18         $C \leftarrow \text{UnsatCore}(\Psi)$, $R \leftarrow s \setminus C$;
19         if $|C| = 1$ then
20             $S_{\text{fallback}} \leftarrow S_{\text{fallback}} \cup C$;
21             if $R \neq \emptyset$ then
22                 $W \leftarrow W \cup \{R\}$;
23         else
24             $T \leftarrow \emptyset$;
25             foreach $c \in C$ do
26                 $T \leftarrow T \cup \{c\}$;
27             for $i$ in range($|R|$) do
                   // distribute residual part evenly into the $|C|$
                       subsets
28                 $T[i \bmod |C|] \leftarrow T[i \bmod |C|] \cup R[i]$;
29             $W \leftarrow W \cup T$;
30 if $S_{\text{fallback}} \neq \emptyset$ then
31     $\text{OA}(\varphi, S_{\text{fallback}})$ ;                // use Algorithm 2 or other method
32 return marked_results;
```

single step. More generally, the gain arises when many predicates are mutually consistent and can therefore be verified collectively.

Algorithm 3 gives the full pseudocode. It proceeds in two phases—(i) a *split phase*, which resolves internal inconsistencies among predicates, and (ii) a *batch*

Table 1: A step-by-step execution of the under-approximation algorithm

Iteration	Current Subset	Unsat Core	Result Subsets	Final Results
1	$\{p_0, p_1, p_2, p_3, p_4, p_5\}$	$\{p_0, p_5\}$	$\{\{p_0, p_1, p_3\}, \{p_5, p_2, p_4\}\}$	$\{?, ?, ?, ?, ?, ?\}$
2	$\{p_0, p_1, p_3\}$	$\emptyset$	$\{\{p_5, p_2, p_4\}\}$	$\{1, 1, ?, 1, ?, ?\}$
3	$\{p_5, p_2, p_4\}$	$\emptyset$	$\{\}$	$\{1, 1, 1, 1, 1, 1\}$

check phase, which tests entire groups conjunctively against the formula φ. Predicates that cannot be resolved within these steps are finally delegated to the over-approximation algorithm, which acts as a fallback.

1. **Split phase (Lines 17–29).** Starting with the entire predicate set P, the algorithm first checks whether the conjunction $\bigwedge P$ is satisfiable *without* the background formula φ. If this internal check is unsatisfiable, the solver returns a *minimal unsatisfiable core* $C \subseteq P$. Since predicates in C cannot all hold simultaneously, we partition the block:
   - If $|C| = 1$, that singleton is delegated to the later fallback stage.
   - Otherwise, each element of C becomes its own sub-block, and the remaining predicates $R := P \setminus C$ are redistributed among these sub-blocks for further refinement.

   This process repeats recursively until each block is either (i) jointly satisfiable or (ii) reduced to a singleton that cannot be further split.
2. **Batch check phase(Lines 5–16).** For every block s produced by the split phase, we issue a single query to check the satisfiability of $\varphi \wedge \bigwedge s$. If the result is satisfiable, the entire block is marked as satisfiable. If it is unsatisfiable, we again extract a minimal unsatisfiable core and delegate it to the later fallback stage.

Any residual predicates that survive splitting and batch checking are delegated to an over-approximation algorithm (Algorithm 2).

Example 2. Consider $x \leq 8$ and predicates $P = \{p_0 : x = 6,\ p_1 : x > 2,\ p_2 : x > 3,\ p_3 : x > 4,\ p_4 : x > 5,\ p_5 : x > 6\}$. Applying Algorithm 3 yields the steps shown in Table 1. All six predicates are therefore resolved in only three solver calls, whereas a linear scan would require six.

Theorem 2. *In the worst case, the split phase performs $n - 1$ solver calls, producing n singleton subsets. Each may require a separate batch check, and up to $\min(k + 1, n)$ additional calls may be needed in the fallback phase. The total number of solver calls is bounded by:*

$$(n-1)\ \textit{(split)} + n\ \textit{(batch check)} + \min(k+1, n)\ \textit{(fallback)}.$$

Theorem 3. *Let $n = |P|$ denote the number of predicates and k the number of satisfiable predicates in P. Then Algorithm 3 performs fewer than $3n$ SMT solver calls in total, and at most two in the best case.*

Proof. We analyze the number of SMT solver calls incurred by each phase of the algorithm. In the split phase, each invocation of Split eliminates at least one predicate from further consideration. Since there are n predicates in total, the number of split calls is at most $n-1$. In the batch phase, each block produced by the split phase is checked once in conjunction with φ. As the number of such blocks is at most n, the number of batch calls is bounded by n. Finally, the fallback procedure performs over-approximate checks on the remaining predicates. By Theorem 1, this requires at most $\min(k+1, n)$ calls. Hence, the total number of solver calls is strictly less than $(n-1) + n + \min(k+1, n) < 3n$.

If both $\bigwedge P$ and $\varphi \wedge \bigwedge P$ are satisfiable, the algorithm terminates after a single split and a single batch call, yielding a total of two SMT solver calls.

3.3 Algorithm Comparison

This subsection contrasts the three predicate–abstraction algorithms discussed so far. Let $n = |P|$ be the number of predicates and k $(0 \leq k \leq n)$ the number of *satisfiable* conjunctions $\varphi \wedge p_i$. All complexity figures count *solver invocations.*

Optimization Heuristics. The algorithms can benefit from standard SMT optimizations that are orthogonal and can be combined.

- *Incremental Solving (Inc)*: Uses push/pop to avoid reasserting φ for each query. Particularly effective for linear scan and under-approximation, where query deltas are small.
- *Model Reuse (Reuse)*: Reuses satisfying models to classify multiple predicates without additional solver calls. Especially useful in over-approximation, where a single model may validate many predicates.

Algorithmic Trade-offs. The three algorithms differ in abstraction strategy and performance characteristics. Table 2 summarizes these trade-offs.

- *Linear Scan (LS) (Algorithm 1)*: The algorithm issues *one* query per predicate. Its chief drawback is that the solver may not capitalise on similarities between predicates, even when we use incremental solving. When n is large, the algorithm may become prohibitively slow.
- *Over-approximation (OA) (Algorithm 2)*: This algorithm excels when only a few predicates are satisfiable, since unsatisfiable disjunctions can eliminate large sets at once. Its performance also depends on the "quality" of models returned: a model that satisfies many predicates allows rapid pruning.
- *Under-approximation (UA)(Algorithm 3)*: Our proposed algorithm is effective when many of the predicates can be satisfied simultaneously, enabling them to be classified in bulk. Otherwise, the overhead from splitting may reduce the advantage. In the worst case, it may rely on the fallback mechanism to handle most of the predicates.

While theoretical bounds provide useful guidance, empirical performance depends heavily on solver heuristics, predicate structure, and application-specific characteristics. Given the influence of domain-specific factors on algorithm performance, we provide a preliminary empirical evaluation to provide insights into their practical behavior in the next section.

Table 2: Comparison of monadic predicate abstraction algorithms

Aspect	Algorithm 1	Algorithm 2	Algorithm 3
Strategy	Check each $\varphi \wedge p_i$	Check $\varphi \wedge \bigvee P$	Check $\varphi \wedge \bigwedge P$
Complexity	$O(n)$	$O(\min(k+1, n))$	$O(2n-1+\min(k+1,n))$

Table 3: Summary of evaluated algorithm variants.

Variant Name	Base Algorithm	Optimizations
LS-Naive	Algorithm 1	None
LS-Inc	Algorithm 1	Incremental solving
LS-Reuse	Algorithm 1	Model reuse
LS-IncReuse	Algorithm 1	Incremental solving + Model reuse
OA	Algorithm 2	None
OA-Inc	Algorithm 2	Incremental solving
UA(OA-Inc)	Algorithm 3	None
UA-Inc(OA-Inc)	Algorithm 3	Incremental solving

4 Preliminary Evaluation

Our evaluation is designed to address the following research questions that illuminate different aspects of the monadic predicate abstraction problem:

- **RQ1**: How do the base algorithms compare in terms of the runtime and the number of solver calls (§ 4.1)?
- **RQ2**: To what extent do incremental solving and model reuse improve practical efficiency of the algorithms (§ 4.2)?
- **RQ3**: How do predicate properties, such as predicate counts and satisfiability ratio, affect algorithm performance (§ 4.3)?

Algorithm Variants. We benchmark several variants of the algorithm, each built on a core strategy and augmented with different optimization combinations. Table 3 summarizes the evaluated configurations. For the LS-based algorithms, we explore all combinations of incremental solving and model reuse, resulting in four variants. The OA strategy inherently supports model reuse, so only the naive and incremental variants are evaluated. For the UA-based approaches, to avoid redundancy and highlight representative performance, we select the best-performing variant (OA-Inc) as the fallback solver.

Benchmarks. To evaluate practical effectiveness, we use 3,987 MPA queries derived from symbolic abstraction tasks over bit-vector logic. Each query contains between 2 and 60 predicates. The benchmarks are drawn from ten real-world programs: perlbmk, glusterd, gap, eon, wrk, tmux, openssl, vortex, darknet, and transmission. The MPA queries were chosen to emphasize difficult cases that stress the abstraction algorithms.

Environment. All experiments are conducted on a server equipped with dual Intel Xeon Platinum 8176 CPUs (2.10 GHz, 28 cores each) and 500GB of RAM,

Table 4: Comparison of average runtime and solver invocation efficiency.

Algorithm	**Runtime (s)**	**Time ↓ (%)**	**#Calls**	**#Call ↓ (%)**
LS-Naive	1278.78	–	38.81	–
LS-Inc	235.80	81.56	38.81	0.00
LS-Reuse	272.52	78.69	11.16	71.25
LS-IncReuse	**142.99**	88.82	11.16	71.25
OA	275.43	78.46	8.68	77.63
OA-Inc	**118.12**	90.76	8.68	77.63
UA(OA-Inc)	**160.14**	87.48	13.69	64.72
UA-Inc(OA-Inc)	249.10	80.52	13.69	64.72

running Ubuntu 20.04 LTS. We use Z3 version 4.14.1 with a timeout of 30 seconds per query. Metrics collected include the total number of solver calls and the wall-clock execution time.

4.1 Comparison of the Base Algorithms (RQ1)

To understand the capabilities of different algorithmic strategies, we first evaluate the core algorithms—LS, OA, and UA—without any additional optimizations. Table 4 summarizes the comparative results.

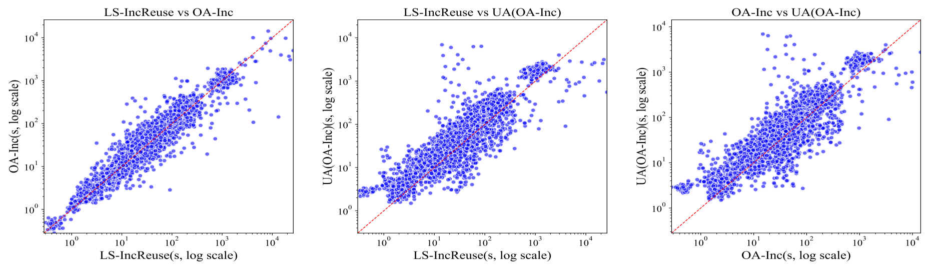
The LS-Naive configuration exhibits the slowest performance, with an average runtime of 1278.78 s. This degradation stems from its sequential, per-predicate evaluation process, which results in a large number of redundant solver invocations. In contrast, both OA and UA deliver substantial improvements in efficiency and solver usage. OA and UA substantially accelerate solving compared with LS-Naive, which trails far behind other algorithms and their variants. Specifically, OA reduces runtime to 78.46% and solver calls to 77.63%, whereas UA reduces them to 87.48% and 64.72%, respectively. Considering solver calls, our methods also have a decrease in solver calls compared to LS, with the least average solver calls being ≈ 8. The OA family benefits from its ability to eliminate infeasible regions early through over-approximation, while the UA strategy is particularly strong on highly satisfiable instances where feasible assignments are quickly identified.

4.2 Impact of Algorithm Optimizations (RQ2)

We next examine how incremental solving and model reuse affect algorithm efficiency. Figure 2, Figure 1, and Table 4 present detailed comparisons between optimized variants and their respective baselines.

Improvements for Each Algorihtm. The scatter plots in Figure 2 show that most data points lie below the diagonal, confirming that optimized variants nearly always outperform their corresponding baselines.

– *Incremental Solving*: Reusing solver states across related queries consistently accelerates convergence and reduces redundant preprocessing. The LS family benefits most, as the algorithm naturally fits the mode. However, the



(a) (b) (c)

Fig. 1: Comparison for best-performance variants of the MPA algorithms.

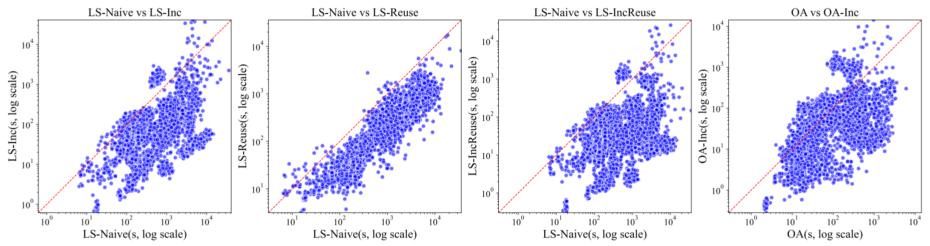
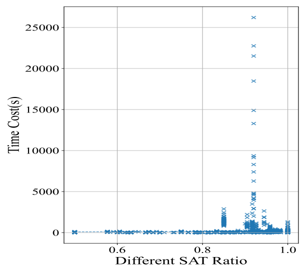


Fig. 2: Comparison between optimized and baseline algorithms.

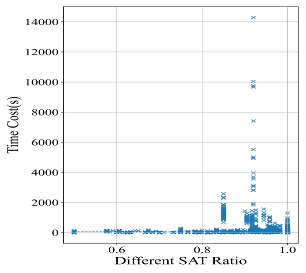
technique introduces slight overhead for UA due to the internal behavior of Z3's lazy SMT engine when operating in incremental mode.

- *Model Reuse*: Significantly decreases solver invocations by up to 71.25% for LS. This optimization proves most valuable in domains where satisfying assignments can be projected across multiple predicates, particularly in symbolic abstraction tasks.
- *Combined Optimizations*: When both optimizations are enabled, performance improves further. The LS-IncReuse variant achieves the lowest runtime among all LS configurations (142.99 s, and 88.8% reduction from LS-Naive). These gains demonstrate that incremental solving and model reuse complement each other effectively.

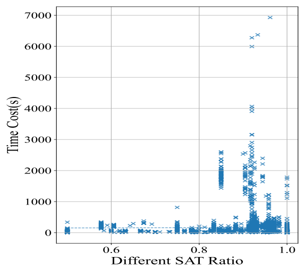
Comparison of Optimized Variants. Figure 1 compares the runtime performance of three optimized MPA variants. The results indicate that no single variant consistently outperforms the others; instead, performance varies across problem instances. In the left panel, LS-IncReuse and OA-Inc exhibit complementary strengths, with data points appearing on both sides of the diagonal—each method outperforms the other on different subsets. The middle panel, comparing LS-IncReuse and UA(OA-Inc), shows a similar distribution, with substantial variance across instances. The right panel, contrasting OA-Inc and UA(OA-Inc), reveals a broader spread, suggesting greater sensitivity to instance characteristics. Across all panels, the methods perform comparably on small instances (lower-left regions), while larger instances exhibit more pronounced and instance-specific differences.



(a) LS-IncReuse

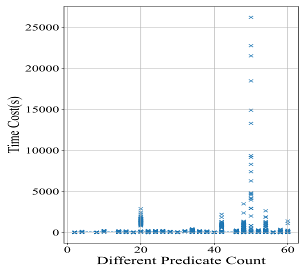


(b) OA-Inc

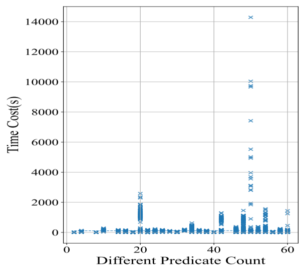


(c) UA(OA-Inc)

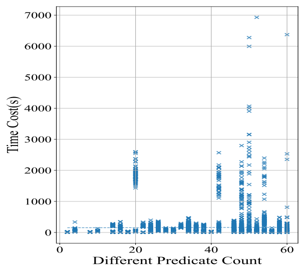
Fig. 3: Time cost on different SAT ratios.



(a) LS-IncReuse



(b) OA-Inc



(c) UA(OA-Inc)

Fig. 4: Time cost on different predicate counts.

4.3 Impact of Problem Characteristics (RQ3)

We evaluate how two key problem characteristics—satisfiability ratio and predicate count—affect solver performance. For each configuration, we report results for the best-performing variant of each algorithm.

Satisfiability Ratio. Figure 3 shows that solver runtime increases sharply as the satisfiability ratio approaches 1.0, indicating that near-satisfiable instances are significantly more difficult. This trend is consistent across all algorithms, but is most pronounced for non-incremental variants, which incur repeated solver invocations and redundant preprocessing.

The three algorithms exhibit distinct performance characteristics across the satisfiability spectrum. UA(OA-Inc) performs best on highly satisfiable instances (ratio > 0.95), quickly identifying solutions with few refinement steps. OA-Inc is most effective on unsatisfiable instances (ratio < 0.7), leveraging core-guided refinement to efficiently eliminate infeasible regions. LS-IncReuse delivers stable but moderate performance across all ratios, benefiting from incremental solving but lacking the aggressive pruning strategies of the other two approaches.

Predicate Count. Figure 4 reports solver performance as a function of predicate count. All algorithms exhibit sublinear scaling, though with varying sensitivity. LS-IncReuse degrades most rapidly, with runtime increasing significantly beyond 30 predicates due to linear scanning overhead. OA-Inc scales more gracefully, benefiting from its ability to refine multiple predicates per iteration. UA(OA-Inc) maintains near-constant performance up to 40 predicates. This efficiency arises from its use of unsatisfiable cores to eliminate large predicate sets early in the refinement process.

5 Discussions

Applicability of MPA. Although our focus is on formulas of the form $\varphi \wedge p_i$, the monadic predicate abstraction (MPA) framework generalizes to a broader class of program analysis tasks that share a common computational structure. In symbolic execution, MPA captures the scenario where a shared path condition φ is conjoined with distinct branch conditions p_i at a control-flow join; each conjunction $\varphi \wedge p_i$ must be checked for feasibility. Similarly, in invariant inference, modern guess-and-check techniques routinely generate dozens of candidates. MPA enables efficient evaluation by batching these candidates.

Limitations of the Study. The benchmark suite covers only a limited subset of MPA applications and may not reflect the full diversity of real-world usage scenarios. The results are contingent on the performance characteristics of the Z3 SMT solver and may not generalize to other solvers with different optimization strategies. In addition, applications that require significantly larger predicate sets may exhibit different scalability behavior. Our evaluation is confined to the theory of bit-vectors; extending the analysis to richer theories—such as arrays, algebraic datatypes, or strings—may expose further limitations or opportunities. Despite these constraints, the study offers a comparative assessment of the algorithms and their variants, providing practical guidance for their deployment in relevant contexts.

Future Work. The MPA problem presents several avenues for further research. On the theoretical side, tighter complexity bounds remain to be established, potentially by exploiting structural properties of the underlying theories or the predicate sets. From an algorithmic perspective, theory-specific optimizations—such as theory-aware lemma caching or semi-decision procedures—may improve performance. Adaptive algorithm selection, possibly via portfolio-based approaches, could further enhance robustness across diverse workloads. Finally, integrating MPA more deeply with domain-specific contexts (e.g., symbolic execution) may enable further optimization opportunities and scalability gains in practical applications.

6 Related Work

Predicate Abstraction. Introduced by Graf and Saïdi [26], predicate abstraction is a foundational technique in model checking and program verification [3, 2, 1]. Although early tools implemented predicate abstraction directly, modern verification frameworks typically employ refined variants, including lazy abstraction with interpolants [27] and implicit predicate abstraction [8] A related body of work investigates symbolic abstraction [11], which seeks the best over-approximation of a formula within a given abstract domain, such as finite-height domains [11], template linear domains [28, 24, 12], polyhedral domains [29], In contrast, the monadic predicate abstraction focuses specifically on determining predicate satisfiability in conjunction with a fixed formula, which can serve as a low-level primitive in the design and implementation of other algorithms.

Constraint Caching for SMT. SMT solvers are integral to modern verification and synthesis tools, enabling reasoning over theories such as bit-vectors, arrays, and linear arithmetic. They are widely used in various applications, such as symbolic execution [30, 31], software model checking [32–35], program synthesis [36, 37], automated repair [38, 39], and refinement type systems [40, 41]. To reduce solver overhead, prior work has explored caching mechanisms to avoid redundant queries [42, 14, 15]. KLEE [43] caches path conditions and counterexamples in symbolic execution, while Green [13] caches and simplifies queries over linear arithmetic. These systems are designed for general-purpose reuse across diverse queries, often arising from different paths or time frames. In contrast, our setting is more structured. We hypothesize that combining client-side optimizations, such as canonicalization, can lead to further performance improvements.

Consequence Finding. Consequence finding aims to compute logical entailments of a formula and is widely studied in deduction, such as computing prime implicants [44]. In the context of circuit verification, equality inference for Boolean functions [45] is a well-established technique; identifying equivalent sub-circuits can significantly reduce the complexity of equivalence checking. In satisfiability modulo theories (SMT), congruence closure is a standard method for inferring equalities from conjunctions involving uninterpreted functions [46]. In program analysis, consequence finding manifests in various forms, including quantifier elimination [47, 48, 47, 49], predicate abstraction, interpolation [50, 51, 27, 50, 52], and implied equalities [53]. The monadic predicative abstraction can be applied to identify consequences of a fixed formula φ with respect to a set of candidate predicates. This restricted form of consequence finding enables localized reasoning within a fixed context.

7 Conclusion

Monadic predicate abstraction (MPA) is a recurring computational pattern that arises in various applications, yet it has remained hidden in plain sight—buried in implementation details rather than celebrated as the fundamental primitive it truly is. This paper formalizes MPA, introduces a new algorithm, and empirically compares three algorithmic strategies, highlighting their trade-offs across benchmarks drawn from symbolic abstraction tasks. We advocate for the elevation of MPA to a first-class abstraction in the design space of program analyses, enabling systematic exploration of algorithmic design choices and facilitating structure-aware optimizations.

Bibliography

[1] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference (VMCAI'09)*.

[2] Shuvendu K Lahiri, Robert Nieuwenhuis, and Albert Oliveras. Smt techniques for fast predicate abstraction. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 424–437, Berlin, Heidelberg, 2006. Springer-Verlag. https://doi.org/10.1007/11817963_39.

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM. https://doi.org/10.1145/378795.378846.

[4] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*.

[5] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM. https://doi.org/10.1145/567752.567778.

[6] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'78)*.

[7] Hongce Zhang, Aarti Gupta, and Sharad Malik. Syntax-guided synthesis for lemma generation in hardware model checking. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'21)*.

[8] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*.

[9] Wensheng Tang, Dejun Dong, Shijie Li, Chengpeng Wang, Peisen Yao, Jinguo Zhou, and Charles Chuan Zhang. Octopus: Scaling value-flow analysis via parallel collection of realizable path conditions. *ACM Trans. Softw. Eng. Methodol.*, 33 (3):66:1–66:33, 2024. https://doi.org/10.1145/3632743.

[10] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 693–706, New York, NY, USA, 2018. ACM. https://doi.org/10.1145/3192366.3192418.

[11] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, volume 2937, pages 252–266. Springer, 2004.

[12] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. Program analysis via efficient symbolic abstraction. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021. https://doi.org/10.1145/3485495.

[13] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 58:1–58:11. ACM, 2012.

[14] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. Greentrie: Efficient constraint solving for software symbolic execution. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 311–320. IEEE, 2015.

[15] Antonio Aquino, Marc Denecker, and Broes De Cat. Utopia: Smt-based formal analysis made easy. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, pages 167–170. IEEE Press, 2019.

[16] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction. In *Static Analysis: 22nd International Symposium (SAS'15)*.

[17] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In *Computer Aided Verification: 27th International Conference (CAV'15)*.

[18] Dejan Jovanović and Bruno Dutertre. Property-directed k-induction. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 85–92. IEEE, 2016.

[19] Omar M Alhawi, Herbert Rocha, Mikhail R Gadelha, Lucas C Cordeiro, and Eddie Batista. Verification and refutation of c programs based on k-induction and invariant inference. *International Journal on Software Tools for Technology Transfer (STTT'21)*.

[20] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 480–491. ACM, 2007. https://doi.org/10.1145/1250734.1250789. URL https://doi.org/10.1145/1250734.1250789.

[21] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse analysis without path conditions. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 930–943. ACM, 2021. https://doi.org/10.1145/3453483.3454086.

[22] Yuandao Cai, Peisen Yao, and Charles Zhang. Canary: practical static detection of inter-thread value-flow bugs. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1126–1140. ACM, 2021. https://doi.org/10.1145/3453483.3454099.

[23] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with smt solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*.

[24] Jörg Brauer and Andy King. Automatic abstraction for intervals using boolean formulae. In *SAS*, volume 6337, pages 167–183. Springer, 2010.

[25] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. Program analysis via efficient symbolic abstraction. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021.

[26] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

[27] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006. https://doi.org/10.1007/11817963_14.

[28] David Monniaux. Automatic modular abstractions for linear constraints. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 140–151, New York, NY, USA, 2009. ACM. https://doi.org/10.1145/1480881.1480899.

[29] Aditya Thakur and Thomas Reps. A method for symbolic computation of abstract operations. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, page 174–192, Berlin, Heidelberg, 2012. Springer-Verlag. https://doi.org/10.1007/978-3-642-31424-7_17.

[30] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. https://doi.org/10.1145/1081706.1081750.

[31] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. pages 322–335, 2006. https://doi.org/10.1145/1180405.1180445.

[32] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *International Conference on Computer Aided Verification*, pages 36–52. Springer, 2013.

[33] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[34] Jiahong Jiang, Liqian Chen, Xueguang Wu, and Ji Wang. Block-wise abstract interpretation by combining abstract domains with smt. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'17, pages 310–329. Springer, 2017.

[35] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti EJ Hyvärinen, and Natasha Sharygina. Hifrog: Smt-based function summarization for software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 207–213. Springer, 2017.

[36] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[37] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 643–659. USENIX Association, 2017.

[38] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM. https://doi.org/10.1145/2884781.2884807.

[39] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[40] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 269–282. ACM, 2014. https://doi.org/10.1145/2628136.2628161.

[41] A. Champion, T. Chiba, N. Kobayashi, and R. Sato. Ice-based refinement type discovery for higher-order functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10805 of *Lecture Notes in Computer Science*, pages 365–383. Springer, 2018. https://doi.org/10.1007/978-3-319-89960-2_20.

[42] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In *Haifa Verification Conference*, pages 284–299. Springer, 2014.

[43] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[44] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 46–52. IEEE, 2013. URL https://ieeexplore.ieee.org/document/6679390/.

[45] C. Leonard Berman and Louise Trevillyan. Functional comparison of logic designs for VLSI circuits. In *1989 IEEE International Conference on Computer-Aided Design, ICCAD 1989, Santa Clara, CA, USA, November 5-9, 1989. Digest of Technical Papers*, pages 456–459. IEEE Computer Society, 1989. https://doi.org/10.1109/ICCAD.1989.76990. URL https://doi.org/10.1109/ICCAD.1989.76990.

[46] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification.* Springer Science and Business Media, 2007.

[47] Peter Backeman, Philipp Rummer, and Aleksandar Zeljic. Bit-vector interpolation and quantifier elimination by lazy reduction. In *2018 Formal Methods in Computer Aided Design (FMCAD'18)*, .

[48] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[49] Ajith K John and Supratik Chakraborty. A quantifier elimination algorithm for linear modular equations and disequations. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 486–503, Berlin, Heidelberg, 2011. Springer-Verlag.

[50] Rahul Sharma, Aditya V Nori, and Alex Aiken. Interpolants as classifiers. In *International Conference on Computer Aided Verification (CAV'12)*.

[51] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In *Proc. 7th Int. Haifa Verification Conf. on Hardware and Software: Verification and Testing*, HVC'11, pages 160–175, 2012. https://doi.org/10.1007/978-3-642-34188-5_15.

[52] Peter Backeman, Philipp Rümmer, and Aleksandar Zeljić. Interpolating bit-vector formulas using uninterpreted predicates and presburger arithmetic. *Formal methods in system design (FMSD'21)*, .

[53] Josh Berdine and Nikolaj Bjørner. Computing all implied equalities via SMT-based partition refinement. Technical Report MSR-TR-2014-57, Microsoft Research, April 2014.