Incremental Predicate Analysis for Regression Verification

QIANSHAN YU, School of Software, Tsinghua University, Key Laboratory for Information System Security, MoE, and Beijing National Research Center for Information Science and Technology, China
FEI HE, School of Software, Tsinghua University, Key Laboratory for Information System Security, MoE, and Beijing National Research Center for Information Science and Technology, China
BOW-YAW WANG, Academia Sinica, Taiwan

Software products are evolving during their life cycles. Ideally, every revision need be formally verified to ensure software quality. Yet repeated formal verification requires significant computing resources. Verifying each and every revision can be very challenging. It is desirable to ameliorate regression verification for practical purposes. In this paper, we regard predicate analysis as a process of assertion annotation. Assertion annotations can be used as a certificate for the verification results. It is thus a waste of resources to throw them away after each verification. We propose to reuse the previously-yielded assertion annotation in regression verification. A light-weight impact-analysis technique is proposed to analyze the reusability of assertions. A novel assertion strengthening technique is furthermore developed to improve reusability of annotation. With these techniques, we present an incremental predicate analysis technique for regression verification. Correctness of our incremental technique is formally proved. We performed comprehensive experiments on revisions of Linux kernel device drivers. Our technique outperforms the state-of-the-art program verification tool CPAchecker by getting 2.8x speedup in total time and solving additional 393 tasks.

CCS Concepts: • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: Software verification, predicate analysis, change impact analysis, incremental verification

ACM Reference Format:
Qianshan Yu, Fei He, and Bow-Yaw Wang. 2020. Incremental Predicate Analysis for Regression Verification . *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2020), 25 pages.

1 INTRODUCTION

Because of performance improvement, new features, or even bug fixes, software is constantly evolving in its life cycles [He et al. 2016; Lehman and Belady 1985; Lehman et al. 1998, 1997; Turski 1996]. Each software revision is enriched with new features or better performance; it may also introduce unexpected behaviors. In order to ensure its quality, software ideally need be formally verified for each revision [D'Silva et al. 2008]. Yet software verification requires significant computing resources. Regression verification is, in the worst case (thinking of verifying from scratch), as complicated as the formal verification. Andronick et al. [2012] reported that the effort of verifying each program revision is proportional (a factor of approximately 3-5) to the effort of changing the program. Thus, astronomical resources will be needed to verify every software revision during its life cycle.

It is, however, not hard to tell the difference of conventional software verification from those in regression. Each revised software is based on a previous version. If the old version has been verified, it does not appear to be economical to verify the revision from scratch again. Intuitively,

Authors' addresses: Qianshan Yu, School of Software, Tsinghua University , Key Laboratory for Information System Security, MoE , Beijing National Research Center for Information Science and Technology, Beijing, China, yqs17@mails.tsinghua.edu.cn; Fei He, School of Software, Tsinghua University , Key Laboratory for Information System Security, MoE , Beijing National Research Center for Information Science and Technology, Beijing, China, hefei@tsinghua.edu.cn; Bow-Yaw Wang, Academia Sinica, Taiwan, bywang@iis.sinica.edu.tw.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

verification information about the old version can be reused if it is unchanged in the verification of the new version. To exploit the resources invested in verifying prior versions of software, techniques have been developed to utilize intermediate verification results [Beyer et al. 2013; Henzinger et al. 2003a; Sery et al. 2012; Yang et al. 2009]. Results of expensive constraint solving have also been stored for regression verification [Aquino et al. 2015; Jia et al. 2015; Visser et al. 2012]. Although these techniques reduce the efforts of regression verification effectively, they are specialized for underlying verification algorithms. It is not entirely clear how the information of prior verification can be reused more generally.

It is perhaps useful to view regression verification in the abstract interpretation framework [Cousot and Cousot 1977]. In abstract interpretation, program behaviors are over-approximated by abstract states in abstract domains. More concretely, an abstract state is computed at a program location of the control flow graph to represent all possible program behaviors at the location. To verify safety properties, it suffices to perform reachability analysis in proper abstract domains to demonstrate that bad abstract states cannot be reached. Note that all program locations are annotated with abstract states after verification.

During regression verification, the control flow graph of the revised program changes from the original program. Abstract states computed for the old version may no longer be valid for the revision. If the revision does not differ from the original significantly, not all abstract states need be recomputed. It suffices to analyze the differences, identify invalid abstract states, and re-compute such abstract states for regression verification. In other words, the annotated control flow graph for the revision can be obtained by revising the annotated control flow graph for the original. One need not perform reachability analysis from scratch in regression verification.

The view of regression verification in abstract interpretation is perhaps intuitive, but details can vary from different abstract domains. In this work, we develop an efficient regression verification technique for predicate analysis as an example. Predicate analysis is a well-established software verification technique [Ball et al. 2001; Das et al. 1999; Graf and Saidi 1997]. It has been combined with other techniques such as counterexample-guided abstraction refinement (CEGAR) [Clarke et al. 2000], lazy abstraction [Henzinger et al. 2002] and interpolation [Christ et al. 2012; McMillan 2006] in industrial software verification tools SLAM [Ball and Rajamani 2001], BLAST [Henzinger et al. 2003b] and CPAchecker [Beyer and Keremoglu 2011] among others.

In predicate analysis, abstract states are represented by assertions over a given set of predicates. After verifying a program, every program location in its control flow graph is annotated with assertions over the predicates. To demonstrate our ideas, we develop techniques to analyze program text differences, identify invalid assertions syntactically, and recompute invalid assertions in the abstract predicate domain for revised programs. Instead of computing assertions at each program location from scratch, our technique simply keeps existing assertions and computes new ones when necessary in regression verification. Observe that our new technique only requires difference analysis and employs the classical reachability analysis algorithm. It differs from standard abstract interpretation algorithms very slightly and is hence compatible with other techniques almost for free.

For evaluation, we implemented our regression verification technique on top of CPAchecker and performed comprehensive experiments on real-world revisions of Linux kernel device drivers. Compared to existing techniques, our new technique solves 393 more tasks and has the speedup of 2.8 among solved ones. To summarize, we have the following contributions:

- We reformulate predicate analysis as assertion annotation and use assertion annotations as intermediate results for regression verification;

- We propose a light-weight impact analysis and strengthening techniques for reusing assertions in predicate analysis;
- We propose an incremental predicate analysis technique for evolving programs and prove its soundness;
- We implement our algorithms on top of CPAchecker and report extensive experiments on revisions of Linux kernel drivers.

The remainder of this paper is organized as follows. Section 2 introduces necessary preliminaries. Section 3 explains the relation between assertion annotation and predicate analysis. Section 4 presents our incremental predicate analysis technique. Section 5 reports evaluation results on our approach. Section 6 discusses related work and Section 7 concludes this paper.

2 PRELIMINARIES

2.1 Program and Control-Flow Automata

We will represent programs by *control-flow automata* (CFA) [Beyer et al. 2007; Henzinger et al. 2002]. Given a program P with a set of *operations* Ops, the CFA of P is a triple (L, l_0, G) where L is the set of *locations*, $l_0 \in L$ is the *initial location*, and $G \subseteq L \times Ops \times L$ is the set of *edges*. Moreover, $l_{err} \in L$ designates the *error* location. A *path* on P is a sequence $l_0 \xrightarrow{op_0} l_1 \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} l_n$ such that $(l_i, op_i, l_{i+1}) \in G$ for every $0 \leq i \leq n-1$.

Let V be the set of program variables. A *concrete state* consists of a program location l and a valuation c of V. The valuation c is also called a *concrete data state*. A *computation* from an initial state (l_0, c_0) of the program is an alternating sequence of concrete states and operations

$$(l_0, c_0) \xrightarrow{op_0} (l_1, c_1) \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} (l_n, c_n)$$

such that $l_0 \xrightarrow{op_0} l_1 \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} l_n$ is a path on P and c_{i+1} is a concrete data state obtained by executing op_i from c_i for every $0 \leq i \leq n-1$. An *error* computation is a computation ending at a concrete state (l_{err}, c_{err}) for some concrete data state c_{err}. Given a program P, the *program verification* problem is to check whether there is an error computation from an arbitrary initial state. If there is no error computation, the program P is *safe*.

2.2 Predicate Analysis

Predicate analysis [Das et al. 1999; Graf and Saidi 1997] is a practical approach for program verification. It uses a set of predicates to abstract concrete data states and performs exhaustive search in the abstract state space. Predicate abstraction is a conservative abstraction. If the error location is unreachable in the abstract model, it is also unreachable in the concrete model. One can safely conclude that the program is safe when the error location cannot be reached in predicate analysis.

A *predicate* is a quantifier-free first-order formula over the program variables V. We use $\top$ to denote *true* and $\bot$ to denote *false*. Let φ be a predicate. A pair (l, φ) represents a set of concrete states: $\{(l, c) \mid c \models \varphi\}$. One can lift computation over sets of concrete states as follows. For any predicate φ, let $SP_{op}(\varphi)$ represent the set of concrete data states reached from a concrete data state satisfying φ after executing op. For instance, suppose $(l_0, op, l_1) \in G$, c_0 an arbitrary concrete data state and $c_1 \in SP_{op}(\top)$. Then $c_0 \xrightarrow{op} c_1$ is a computation along the path $l_0 \xrightarrow{op} l_1$ for $c_0 \models \top$. If $SP_{op}(\top)$ could be represented by a predicate, one would be able to solve the program verification problem by computing $SP_{op_i}(\bullet)$ for every op_i iteratively.

However, $SP_{op_i}(\bullet)$ is not a predicate and can be hard to compute in general. To improve efficiency, predicate analysis only allows predicates derived from a fixed predicate precision. A *(predicate) precision* λ is a set of *atomic* predicates [Beyer et al. 2013]. A *cube* (over λ) is a conjunction of atomic

predicates in λ. A formula is in *disjunctive normal form* (over λ) if it is a disjunction of cubes over λ. An *abstract state* is a pair (l, ϕ) where l is a location and ϕ is a cube of predicates in λ. ϕ is called an *abstract data state*.

The *predicate abstraction* $(\varphi)^\lambda$ of a formula φ is the strongest conjunction of predicates in λ that are entailed by φ

$$(\varphi)^\lambda := \bigwedge \{p \in \lambda \mid \varphi \Rightarrow p\}.$$

Let $(l, op, l') \in G$ and ϕ be an abstract data state. We hence define the *abstract successor* of (l, ϕ) with respect to op to be $(l', (SP_{op}(\phi))^\lambda)$. Since $(SP_{op}(\phi))^\lambda$ is itself a predicate, one can compute abstract successors iteratively. Note that $(SP_{op}(\phi))^\lambda$ is but an over-approximation to $SP_{op}(\phi)$.

A typical predicate analysis algorithm is depicted in Algorithm 1. The algorithm keeps updating two sets of abstract states, that is, a set *reached* of reachable abstract states found so far and a set *waitlist* of abstract states that are going to be processed. Initially, only $(l_0, \top)$ is in the *waitlist* (step 1). In step 2, the algorithm explores all reachable abstract states in a breadth-first fashion. At each iteration, the algorithm fetches an abstract state from *waitlist* (line 3, in a FIFO order), say (l, ϕ), and then computes all its successors. A successor (l', ϕ') is skipped if it is covered by the set *reached*, that is, there exists an abstract state $(l'', \phi'') \in reached$ such that $l' = l''$ and $\phi' \Rightarrow \phi''$. Otherwise, it is added to *reached* and *waitlist* (lines 6–7). The above process repeats until *waitlist* is empty. If the error location l_{err} is not in the set *reached* returned by the algorithm, we conclude that the program is safe.

Algorithm 1: PA(P, λ)

Input: A program P, an abstract precision λ
Output: A set *reached* of reachable abstract states

```
1  waitlist ← {(l0, ⊤)}, reached ← {(l0, ⊤)} ;          /* initialization */
2  while waitlist ≠ ∅ do                                /* compute the reached set */
3      pop (l, φ) from waitlist;
4      foreach l′ with (l, op, l′) ∈ G do
5          φ′ ← (SP_op(φ))^λ ;                          /* abstract successor computation */
6          if (l′, φ′) is not covered by reached then
7              waitlist ← waitlist ∪ {(l′, φ′)};
8              reached ← reached ∪ {(l′, φ′)};
9          end
10     end
11 end
12 return reached;
```

2.3 Counterexample-Guided Abstraction Refinement

In predicate analysis, the precision λ defines the level of abstraction. The precision must be at a proper level. A too-coarse precision may induce spurious counterexamples; a too-fine precision, however, may lead to state space explosion. Finding a proper precision appears to require ingenuity.

Counterexample-guided abstraction refinement [Clarke et al. 2000] provides a framework for automatically finding a proper precision. Consider a path $l_0 \xrightarrow{op_0} l_1 \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} l_n \xrightarrow{op_n} l_{err}$. After predicate analysis, we obtain ϕ_i's and ϕ_{err} such that $\phi_0 = \top$, $(SP_{op_i}(\phi_i))^\lambda \Rightarrow \phi_{i+1}$ for $0 \leq i \leq n-1$

and $(SP_{op_n}(\phi_n))^\lambda \Rightarrow \phi_{err}$. Then the sequence

$$\sigma = (l_0, \phi_0) \xrightarrow{op_0} (l_1, \phi_1) \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} (l_n, \phi_n) \xrightarrow{op_n} (l_{err}, \phi_{err})$$

is called an *abstract counterexample*. Since each ϕ_{i+1} only approximates $SP_{op_i}(\phi_i)$ by definition, an abstract counterexample does not necessarily contain a computation. If there is a computation $\pi = (l_0, c_0) \xrightarrow{op_0} (l_1, c_1) \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} (l_n, c_n) \xrightarrow{op_n} (l_{err}, c_{err})$ with $c_i \models \phi_i$ for $0 \leq i \leq n$ and $c_{err} \models \phi_{err}$, then the abstract counterexample σ is *feasible* and the computation π is a *counterexample* witnessing the error. Otherwise, σ is *spurious*.

When an abstract counterexample is spurious, it means the abstract precision is too coarse. Techniques are available to refine abstraction by adding more predicates to the abstract precision. After refining the abstract precision, more assertions are available for annotation. Predicate analysis is more likely to find a certified annotation or a feasible abstract counterexample.

Starting from an initial precision (usually, the empty set), the CEGAR framework iteratively checks if the program is safe or not using the abstract semantics with the current precision. If it is safe, the program is also safe. The underlying verification algorithm thus terminates and reports "safe". Otherwise, the analyzer returns an abstract counterexample. The algorithm then checks if the abstract counterexample is feasible or not. If it is, the framework terminates and reports "unsafe". Otherwise, the precision is too coarse and needs to be refined to remove the spurious abstract counterexample. The above process repeats until either "safe" or "unsafe" is reported.

The precision is not necessarily unchanged throughout the program [Henzinger et al. 2002]. In general, a precision is a mapping that associates a predicate set on each location of the program. In this paper, we assume a single predicate set for a precision to simplify the discussion.

3 ASSERTION ANNOTATION

After predicate analysis, each location is associated with a set of abstract states. These abstract states contain information to be reused in regression verification. We will characterize them by Hoare triples.

3.1 Hoare Triples and Annotation

Let φ and ψ be predicates and op an operation. A *Hoare triple* is of the form $\{\varphi\}op\{\psi\}$. The triple $\{\varphi\}op\{\psi\}$ is *valid* if the program always arrives at a state satisfying ψ after executing op from a state satisfying φ. By definition, $\{\varphi\}op; op'\{\psi\}$ is valid if there is a predicate ρ such that both $\{\varphi\}op\{\rho\}$ and $\{\rho\}op'\{\psi\}$ are valid. Let $op_1; op_2; \cdots op_n$ be a sequence of operations. The validity of $\{\varphi\}op_1; op_2; \cdots op_n\{\psi\}$ is defined inductively.

Let an *assertion* be a disjunction of abstract states. Define annotations over CFAs as follows.

Definition 3.1. An *annotation* $\mathcal{I}$ of P is a mapping from locations of P to assertions.

Moreover, an annotation of a CFA is *valid* if the initial location is mapped to true and every edge of the CFA forms a valid Hoare triple.

Definition 3.2. An *annotation* $\mathcal{I}$ of P is *valid* if: 1) $\mathcal{I}(l_0) = \top$, and 2) for every edge (l, op, l') of P, $\{\mathcal{I}(l)\}op\{\mathcal{I}(l')\}$ is valid.

Intuitively, a valid annotation $\mathcal{I}$ of a CFA P is an over-approximation of program behaviors. Let $(l_0, c_0) \xrightarrow{op_0} (l_1, c_1) \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} (l_n, c_n)$ be a computation of the program P. For any valid annotation $\mathcal{I}$ of P, we have $c_i \models \mathcal{I}(l_i)$ for every $0 \leq i \leq n$. Note that the annotation maps every location to $\top$ is trivially valid. We are interested in certified annotations.

Definition 3.3. An annotation $\mathcal{I}$ of P is *certified* if it is valid and $\mathcal{I}(l_{err}) = \bot$.

Let $\mathcal{I}$ be a certified annotation. Suppose there is a computation $(l_0, c_0) \xrightarrow{op_0} (l_1, c_1) \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} (l_n, c_n) \xrightarrow{op_n} (l_{err}, c_{err})$ from the initial location to the error location. Since $\mathcal{I}$ is certified, $\mathcal{I}(l_{err}) = \bot$ and hence $c_{err} \models \bot$. This is absurd. Subsequently, if there is a certified annotation $\mathcal{I}$, then there is no computation from the initial location l_0 to the error location l_{err}.

Lemma 3.4. *A program P is* safe *if there exists a certified annotation.*

3.2 Assertion Annotation from Predicate Analysis

Recall that abstract states are computed at each location after predicate analysis. Let *reached* be the set of reached abstract states computed by predicate analysis (Algorithm 1). It is easy to construct an annotation. For each location l, let $\mathcal{I}(l)$ be the disjunction of all reached abstract data states at l (Algorithm 2).

Algorithm 2: Annotation$(P, reached)$

Input: A CFA P and a set *reached* of abstract states of P
Output: An annotation $\mathcal{I}$ of P

```
1 foreach l of P do I(l) ← ⊥ ;
2 foreach (l, φ) ∈ reached do
3 |   I(l) ← I(l) ∨ φ
4 end
5 return I;
```

Lemma 3.5. *The annotation constructed by Algorithm 2 is valid. Particularly, if the program is safe, the constructed annotation is certified.*

Proof. First, $\mathcal{I}(l_0) = \top$ holds (Algorithm 1). Second, let $g = (l, op, l')$ be an arbitrary edge. Recall that *reached* is a fixed point of the data-flow equation system of the program [Nielson et al. 2015]. $\mathcal{I}(l)$ and $\mathcal{I}(l')$ encode the set of abstract data states at l and l' in *reached*, respectively. Thus $\{\mathcal{I}(l)\}op\{\mathcal{I}(l')\}$ is valid. The annotation $\mathcal{I}$ is valid (Definition 3.2).

If the program is safe, $\mathcal{I}(l_{err})$ must be $\bot$. The constructed annotation is thus certified. □

The annotation $\mathcal{I}(l)$ represents the set of all reachable abstract states at l. If the assertion at l_{err} is $\bot$, we conclude that the program is safe and return the annotation as a proof. The constructed annotation is essentially a set of assertions $\{\mathcal{I}(l) \mid l \in L\}$. The number of assertions is limited by the number of locations (or, the number of blocks) in the program. It is quite compact for storing, efficient for processing, and also relatively easy for reading. Naturally, the assertion annotation is a suitable intermediate result for reuse in incremental predicate analysis.

3.3 Assertion Annotation for CEGAR

The concept of assertion annotation can also be adapted to the framework of counterexample-guided abstraction refinement (CEGAR) [Clarke et al. 2000]. Let $\mathcal{U}$ be the set of all assertions that can be used for annotations in predicate analysis. Each assertion $\varphi \in \mathcal{U}$ is in the disjunctive normal form with all atomic predicates from the precision λ. At the beginning of CEGAR, λ is empty and $\mathcal{U} = \{\top, \bot\}$. The error location is very likely to be annotated with $\top$ and gives a non-certified annotation.

Let $\mathcal{I}$ be a valid annotation of P with $\mathcal{I}(l_{err}) \neq \bot$. For any path $l_0 \xrightarrow{op_0} l_1 \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} l_n \xrightarrow{op_n} l_{err}$ on P, we have $\{\mathcal{I}(l_i)\}op_i\{\mathcal{I}(l_{i+1})\}$ is valid for every $0 \leq i \leq n-1$ and $\{\mathcal{I}(l_n)\}op_n\{\mathcal{I}(l_{err})\}$.

Hence

$$\sigma = (l_0, \mathcal{I}(l_0)) \xrightarrow{op_0} (l_1, \mathcal{I}(l_1)) \xrightarrow{op_1} \cdots \xrightarrow{op_{n-1}} (l_n, \mathcal{I}(l_n)) \xrightarrow{op_n} (l_{err}, \mathcal{I}(l_{err}))$$

is an abstract counterexample. If σ is feasible, there is a counterexample witnessing the error. Otherwise, new predicates can be found and added to the precision λ. The set $\mathcal{U}$ of assertions and hence the abstraction are refined. One then computes another valid annotation. The process continues until a certified annotation or a counterexample is found.

CEGAR may involve a number of iterations. Only when the program is safe, and at the last CEGAR iteration, can the constructed annotation be certified. We thus choose to only construct the annotation at the end of the last iteration of CEGAR. The benefits are two-folds: (1) the whole technique is efficient with only one construction of annotation, and (2) the implementation cost is relatively low.

4 INCREMENTAL PREDICATE ANALYSIS

4.1 Overview

Given two program versions P and Q, the individual verification of P or Q can both be considered as the process of constructing annotations. Assume P has already been verified, the goal of incremental predicate analysis is to construct an annotation $\mathcal{I}_Q$ for Q by reusing the previously-generated $\mathcal{I}_P$ for P. The main tasks include a pre-processing procedure which reuses an annotation, and a post-processing procedure which constructs an annotation.

More precisely, for each location l of Q, we find the corresponding location of ℓ in P (Section 4.2), and utilize the assertion $\mathcal{I}_P(\ell)$ to construct $\mathcal{I}_Q(l)$. Note that we cannot directly set $\mathcal{I}_Q(l) = \mathcal{I}_P(\ell)$ due to program changes. The following research question need be addressed: *How does the revision Q change the previously-generated assertions $\mathcal{I}_P$?* To this end, we propose the technique of *change impact analysis* (Section 4.3) to find the set of all impacted variables at each location. If all variables of an assertion are not impacted by the program changes, we can safely reuse it. Otherwise, we employ a *strengthen* algorithm to reuse part of that assertion (Section 4.4). Finally, the IPA algorithm (Section 4.5) performs incremental predicate analysis. It constructs an annotation $\mathcal{I}_Q$ for further reuse, and then reports the verification result.

The following design rules are observed in the development of our technique:

- The reuse procedure must be *sound*. The reused assertions must semantically hold for Q;
- The reuse procedure need *not be complete*. Not all reusable assertions (or all reusable parts of the assertions) need be precisely identified; and
- The reuse procedure should be as *efficient* as possible.

In the following, we use subscripts P, Q to distinguish elements such as locations, edges, annotations of these two versions.

Example 4.1. We use a simple program to explain our technique. This program is shown in the left side of Fig. 2(a) and its CFA is shown in the left side of Fig. 3. In the program, we declare an array of length N. The `do-while` loop in the `main` procedure repeatedly updates the variable `old` as `new`. The branch condition at ℓ_2 denotes a nondeterministic condition. When exiting the loop, the array index `i` is checked. A warning is raised if the index is out of bound. Finally, the program checks if the assertion `new==old` holds or not at ℓ_8. Two special nodes ℓ_{err} and ℓ_{ret} are used to represent the error and return locations, respectively.

After predicate analysis with the precision $\lambda = \{i < N, new = old\}$, a possible annotation is shown in the right side of Fig. 3. The annotation on the error location is $\bot$. The annotation is certified. The program is therefore safe.

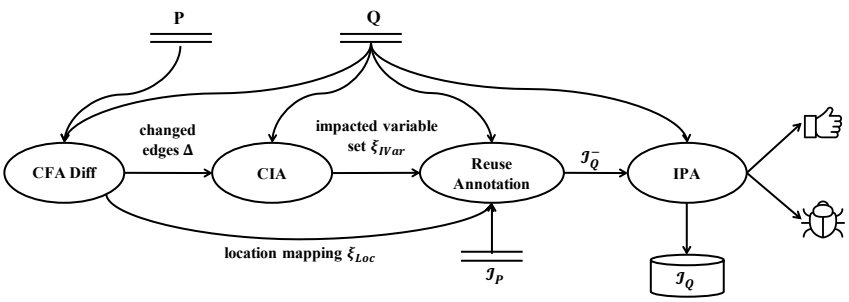


Fig. 1. Overview of incremental predicate analysis

```
0:int q[N];
  main(){
     int i=0;
1:   do {
       old=new;
2:     if(*) {
3:       new++;
4:       q[i++]=old;
       }
5:   } while(new != old);
6:   if(i >= N)
7:     printf("WARNING");
8:   assert(new == old);
  }
```

(a) Old version

```
...
3.1:  if(i == N-1)
3.2:     i--;
3.3:  new++;
4:    q[i++]=new;
...
```

(b) New version

Fig. 2. An example program

4.2 Program Differences

In order to analyze impacts of program changes between control flow automata of P and of Q, we need syntactic differences between the two automata. Their differences are recorded in the following two structures:

- $\Delta \subseteq G_Q$ is the set of (syntactically) changed edges of Q against P, and
- $\xi_{Loc} : L_Q \to L_P \cup \{\clubsuit\}$ is a mapping from Q's locations to P's locations or $\clubsuit$ where $\xi_{Loc}(l) = \clubsuit$ indicates that l has no matched location in P.

To compute program differences, we implemented a CFADiff procedure based on the *similarity flooding* algorithm [Melnik et al. 2002]. Given two control flow automata P and Q, we build a *pairwise connectivity graph* (PCG) by creating a node for each pair of locations $(l, \ell) \in L_Q \times L_P$, and connecting an edge from (l, ℓ) to (l', ℓ') iff $(l, op, l') \in G_Q$ and $(\ell, op, \ell') \in G_P$. We use the PCG to compute the *similarity* between each pair of locations in $L_Q \times L_P$. For each location l, let $str(l)$ be the concatenation of operations (as strings) on all edges entering or emitting l, called the *representative string* of l. The similarity of (l, ℓ) is initialized as the string similarity of $str(l)$ and $str(\ell)$. Then we follow the PCG edges to propagate the similarity values of each node to other nodes [Melnik et al. 2002]. Finally, we compute ξ_{Loc} as the set of pairs of locations whose similarity values exceed a pre-defined threshold. The set Δ can be easily obtained with the assist of the mapping ξ_{Loc}.

Example 4.2. Recall the program in Fig. 2(a). If the `do-while` loop is executed more than N times, the program may raise a warning. To rule out this warning, we revise the program such that the

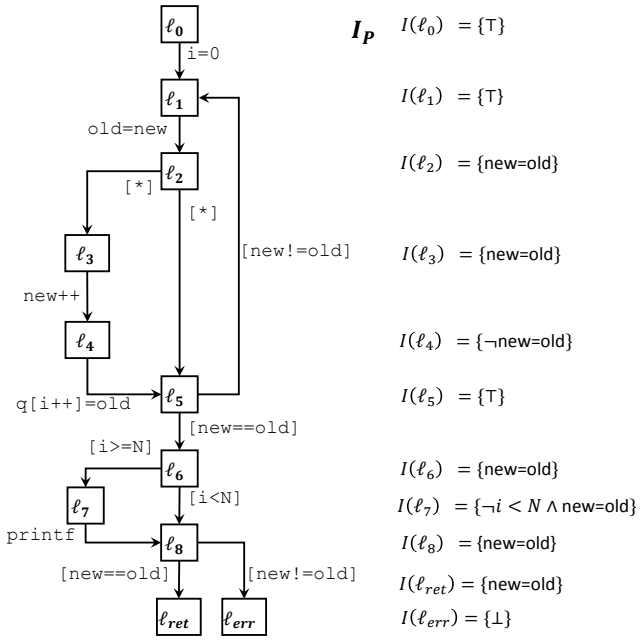


Fig. 3. CFA of program in Fig. 2(a)

original blocks op_3 and op_4 are replaced by $op_{3.1}$ $op_{3.2}$, $op_{3.3}$ and op_4 (Fig. 2(b)). Fig. 4 shows the control flow automaton of the revised program.

Comparing control flow automata of Fig. 3 and Fig. 4, we obtain the set of changed edges $\Delta = \{(l_{3.1}, \texttt{"[i==N-1]"}, l_{3.2}), (l_{3.1}, \texttt{"[i!=N-1]"}, l_{3.3}), (l_{3.2}, \texttt{"}i--\texttt{"}, l_{3.3}), (l_4, \texttt{"q[i++]=new"}, l_5)\}$, and the location mapping

$$\xi_{Loc}(l_i) = \begin{cases} \ell_3 & \text{if } i = 3.1 \\ \clubsuit & \text{if } i \in \{3.2, 3.3\} \\ \ell_i & \text{otherwise.} \end{cases}$$

Note that locations in P and Q are denoted using ℓ and l, respectively. The changed edges Δ are red in Fig. 4.

4.3 Change Impact Analysis

Denote Δ the set of changed edges. Change impact analysis (Algorithm 3) computes the set of impacted variables at each location of Q. We first compute the impacted edges by Δ using the forward slicing technique [Orso et al. 2003] (lines 1 – 2). Specifically, for each edge $g \in \Delta$, we analyze the set of impacted edges using CheckImpact algorithm and then compute their union set as Δ^*.

We then set Δ^* as the criterion and analyze the set of impacted variables at each location $l \in L_Q$. The impact analysis algorithm computes iteratively (lines 5 – 22). Let $reached^L$ and $waitlist^L$ be the set of locations that have been reached and that are to be processed, respectively. Let l be the current location. Note that l is fetched from $waitlist^L$ in the FIFO order. For any successor location l' of l, the impacted variables $\xi_{IVar}(l)$ at l are propagated to l' (line 9). If the edge (l, op, l') is in Δ^*, the impacted variables by op are also added. Specifically, if the operation op is an `assume` statement,

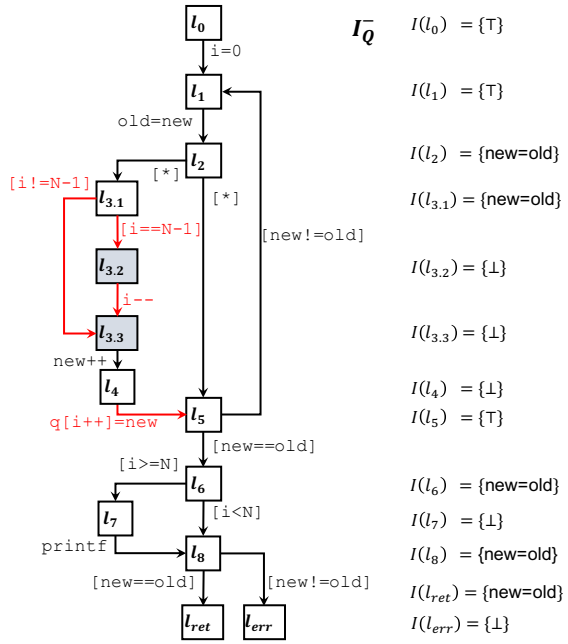


Fig. 4. CFA of program in Fig. 2(b)

all variables in op ($\textsc{Vars}(op)$) are added to $\xi_{IVar}(l')$ due to control impact; if op is an `assignment` statement, the variable written by op ($\textsc{LVar}(op)$) are added to $\xi_{IVar}(l')$ due to data impact. Finally, if l' is not previously visited or $\xi_{IVar}(l')$ is updated in this iteration, we add l' to $waitlist^L$.

The CheckImpact algorithm is listed in Algorithm 4. Given an edge g as input, the algorithm computes the set of all edges that are (directly or indirectly) impacted by g. Let $reached^G$ and $waitlist^G$ be the sets of edges that have been explored and that are to be processed, respectively. Let $g = (l, op, l')$ be the edge taken from $waitlist^G$ (in the FIFO order). If op is an `assume` statement, the edge g is a conditional jump edge. All statements dominated by g compose the forward slice Δ^*_{fw} (line 5). If op is an `assignment` statement with the left-hand-side variable v, the forward slice Δ^*_{fw} is the set of statements that read v after op (line 7). At the end of the iteration, edges in Δ^*_{fw} are all added to $waitlist^G$ except those that have been processed (line 11). The resulting Δ^*_g is the union of forward slices computed in all iterations.

Note that the above analysis does not require any semantic computation. It uses the light-weight location analysis only and is computationally efficient. Moreover, Δ^* is always an over-approximation of the semantically impacted set of edges by forward slicing. The mapping ξ_{IVar} is also computed conservatively such that $\xi_{IVar}(l)$ is a superset of semantically impacted variables at l. We thus can safely use $\xi_{IVar}(l)$ to find the non-impacted variables at each location l.

Example 4.3. By change impact analysis on the changed edges Δ of Fig. 4, the set of impacted edges is $\Delta \cup \{(l_6, \texttt{"[i>=N]"}, l_7), (l_6, \texttt{"[i<N]"}, l_8), (l_7, \texttt{"printf"}, l_8)\}$, the resulting affected variable mapping is

$$\xi_{IVar}(l_i) = \begin{cases} \emptyset & \text{if } i = 0 \\ \{\texttt{i}, \texttt{q}\} & \text{otherwise.} \end{cases}$$

Algorithm 3: CIA(Q, Δ)

Input: Q, Δ
Output: The impacted variable set $\xi_{IVar} : L \rightarrow 2^V$

```
   /* compute the impacted edges                                          */
1  Δ* ← ∅ ;
2  foreach g ∈ Δ do Δ* ← Δ* ∪ CheckImpact(g) ;
   /* compute the impacted variables at each location                     */
3  foreach l ∈ L_Q do ξ_IVar(l) ← ∅ ;
4  reached^L ← {l_0}, waitlist^L ← {l_0};
5  while waitlist^L ≠ ∅ do
6      pop l from waitlist^L ;
7      foreach l' such that g = (l, op, l') ∈ G_Q do
8          old ← ξ_IVar(l');
9          ξ_IVar(l') ← ξ_IVar(l') ∪ ξ_IVar(l) ;                  /* propagation */
10         if g ∈ Δ* then
11             if op is an assume statement then
12                 ξ_IVar(l') ← ξ_IVar(l') ∪ Vars(op) ;           /* control impact */
13             else if op is an assignment statement then
14                 ξ_IVar(l') ← ξ_IVar(l') ∪ LVar(op) ;           /* data impact */
15             end
16         end
17         if l' ∉ reached^L or ξ_IVar(l') ≠ old then
18             waitlist^L ← waitlist^L ∪ {l'};
19         end
20         reached^L ← reached^L ∪ {l'};
21     end
22 end
23 return ξ_IVar;
```

4.4 Annotation Reuse

The basic idea of our incremental predicate analysis is to construct an initial annotation for Q by reusing the previously generated annotation for P. Due to program changes, not all assertions of P apply to Q. We rely on ξ_{Loc} and ξ_{IVar} to determine if an assertion of P can be reused in Q or not. If not, we furthermore develop a technique for strengthening the original assertion for reusing.

Algorithm 5 depicts our ReuseAnnotation procedure. It starts from $(l_0, \top)$ and traverses all reachable abstract states of Q. Specifically, denote $reached^L$ to be the set of visited locations and $waitlist$ the set of abstract states to be processed. Let (l, ϕ) be the current abstract state taken from $waitlist$ and l' be a successor location of l. If $\xi_{Loc}(l') = \clubsuit$ (line 5), then nothing can be reused. We set $\mathcal{I}_Q^-(l')$ to $\bot$ and continue the traversal. Otherwise, we utilize its previous annotation $\mathcal{I}_P(\xi_{Loc}(l'))$ to construct $\mathcal{I}_Q^-(l')$. More precisely, if l' is not previously visited, we apply the Strengthen algorithm on $\mathcal{I}_P(\xi_{Loc}(l'))$ to get a candidate assertion ϕ' (line 9). If l' has been visited, we use its annotation $\mathcal{I}_Q^-(l')$ as the candidate (line 11). If the candidate ϕ' is $\bot$ or it falsifies the Hoare triple $\{\phi\}op\{\phi'\}$, $\mathcal{I}_Q^-(l')$ is set to $\bot$. Otherwise, $\mathcal{I}_Q^-(l')$ is set to ϕ'. The sets $reached^L$ and $waitlist$ are updated accordingly.

Algorithm 4: CheckImpact(g)

Input: An edge g
Output: The set Δ_g^* of edges impacted by g

1 $\Delta_g^* \leftarrow \{g\}$, $reached^G \leftarrow \{g\}$, $waitlist^G \leftarrow \{g\}$;
2 **while** $waitlist^G \neq \emptyset$ **do**
3 pop $g = (l, op, l')$ from $waitlist^G$;
4 **if** *op is an* `assume` *statement* **then**
5 $\Delta_{fw}^* \leftarrow$ ControlImpact(g) ; `/* control impact */`
6 **else if** *op is an* `assignment` *statement* **then**
7 $\Delta_{fw}^* \leftarrow$ RAW(g) ; `/* Read After Write - data impact */`
8 **end**
9 $\Delta_g^* \leftarrow \Delta_g^* \cup \Delta_{fw}^*$;
10 $reached^G \leftarrow reached^G \cup \{g\}$;
11 $waitlist^G \leftarrow waitlist^G \cup \Delta_{fw}^* \setminus reached^G$;
12 **end**
13 **return** Δ_g^*;

Algorithm 6 shows our Strengthen algorithm. It takes an assertion φ and a location l as inputs and returns a strengthened assertion ψ. Let $\xi_{IVar}(l)$ be the set of impacted variables at l. If the set of variables in φ (Vars(φ)) is disjoint from $\xi_{IVar}(l)$, φ is irrelevant to program changes and can be safely reused (line 2). Otherwise, φ cannot be reused directly. In the latter case, we wish to find reusable parts of φ. Given that φ is in the disjunctive norm form (see Algorithm 2), we check whether each *cube* of φ is impacted by the program changes or not (line 6). Cubes that are irrelevant to program changes are added to ϕ (lines 6 – 7). Since ψ is obtained by removing impacted cubes from φ, we have $\psi \Rightarrow \varphi$.

Algorithm 5 is quite efficient. During the exploring procedure, all abstract data states in Algorithm 5 are not computed semantically but strengthened from existing assertions syntactically. Secondly, the algorithm need not explore all locations of Q. It stops exploring successors of the location l only if the constructed annotation at l is $\bot$.

Lemma 4.4. *Let $\mathcal{I}_Q^-$ be the annotation returned by Algorithm 5, for any edge (l, op, l') of Q with $\mathcal{I}_Q^-(l') \neq \bot$, $\{\mathcal{I}_Q^-(l)\}op\{\mathcal{I}_Q^-(l')\}$ is valid.*

Example 4.5. Recall the motivating example. After applying ReuseAnnotation, we get the annotation $\mathcal{I}_Q^-$ as shown in the right side of Fig. 4, where the assertion $\mathcal{I}_Q^-(l_{3.2})$ is set to $\bot$ since $l_{3.2}$ is newly added location. The assertions $\mathcal{I}_Q^-(l_{3.3})$ and $\mathcal{I}_Q^-(l_4)$ are also $\bot$ because the exploration from $l_{3.2}$ is stopped. $\mathcal{I}_Q^-(l_7)$ is strengthened to $\bot$ since the variables in $\mathcal{I}_P^-(\xi_{Loc}(l_7))$ overlaps $\xi_{IVar}(l_7)$.

4.5 Incremental Algorithm for Regression Verification

Our incremental predicate analysis algorithm takes a program Q and an initial annotation $\mathcal{I}_Q^-$ as inputs. It returns either "safe" with a certified annotation $\mathcal{I}_Q$ or "unsafe" with a feasible abstract counterexample σ. Standard predicate analysis can be seen as a special case of our algorithm by setting the initial annotation on each location to $\bot$.

Algorithm 7 presents our incremental predicate analysis algorithm integrated with the CEGAR framework. The algorithm consists of three phases. In the first phase, the algorithm initializes

Algorithm 5: REUSEANNOTATION(Q, $\mathcal{I}_P$, ξ_{Loc}, ξ_{IVar})

Input: A program Q, an annotation $\mathcal{I}_P$ for P, a location mapping ξ_{Loc}, and an impacted-variable mapping ξ_{IVar}

Output: An annotation $\mathcal{I}_Q^-$ for Q

```
1  reached^L ← {l_0}, waitlist ← {(l_0, ⊤)} ;
2  while waitlist ≠ ∅ do
3      pop (l, φ) from waitlist ;
4      foreach l' such that (l, op, l') ∈ G_Q do
5          if ξ_Loc(l') = ♣ then                                  /* new location */
6              I_Q^-(l') ← ⊥; Continue;
7          end
8          if l' ∉ reached^L then
9              φ' ← STRENGTHEN(I_P(ξ_Loc(l')), l')                /* candidate assertion */
10         else
11             φ' ← I_Q^-(l');
12         end
13         if φ' = ⊥ or {φ}op{φ'} is invalid then                 /* validate */
14             I_Q^-(l') ← ⊥ ;
15         else
16             I_Q^-(l') ← φ' ;
17             reached^L ← reached ∪ {l'};
18             waitlist ← waitlist ∪ {(l', I_Q^-(l'))};
19         end
20     end
21 end
22 return I_Q^-;
```

Algorithm 6: STRENGTHEN(φ, l)

Input: An assertion φ in DNF, and a location l of Q

Output: A strengthened assertion ψ

```
1  if VARS(φ) ∩ ξ_IVar(l) = ∅ then
2      ψ ← φ ;
3  else
4      ψ ← ⊥ ;
5      foreach cube ∈ φ do
6          if VARS(cube) ∩ ξ_IVar(l) = ∅ then
7              ψ ← ψ ∨ cube ;
8          end
9      end
10 end
11 return ψ;
```

reached and *waitlist* by $\mathcal{I}_Q^-$ (lines 1 – 2). The initial abstract precision is initialized as the set of atomic predicates occurred in $\mathcal{I}_Q^-$ (line 3). The second phase (lines 4 – 13) is exactly as same as in classical predicate analysis (Algorithm 1). After a fixed point of *reached* is obtained, we check if the error location is reached in the third phase. If it is not in *reached*, the algorithm concludes that the program is "safe" and returns the annotation as a proof (lines 15 – 16). Otherwise, an abstract counterexample $(l_0, \top) \xrightarrow{op_0} \cdots \xrightarrow{op_n} (l_{err}, \phi_{err})$ is obtained. If this abstract counterexample is spurious, the algorithm employs the Refine algorithm [Beyer et al. 2008; Cimatti et al. 2008; McMillan 2006] for refining the current precision λ (line 22) and restarts the second phase.

Our incremental algorithm extends classical predicate analysis by allowing an initial annotation. Note that the annotation $\mathcal{I}_Q^-$ is constructed from $\mathcal{I}_P$. For each location l, the annotation $\mathcal{I}_Q^-(l)$ represents a set of abstract states that were explored in the verification of P. By reusing $\mathcal{I}_Q^-(l)$, our algorithm removes the redundancy of exploring same abstract states in $\mathcal{I}_P(l)$ during the verification of Q. In other words, the exploring history in the verification of the original program is reused in regression verification. If the program Q is safe, the returned $\mathcal{I}_Q$ can also be used for regression verification of subsequent revisions.

Lemma 4.6. *The annotation $\mathcal{I}_Q$ constructed at line 15 in Algorithm 7 is a certified annotation.*

Proof. Let l_0 be the initial location of Q, we have $\mathcal{I}_Q^-(l_0) = \top$ (line 1, Algorithm 5) and $\mathcal{I}_Q(l_0) = \mathcal{I}_Q^-(l_0)$ (line 2, Algorithm 7).

For any abstract state $(l', \phi') \in reached$ with l' different from l_0, there are two cases:

- if the abstract data state ϕ' is set by $\mathcal{I}_Q^-$ (line 1, Algorithm 7), there must be another abstract state $(l, \phi) \in reached$ such that $\phi = \mathcal{I}_Q^-(l)$ and $\{\phi\} op \{\phi'\}$ is valid (Lemma 4.4).
- if the abstract data state ϕ' is computed at at line 7 of Algorithm 7, there must be a predecessor abstract state $(l, \phi) \in reached$ such that $\phi' = (SP_{op}(\phi))^{\lambda}$. Hence $\{\phi\} op \{\phi'\}$ is valid.

To summarize, for any abstract state $(l', \phi') \in reached$, there must be an abstract state $(l, \phi) \in reached$ such that $\{\phi\} op \{\phi'\}$ is valid. Given that $\mathcal{I}_Q(l)$ and $\mathcal{I}_Q(l')$ are both disjunctions of reached abstract data states (line 3, Algorithm 2), we conclude that $\{\mathcal{I}_Q(l)\} op \{\mathcal{I}_Q(l')\}$ is valid.

Therefore, $\mathcal{I}_Q$ is valid (Definition 3.2). Moreover, since l_{err} is not reached, its annotation is $\bot$. The constructed annotation is certified (Definition 3.3). □

Theorem 4.7. *If Algorithm 7 returns ("Safe", $\mathcal{I}_Q$), the program is safe and $\mathcal{I}_Q$ is a certified annotation. If Algorithm 7 returns ("Unsafe", σ), the program is unsafe and σ is a feasible abstract counterexample.*

4.6 Discussion

Our technique can be understood more easily through abstract interpretation [Cousot and Cousot 1977]. If one sees predicate analysis as an instance of abstract interpretation, annotations are simply fixed points in the predicate abstract domain. The program verification problem is hence reduced to fix-point computation in abstract interpretation. In order to verify a revised program, it suffices to compute a fixed point for the revision based on the fixed point for the original. To do so, we first identify potential semantic changes through syntactic heuristics in change impact analysis (Section 4.3). Based on impact analysis, unaffected abstract states computed in the original program are reused or strengthened by syntactic heuristics. Fixed point computation for the revised program then starts from such abstract states (Section 4.5). Compared to accurate semantic impact or strengthening techniques, our syntactic heuristics are much more efficient. Trade-off between efficiency and accuracy is again not uncommon in abstract interpretation. We carefully design our incremental predicate analysis for efficiency. Our design choices will be justified in the next section.

Algorithm 7: IPA($Q, \mathcal{I}_Q^-$)

Input: A program Q and an initial annotation $\mathcal{I}_Q^-$

Output: "Safe" and a certificate $\mathcal{I}_Q$, or "unsafe" and a feasible abstract counterexample σ

```
   /* Phase 1: Initialization                                          */
 1 reached ← {(l, I_Q^-(l)) | I_Q^-(l) ≠ ⊥};
 2 waitlist ← {(l, I_Q^-(l)) | I_Q^-(l) ≠ ⊥};
 3 λ ← {p | p occurs in I_Q^-(l) for some l};
   /* Phase 2: Compute the reachable set w.r.t. λ                      */
 4 while waitlist ≠ ∅ do
 5     pop (l, φ) from waitlist;
 6     foreach l' with (l, op, l') ∈ G do
 7         φ' ← (SP_op(φ))^λ;
 8         if (l', φ') is not covered by reached then
 9             reached ← reached ∪ {(l', φ')};
10             waitlist ← waitlist ∪ {(l', φ')};
11         end
12     end
13 end
   /* Phase 3: Report or refine                                        */
14 if ∄φ_n.(l_err, φ_n) ∈ reached then
15     I_Q ← ANNOTATION(Q, reached);
16     return ("Safe", I_Q);
17 else
18     Let σ be an abstract counterexample (l_0, ⊤) --op_0--> ··· --op_n--> (l_err, φ_err) in reached;
19     if σ is feasible then
20         return ("Unsafe", σ);
21     else
22         λ ← REFINE(λ, σ);
23         goto 4 ;                                                    // Phase 2
24     end
25 end
```

Our technique can also be applied to other abstract domains. We briefly explain how to adapt our techniques in the explicit-value domain. In the explicit-value domain, we wish to track explicit values of certain program variables. An abstract precision λ in the explicit-value domain is a subset $X \subseteq V$ of program variables. We are interested in values of program variables in X.

Consider an integer variable $v \in X$, its abstract domain is $\mathbb{Z} \cup \{\top_Z, \bot_Z\}$. $\top_Z$ means v can take any values in $\mathbb{Z}$. $\bot_Z$ indicates no possible assignment for v. An abstract state in the explicit-value domain is a pair (l, s) where l is a location and s is a valuation of all variables in λ.

Change impact analysis relies only one program texts and is applicable regardless of abstract domains. Let (l, s_1), (l, s_2) be two abstract states in the explicit-value domain. For any $v \in X$, define

$$(s_1 \vee s_2)(v) = \begin{cases} s_1(v) & if\ s_1(v) = s_2(v) \\ \top_Z & otherwise. \end{cases}$$

That is, the join of any two abstract data states is another abstract data state. Recall Algorithm 2. The above definition ensures that assertions on any location is an abstract data state in the explicit-value domain. Given a set of impacted variables $\xi_{IVar}(l)$ at l and an abstract data state s, define the abstract data state $\textsc{Strengthen}(s)$ as follows (Algorithm 6).

$$\textsc{Strengthen}(s)(v) = \begin{cases} \bot_Z & v \in \xi_{IVar}(l) \\ s(v) & v \notin \xi_{IVar}(l) \end{cases}$$

for any $v \in X$. With slight but standard modification, Algorithm 7 can then be used to analyze explicit values of variables incrementally.

5 EVALUATION

In this section, we describe our implementation and conduct extensive experiments on industrial programs to evaluate performance of our technique. Through the experiments, we will answer the following research questions:

- **RQ1:** How effective is our technique compared to classical predicate analysis in practice?
- **RQ2:** How significant is the impact of annotation reuse on efficiency of our technique?

5.1 Implementation

We implemented our algorithms on top of the software verification tool CPAchecker [Beyer and Keremoglu 2011]. CPAchecker provides a configurable program analysis framework and offers mainstream software verification algorithms such as predicate analysis and CEGAR. By adopting these techniques from CPAchecker, it is straightforward to integrate our incremental algorithm (Algorithm 7) in the program analysis framework.

Specifically, we implement the following features in CPAchecker to realize our incremental predicate analysis technique:

- a CFADiff procedure[1];
- a static change impact analysis procedure;
- a persistent representation (the SMT-LIB format [Barrett et al. 2010]) for annotations; and
- the annotation reuse and strengthening procedures.

5.2 Experimental Setup

We used 1,119 real revisions of 62 Linux device drivers to conduct the empirical evaluation. All revisions were extracted from the Linux kernel and prepared for verification by the LDV toolkit [Khoroshilov et al. 2010; Mandrykin et al. 2012]. A device driver may have multiple specifications obtained from [Beyer et al. 2013]. A *verification task* consists of a program revision and a specification. For each device driver, the initial version is verified from scratch. Verification tasks of the same device driver and the same specification are performed incrementally. The annotation of the previous revision is reused in the next revision. There are 259 driver/specification pairs (called *verification cases*) resulted in 4,193 verification tasks. Among them, 3,934 are regression verification tasks.

[1]The implementation is based on https://github.com/kientuong114/SimilarityFlooding.

All experiments are carried out on a machine with 2.6 GHz Intel Xeon E5-2640 CPU with 32 cores and 128GB RAM. We use Ubuntu 16.04 (64-bit) with Linux 4.4.0 and jdk1.8.0. CPAchecker is configured with the *predicateAnalysis-ABE* [Beyer et al. 2010] option. Each verification task is limited to 300 seconds in CPU time, 2 CPU cores and 10 GB in Java heap size.

5.3 Overall Results of *IPA* (RQ1)

We present the overall experimental results of our technique in this section. Classical predicate analysis (called *PA*) is the baseline algorithm in this comparison. *IPA* refers to our incremental predicate analysis technique. To evaluate the effect of *IPA* and answer **RQ1**, we use the same configurations for *PA* and *IPA* in CPAchecker.

Detailed experimental results are shown in Table 1. Each line represents a *verification case*. Only 40 best and 10 worst cases are listed due to space limitation. In *Verification Case*, the following information is provided:

- *Driver* is the name of device driver;
- *Spec.* is the name of specification (see [Beyer et al. 2013]);
- *LoC* denotes the lines of code for the initial version;
- *#Solve* is the number of regression verification tasks solved by both techniques. The expression $X + Y$ means that X are solved by both techniques and additional Y tasks are solved by *IPA*;
- T_{1st} is the analysis time for verifying the first version.

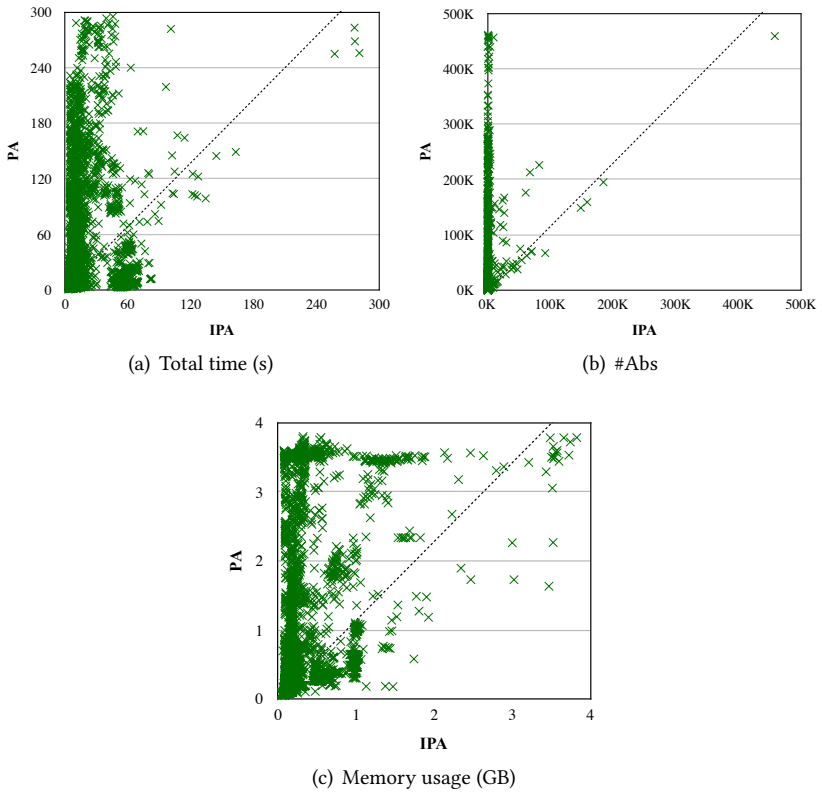
The columns *PA* and *IPA* present the experimental results of baseline and our technique. We report the number of abstract states (*#Abs*) and the analysis time T_a of regression verification tasks solved by both techniques in Table 1. The metric *#Abs* measures the computation efforts of abstract state space, T_{diff} measures the time for calculating program differences (CFADiff), T_{anno} represents the time for reusing annotations (ReuseAnnotation), and T_{IO} represents the time for loading/writing annotations. We also count the total time required by *IPA*, i.e., $T_{total} = T_a + T_{diff} + T_{anno} + T_{IO}$. Note that all the time metrics are the CPU time in seconds. Additionally, the *FSize* column lists the size of annotation files (in kilobyte). The SU_a and *SU* columns show the average speedups of *IPA* over *PA* on the analysis time (calculated by T_a^{PA}/T_a^{IPA}) and on the total time (calculated by T_a^{PA}/T_{total}^{IPA}), respectively. Table 1 is sorted by SU_a. The last two rows (*Total* and *Average*) show the total and average amount of solved verification cases, respectively. Note that 18 verification cases cannot be solved by either techniques. Table 1 thus contains 241 verification case results. Although *IPA* solved 393 more regression verification tasks, Table 1 only reports the 3,407 tasks solved by both techniques.

From the table, we see that *IPA* wins in all verification cases in regard to analysis time (T_a) and the average speedup of analysis time (SU_a) is 16.8x. Comparing the columns *#Abs* of *PA* and *IPA*, we find that our incremental technique reduces the number of abstract states dramatically (97.6% in total). This is consistent with the overall speedup of analysis time (SU_a). It suggests that the computation of abstract states takes up the majority of time in predicate analysis, and *IPA* successfully reduces such computation. When considering the total time, the average speedup (*SU*) is 2.8x. The average time of T_{diff} and T_{anno} among all solved verification cases are 170.1 and 62.9 seconds, respectively, both longer than the average analysis time (46.6s). The annotation I/O time (T_{IO}) and the annotation file size (*FSize*) are negligible – the average time of annotation I/O among all solved cases is 4.8 seconds, and the average size of annotation files among all solved cases is 34.1KB. Looking at the worst 10 cases in Table 1, *IPA* consumes more total time than *PA*. This is mainly due to the unavoidable overheads of CFADiff and ReuseAnnotation procedures. On the other hand, *IPA* also benefits from these two procedures by achieving 2.8x overall speedup.

Table 1. Overall experimental results on real revisions

Verification Case					PA		IPA							SU_a	SU
Driver	Spec.	LoC	#Solve	T_{1st}	#Abs	T_a	#Abs	T_a	T_{diff}	T_{anno}	T_{IO}	T_{total}	FSize		
wm831x	39_7a	5183	33	94.8	7.7M	6.4K	2.0K	14.0	126.6	74.3	9.4	224.3	56.5	459.4	28.6
rtl28xxu	39_7a	10263	9	19.1	43.4K	227.5	30	0.6	34.4	41.5	2.4	78.8	14.2	373.0	2.9
wm831x	08_1a	4579	33	57.8	3.9M	4.1K	2.0K	12.8	129.9	73.4	6.5	222.6	40.3	321.5	18.4
wl12xx	32_7a	6813	37	14.0	341.0K	671.3	167	2.2	136.3	43.1	6.9	188.5	43.2	307.9	3.6
wl12xx	08_1a	6661	37	14.0	324.7K	630.7	167	2.3	139.1	43.7	5.1	190.2	41.0	279.1	3.3
wl12xx	39_7a	8578	37	84.8	3.9M	3.5K	947	14.0	142.3	69.8	12.7	238.8	94.6	250.9	14.7
wm831x	39_7a	4174	19	44.9	369.6K	656.5	264	2.6	50.5	34.5	3.1	90.7	29.1	250.6	7.2
cx231xx	39_7a	9959	1+11	229.6	272.7K	239.7	60	1.0	6.2	4.3	0.3	11.7	2.9	247.1	20.5
wm831x	68_1	6569	2	88.4	347.6K	277.2	259	1.2	5.3	5.7	0.5	12.7	3.4	221.7	21.9
rtl28xxu	32_7a	9296	9	9.2	13.6K	116.2	30	0.5	35.5	39.1	1.7	76.9	10.6	219.2	1.5
gpio	32_7a	3890	19	25.9	287.4K	540.2	269	2.8	51.1	33.7	2.9	90.4	25.4	195.0	6.0
i915	68_1	13916	78	43.3	2.1M	4.1K	4.9K	23.0	1.3K	319.9	31.7	1.7K	150.3	179.4	2.4
wm831x	32_1	5339	2	63.5	261.2K	253.2	259	1.5	5.9	6.5	0.6	14.6	3.2	164.4	17.4
rtl28xxu	08_1a	8705	9	5.5	4.1K	72.7	30	0.5	34.9	36.6	1.3	73.3	8.5	161.6	1.0
i915	32_1	13557	78	32.7	1.4M	3.4K	4.9K	22.4	1.4K	312.8	32.6	1.8K	145.5	150.5	1.9
i915	08_1a	11157	78	34.2	1.1M	3.5K	4.8K	26.0	1.1K	292.9	32.5	1.5K	144.5	135.0	2.4
gpio	08_1a	3524	19	16.2	94.6K	273.6	264	2.5	52.3	32.5	2.2	89.4	21.3	111.7	3.1
tcm_loop	39_7a	12515	30+10	134.3	5.6M	5.1K	3.7K	46.5	133.0	93.4	18.4	291.2	167.5	109.4	17.4
cx231xx	08_1a	8241	1+11	68.7	24.1K	73.8	60	0.7	7.3	4.7	0.1	12.8	1.2	101.1	5.8
cx231xx	32_7a	8393	1+2	68.7	24.1K	74.6	60	0.7	7.1	4.1	0.1	12.0	1.3	100.8	6.2
uartps	39_7a	6518	2	146.1	346.7K	284.2	184	2.9	8.8	10.7	0.8	23.3	14.8	96.7	12.2
uartps	08_1a	5239	2	91.2	95.2K	198.4	124	2.1	9.0	9.0	0.6	20.6	6.0	96.3	9.6
vsxxxaa	68_1	5869	1	32.2	33.1K	49.8	69	0.5	4.9	1.7	0.3	7.4	1.5	94.0	6.7
uartps	32_7a	5393	2	88.3	95.4K	187.4	124	2.1	8.7	9.6	0.6	21.0	6.2	88.8	8.9
dp83640	08_1a	7454	15	130.0	2.4M	2.0K	3.5K	23.8	77.3	57.2	6.2	164.6	42.8	85.3	12.4
vsxxxaa	32_1	5484	1	25.5	21.4K	39.1	69	0.5	4.2	2.0	0.2	6.9	1.4	78.3	5.7
vsxxxaa	43_1a	4373	1	18.3	6.2K	19.0	34	0.3	3.6	1.6	0.2	5.6	1.7	72.9	3.4
i915	39_7a	14298	78	68.0	4.1M	6.6K	20.8K	105.8	1.2K	379.6	49.4	1.7K	327.1	62.5	3.9
sercos3	08_1a	3908	4	6.9	2.8K	27.8	28	0.5	16.3	5.6	0.6	23.0	5.0	57.9	1.2
uartlite	39_7a	6216	8	88.2	666.8K	679.6	1.0K	12.0	31.4	27.3	3.3	74.0	29.9	56.7	9.2
tcm_loop	08_1a	10264	40	40.6	2.8M	2.8K	8.4K	50.4	193.7	84.5	13.1	341.7	66.7	55.1	8.1
az6007	08_1a	7912	4	84.9	1.6M	807.1	4.1K	14.9	19.9	19.0	0.9	54.7	4.9	54.2	14.7
farsync	43_1a	7046	8	32.1	536.4K	458.8	1.3K	8.5	125.9	20.8	2.8	158.0	13.6	54.0	2.9
catc	32_1	6245	9	21.4	294.7K	427.5	1.0K	8.2	57.0	21.8	2.0	89.0	11.1	52.3	4.8
ems_usb	39_7a	9647	20	168.9	4.1M	3.3K	12.5K	63.8	98.0	34.6	9.0	205.4	90.3	51.4	16.0
sil164	39_7a	7026	2	74.3	134.4K	149.8	302	3.2	13.5	6.0	0.6	23.3	5.5	47.1	6.4
matroxfb	39_7a	6148	6	42.8	145.9K	211.9	539	4.6	29.9	14.1	2.4	51.0	18.3	46.6	4.2
keyspan	43_1a	4729	2	3.8	570	7.6	18	0.2	7.7	3.2	0.2	11.3	2.3	44.5	0.7
tcm_loop	32_7a	10415	40	40.5	3885.8K	3.2K	9.7K	73.9	190.3	103.7	16.1	384.0	116.2	42.7	8.2
mtdoops	39_7a	4036	34	22.2	160.3K	587.3	923	13.9	82.2	95.0	5.8	196.8	51.5	42.4	3.0
⋮															
paride	32_7a	5163	8	37.0	85.3K	319.6	70.6K	258.8	33.8	94.8	1.8	389.2	10.0	1.2	0.8
magellan	08_1a	3814	1	15.0	3.2K	15.3	3.2K	12.5	5.9	0.0	0.2	18.6	1.4	1.2	0.8
abyss	32_7a	4036	3	3.0	150.2K	131.6	150.2K	113.1	27.5	3.6	0.4	144.6	3.1	1.2	0.9
algo-pca	43_1a	3031	6	1.5	672	16.8	672	14.5	24.8	0.0	0.7	40.0	5.7	1.2	0.4
ar7part	43_1a	1000	1	0.6	28	1.4	28	1.3	1.9	0.0	0.1	3.2	0.8	1.1	0.4
budget	32_7a	6243	4+2	91.2	250.2K	405.5	250.1K	390.7	25.8	3.9	1.5	421.9	4.7	1.0	1.0
mos7840	32_7a	8869	28+30	5.5	23.5K	159.6	22.0K	155.9	301.1	67.9	10.0	534.8	46.4	1.0	0.3
twidjoy	08_1a	3763	1	10.7	2.0K	9.7	2.0K	9.5	5.7	0.0	0.0	15.2	0.0	1.0	0.6
spaceorb	08_1a	3879	1	15.4	4.6K	14.4	4.6K	14.3	5.8	0.0	0.0	20.1	0.0	1.0	0.7
dmx3191d	32_7a	6842	1	1.9	48.6K	50.1	45.9K	49.8	22.1	0.8	0.3	73.0	1.1	1.0	0.7
Total	-	1.5M	3407 +393	8.2K	153.7M	188.6K	3.7M	11.2K	41.0K	15.1K	1.2K	68.5K	8.2K	-	-
Average	-	6.3K	15	34.0	637.8K	782.7	15.5K	46.6	170.1	62.9	4.8	284.4	34.1	16.8	**2.8**

Fig. 5 compares *PA* and *IPA* in total time, number of abstract states, and memory usage. Each (green) cross compares a metric between two techniques for a regression verification task (3,407 crosses plotted in total). The X- and Y-axes represent the results of *IPA* and *PA*. Crosses above the dotted diagonal line indicate that *IPA* performs better. Fig. 5(a) compares total time. We observe numerous verification tasks near the Y-axis, where *IPA* performs exceptionally better. The verification tasks that below the diagonal take more time on the CFADiff and ReuseAnnotation procedures. Similarly, each cross in Fig. 5(b) compares *IPA* and *PA* in the number of abstract states. The crosses lying on the diagonal represent tasks where both techniques behave similarly. And the crosses close to the Y-axis represent cases where *IPA* reused most of the annotations from previous revisions. Only in very few tasks that *IPA* introduces more abstract states (crosses blew the



(a) Total time (s)

(b) #Abs

(c) Memory usage (GB)

Fig. 5. Comparison of analysis time, number of abstract states and memory usage between *PA* and *IPA*

diagonal), which reveals that in the worst case, the annotations include redundant predicates for the current verification. Recall the line 3 in Algorithm 7, the initial precision λ will be initialized with such redundant atomic predicates, which may introduces more abstract states. Fig. 5(c) compares memory usage of the two techniques. Again, the crosses tend to be above the diagonal. They indicate that *IPA* requires less memory in most tasks. The crosses below the diagonal indicates the verification tasks that consume more memory on the CFADiff and ReuseAnnotation procedures.

IPA solves additional 393 tasks with 43.7 seconds on average. The pie chart in Fig. 6 presents the distribution of analysis time for the 393 tasks. The majority of them (82%) take less than 1 minutes. About 8% of tasks need more time to solve (more than 2 minutes). Fig. 6 demonstrates the efficiency of *IPA* on tasks unsolved by *PA* within 300 seconds time limit.

Answer to RQ1. Extensive experiments on industrial programs suggest that our incremental predicate analysis technique is effective and efficient compared with the classical predicate analysis.

5.4 Effect of Annotation Reuse (RQ2)

Reusability of annotation is a vital metric in our experiments. Fig. 7 illustrates the relationship between *reusable rate (RR)* and the speedup in analysis time (T_a). In the figure, the Y-axis shows the speedup gained by *IPA*. The X-axis represents the minimal *RR* in verification tasks. For example, if the minimal *RR* is 0, SU_a is equal to 16.8. Similarly, we consider the 3,407 tasks solved by both

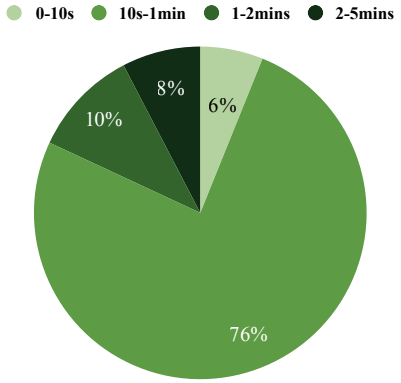


Fig. 6. Distribution of analysis time for *IPA* on 393 timeout tasks

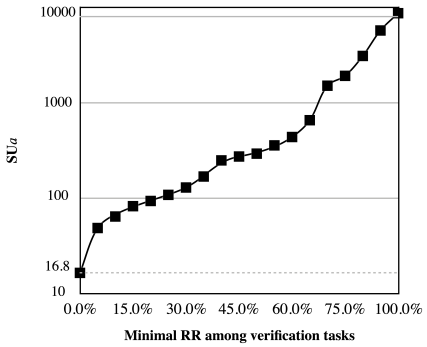


Fig. 7. Speedup on verification tasks with specific minimal *RR*

techniques. In general, the speedup increases considerably with the growth of minimal *RR*. Fig. 7 demonstrates the benefits of annotation reuse.

Answer to RQ2. Annotation reuse has a noticeable and positive impact on the effectiveness of *IPA* in analysis time.

5.5 Discussion

Our implementation of the CIA algorithm is simple. For example, for a given predicate $i < N$, if variable i is considered to be impacted by CIA, all assertions containing $i < N$ are affected. The predicate $i < N$ may not be impacted when i changed. In this respect, a more precise CIA algorithm should be considered. However, our technique achieves considerable performance by introducing only a little overhead into *IPA* in current implementation.

In our experiments, only C programs of Linux device drivers are evaluated. However, it does not mean that our technique is limited to specific language or scope. In fact, our technique is capable of programs that can be verified using predicate analysis on CFA. The Linux device drivers contain many complex syntactic structures and such as struct, array, loop and so on. The effectiveness on this scope thus convinces us that our technique is adaptive to other scopes.

6 RELATED WORK

IPA is built on top of classical predicate analysis and change impact analysis. The related techniques are reviewed below.

Predicate Analysis

Predicate analysis and related techniques have been studied for a long time. Graf and Saidi [1997] firstly parameterized predicate abstraction by tracking values of a set of predicates in the abstract model. Clarke et al. [2000] proposed the counterexample-guided abstraction refinement framework. Henzinger et al. [2004] proposed to combine Craig interpolation with predicate abstraction. Their technique provides an efficient way of discovering new predicates from spurious counterexamples. In [Henzinger et al. 2002], the authors introduced a paradigm that refines abstract models on demand by allowing different abstract precisions for different program locations. The techniques above are implemented in CPAchecker. Block-abstraction memorization was proposed in [Wonisch and Wehrheim 2012] where predicate analysis can benefit from the cached information of previous block analysis. These techniques aim to optimize the predicate analysis to verify single program version. In contrast, *IPA* is designed for dealing with two programs, such as regression verification.

Change Impact Analysis

Change impact analysis [Arnold 1996] is the process of identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change. Many CIA techniques have recently been applied to the source code level [Gallagher and Lyle 1991; Ryder and Tip 2001]. The source code level CIA can be classified into three categories: static approach, dynamic approach, and online approach. The static approach extracts facts from source codes or documents to assess impacts of the change [Antoniol et al. 2000; Chaumun et al. 2002; Gallagher and Lyle 1991; Ryder and Tip 2001]. The dynamic and online approaches collect runtime information from program execution [Apiwattanapong et al. 2005; Breech et al. 2004]. The CIA technique that we implemented is a static code-level CIA. The result that we get by CIA algorithm is not the set of impacted statements but the affected variables for every program location.

Other methods used change impact analysis to invalidate summaries or guide incremental symbolic execution [Godefroid et al. 2011; Yang et al. 2014]. We utilize this technique to identify the reusable annotations of predicate analysis. To the best of our knowledge, ours is the first to apply CIA to predicate analysis-based regression verification.

Regression Verification

Regression verification is firstly introduced by [Godlin and Strichman 2009], where it mainly refers to the equivalence checking between revisions. Later, this terminology has been extended in kinds of literature (e.g., [Beyer et al. 2013; Fedyukovich et al. 2013; Visser et al. 2012; Yang et al. 2009]) for referring the procedure of re-verifying or re-analyzing the changed systems, where the techniques of regression verification are often based on the reuse of previously-computed intermediate results. As a result, there are two research lines for regression verification, i.e., equivalence checking and reusing intermediate results.

In the research line of regression verification as equivalence checking, the group of approaches take two programs as input and conduct equivalence check. The method for proving conditional equivalence of two programs by isolating and abstracting functions is proposed in [Chaki et al. 2012; Godlin and Strichman 2009]. Some approaches can check that a property keeps satisfied by programs changes. Lahiri et al. [2013] applied the equivalence check of two programs to detect if the properties that hold previously can also hold in new program version. Klebanov et al. [2018] studied

how equivalence proofs can be conducted effectively if the programs to be compared have a similar control flow. Instead of employing the equivalence check, our approach takes the light-weight CIA to approximate the potential changed intermediate verification result. And then we apply the reusable intermediate result to the new program version.

In the literature of regression verification as re-verifying or re-analyzing the changed systems, researchers proposed various information for reusing in regression verification. The type of reused information depends on the underlying verification techniques. In [Henzinger et al. 2003a; Lauterburg et al. 2008; Yang et al. 2009], state space graph is proposed to be reused in regression verification, which is usually computed by explicit model checking. However, the state-space graph may be too large to be efficiently storing and processing. In [Fedyukovich et al. 2013; Sery et al. 2012], the authors proposed to reuse Craig interpolation-based function summaries in regression verification. This technique is mainly suitable for bounded model checking, and the interpolation computation may incur considerable overheads. Rothenberg et al. [2018] proposed a regression verification technique that reuses the previously generated *Floyd-Hoare automata*. The underlying verification algorithm in [Rothenberg et al. 2018] is *trace abstraction*, which is quite different from the predicate analysis. We propose to reuse assertion annotations that can be easily obtained from the results of predicate analysis. We show that annotations are a good candidate of intermediate results for reuse in incremental predicate analysis.

There are also some techniques that are compatible with predicate analysis and thus can be combined with our techniques. In [Aquino et al. 2015; Jia et al. 2015; Visser et al. 2012], researchers proposed techniques for caching and reusing constraint solving results. Given that constraint solving is at the lower level than verification, these reusing techniques can be easily integrated (after necessary adaption) with our approach. The work [Beyer et al. 2013] proposed to reuse the abstract precision for CEGAR-based regression verification. The abstract precision at the last iteration of CEGAR for verifying the older program version is reused as the initial abstract precision in regression verification. Note that the abstract precision is high-level information of CEGAR. Precision reuse does not deal with the model checking process in each iteration of CEGAR. In contrast, our approach can handle the state space exploring process of predicate analysis. By reusing the previously-generated assertions, a large portion of abstract state space can be pruned.

Other relevant works are [Fedyukovich et al. 2014, 2016] where the safe inductive invariants were proposed for reuse. By adapting to the program changes, these invariants can form a valid safety proof for optimized programs. However, this technique can only be applied to compiler optimizations where the only permitted change is the reordering of program statements. In contrast, our technique allows any modifications to programs including adding, revising or deleting statements. This technique is not applicable to most programs in the benchmark that contain statement changes. It was thus not compared in the experiments.

7 CONCLUSION

This paper proposed an incremental predicate analysis technique by reusing assertion annotation. The assertion annotation for the previous revisions was proposed to be reused in the verification of new revisions. A light-weight impact-analysis technique was designed for analyzing reusability of assertions. A novel assertion strengthening technique was further developed. Soundness of our technique was formally established. The proposed approach has been realized in CPAchecker. Extensive experiments on real-world revisions of Linux drivers show promising performance of the approach. In the future, we plan to work out incremental techniques in other abstract domains. It would also be interesting to improve impact analysis to raise reusability rates without sacrificing too much efficiency.

ACKNOWLEDGMENTS

This work was partially funded by the NSF of China (No. 61672310), the National Key R&D Program of China (No. 2018YFB1308601), and the Guangdong Science and Technology Department (No. 2018B010107004).

REFERENCES

June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. 2012. Large-Scale Formal Verification in Practice: A Process Perspective. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, New York, NY, USA, 1002–1011. https://doi.org/10.1109/ICSE.2012.6227120

Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. 2000. Identifying the Starting Impact Set of a Maintenance Request: A Case Study. In *4th European Conference on Software Maintenance and Reengineering* (Zurich, Switzerland) *(CSMR 2000)*. IEEE Press, New York, NY, USA, 227–230. https://doi.org/10.1109/CSMR.2000.827331

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2005. Efficient and Precise Dynamic Impact Analysis Using Execute-after Sequences. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 432–441. https://doi.org/10.1145/1062455.1062534

Andrea Aquino, Francesco A. Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. 2015. Reusing Constraint Proofs in Program Analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 305–315. https://doi.org/10.1145/2771783.2771802

Robert S. Arnold. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press, Washington, DC, USA.

Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 203–213. https://doi.org/10.1145/378795.378846

Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 260–264. https://doi.org/10.1007/3-540-44585-4_25

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories* (Edinburgh, England), Vol. 13. 14.

Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Lugano, Switzerland) *(FMCAD '10)*. FMCAD Inc, Austin, Texas, 189–198. https://doi.org/10.5555/1998496.1998532

Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. 2013. Precision Reuse for Efficient Regression Verification. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 389–399. https://doi.org/10.1145/2491411.2491429

Dirk Beyer, Damien Zufferey, and Rupak Majumdar. 2008. CSIsat: Interpolation for LA+EUF. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 304–308. https://doi.org/10.1007/978-3-540-70545-1_29

Ben Breech, Anthony Danalis, Stacey A. Shindo, and Lori L. Pollock. 2004. Online Impact Analysis via Dynamic Compilation Technology. In *20th International Conference on Software Maintenance* (Chicago, IL, USA) *(ICSM '04)*. IEEE Press, New York, NY, USA, 453–457. https://doi.org/10.1109/ICSM.2004.1357834

Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. 2012. Regression Verification for Multi-threaded Programs. In *Verification, Model Checking, and Abstract Interpretation*. Springer, Berlin, Heidelberg, 119–135. https://doi.org/10.1007/978-3-642-27940-9_9

M.Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, and François Lustman. 2002. A change impact model for changeability assessment in object-oriented software systems. *Science of Computer Programming* 45, 2 (2002), 155 – 174. https://doi.org/10.1016/S0167-6423(02)00058-8

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software*. Springer, Berlin, Heidelberg, 248–254. https://doi.org/10.1007/978-3-642-31759-0_19

Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. 2008. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 397–412. https://doi.org/10.1007/978-3-540-78800-3_30

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 154–169. https://doi.org/10.1007/10722167_15

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Satyaki Das, David L. Dill, and Seungjoon Park. 1999. Experience with Predicate Abstraction. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 160–171. https://doi.org/10.1007/3-540-48683-6_16

V. D'Silva, D. Kroening, and G. Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2014. Incremental Verification of Compiler Optimizations. In *NASA Formal Methods*. Springer, Berlin, Heidelberg, 300–306. https://doi.org/10.1007/978-3-319-06200-6_25

Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property Directed Equivalence via Abstract Simulation. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 433–453. https://doi.org/10.1007/978-3-319-41540-6_24

Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. 2013. eVolCheck: Incremental Upgrade Checker for C. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 292–307. https://doi.org/10.1007/978-3-642-36742-7_21

Keith Brian Gallagher and James R. Lyle. 1991. Using program slicing in software maintenance. *IEEE transactions on software engineering* 17, 8 (1991), 751–761. https://doi.org/10.1.1.39.1532

Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. 2011. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Static Analysis*. Springer, Berlin, Heidelberg, 112–128. https://doi.org/10.1007/978-3-642-23702-7_12

Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proceedings of the 46th Annual Design Automation Conference* (San Francisco, California) *(DAC '09)*. Association for Computing Machinery, New York, NY, USA, 466–471. https://doi.org/10.1145/1629911.1630034

Susanne Graf and Hassen Saidi. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 72–83. https://doi.org/10.1007/3-540-63166-6_10

Fei He, Shu Mao, and Bow-Yaw Wang. 2016. Learning-Based Assume-Guarantee Regression Verification. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 310–328. https://doi.org/10.1007/978-3-319-41528-4_17

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from Proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 232–244. https://doi.org/10.1145/964001.964021

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. 2003a. *Extreme Model Checking*. Springer, Berlin, Heidelberg, 332–358. https://doi.org/10.1007/978-3-540-39910-0_16

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 58–70. https://doi.org/10.1145/503272.503279

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003b. Software Verification with BLAST. In *Model Checking Software*. Springer, Berlin, Heidelberg, 235–239. https://doi.org/10.1007/3-540-44829-2_17

Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/2771783.2771806

Alexey Khoroshilov, Vadim Mutilin, Alexander Petrenko, and Vladimir Zakharov. 2010. Establishing Linux Driver Verification Process. In *Perspectives of Systems Informatics*. Springer, Berlin, Heidelberg, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14

Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2018. Automating regression verification of pointer programs by predicate abstraction. *Formal methods in system design* 52, 3 (2018), 229–259. https://doi.org/10.1007/s10703-017-0293-8

Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 345–355. https://doi.org/10.1145/2491411.2491452

Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. 2008. Incremental State-Space Exploration for Programs with Dynamically Allocated Data. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.

1145/1368088.1368128

M. M. Lehman and L. A. Belady. 1985. *Program Evolution: Processes of Software Change.* Academic Press Professional, Inc., USA.

Manny M Lehman, Dewayne E Perry, and Juan F Ramil. 1998. Implications of evolution metrics on software maintenance. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (Bethesda, Maryland, USA) *(ICSM '98)*. IEEE Press, New York, NY, USA, 208–217. https://doi.org/10.1109/ICSM.1998.738510

Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. 1997. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium* (Albuquerque, NM, USA). IEEE Press, New York, NY, USA, 20–32. https://doi.org/10.1109/METRIC.1997.637156

Mikhail U Mandrykin, Vadim S Mutilin, EM Novikov, Alexey V Khoroshilov, and PE Shved. 2012. Using Linux device drivers for static verification tools benchmarking. *Programming and Computer Software* 38, 5 (2012), 245–256. https://doi.org/10.1134/S0361768812050039

Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 123–136. https://doi.org/10.1007/11817963_14

Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. 2002. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings 18th International Conference on Data Engineering* (San Jose, CA, USA). IEEE Press, New York, NY, USA, 117–128. https://doi.org/10.1109/ICDE.2002.994702

Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-03811-6

Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging Field Data for Impact Analysis and Regression Testing. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Helsinki, Finland) *(ESEC/FSE-11)*. Association for Computing Machinery, New York, NY, USA, 128–137. https://doi.org/10.1145/940071.940089

Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. 2018. Incremental Verification Using Trace Abstraction. In *Static Analysis*. Springer, Berlin, Heidelberg, 364–382. https://doi.org/10.1007/978-3-319-99725-4_22

Barbara G. Ryder and Frank Tip. 2001. Change Impact Analysis for Object-Oriented Programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01)*. Association for Computing Machinery, New York, NY, USA, 46–53. https://doi.org/10.1145/379605.379661

Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2012. Incremental upgrade checking by means of interpolation-based function summaries. In *Formal Methods in Computer-Aided Design (FMCAD), 2012* (Cambridge, UK). IEEE Press, New York, NY, USA, 114–121. http://ieeexplore.ieee.org/document/6462563/

Wladyslaw M Turski. 1996. Reference Model for Smooth Growth of Software Systems. *IEEE Trans. Softw. Eng.* 22, 8 (Aug. 1996), 1. https://doi.org/10.1109/32.536959

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/2393596.2393665

Daniel Wonisch and Heike Wehrheim. 2012. Predicate Analysis with Block-Abstraction Memoization. In *Formal Methods and Software Engineering*. Springer, Berlin, Heidelberg, 332–347. https://doi.org/10.1007/978-3-642-34281-3_24

Guowei Yang, Matthew B Dwyer, and Gregg Rothermel. 2009. Regression model checking. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE Press, New York, NY, USA, 115–124. https://doi.org/10.1109/ICSM.2009.5306334

Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 24, 1, Article 3 (Oct. 2014), 42 pages. https://doi.org/10.1145/2629536