RI 12003, 23 April 2012 Computer Science

IBM Research Report

Alternate and Learn: Finding witnesses without looking all over

Nishant Sinha, Nimit Singhania
IBM Research Lab, India

Satish Chandra, Manu Sridharan
IBM T. J. Watson Research Center, USA

IBM Research Division
Almaden - Austin - Beijing - Delhi – Bangalore - Haifa - T.J. Watson - Tokyo - Zurich

Alternate and Learn:
Finding witnesses without looking all over

Nishant Sinha[1], Nimit Singhania[1], Satish Chandra[2], and Manu Sridharan[2]

[1] IBM Research Labs, India
[2] IBM T. J. Watson Research Center, U.S.A.

Abstract. Most symbolic bug detection techniques perform search over the program control flow graph based on either forward symbolic execution or backward weakest preconditions computation. The complexity of determining interprocedural all-path feasibility makes it difficult for such analysis to judge upfront whether the behavior of a particular caller or callee procedure is relevant to a given property violation. Consequently, these methods analyze several program fragments irrelevant to the property, often repeatedly, before arriving at a goal location or an entrypoint, thus wasting resources and diminishing their scalability.
This paper presents a systematic and scalable technique for *focused* bug detection which, starting from the goal function, employs alternating backward and forward exploration on the program call graph to lazily infer a small *scope* of program fragments, sufficient to detect the bug or show its absence. The method learns *caller* and *callee* invariants for procedures from failed exploration attempts and uses them to direct future exploration towards a scope pertinent to the violation.

1 Introduction

Even though sophisticated static analysis methods for bug detection exist [6, 12, 18, 16], the scalability of these methods is restricted. This is somewhat surprising given that most bugs can be attributed to program behavior in a small set of program regions, i.e., a *small scope* [16, 11].

We believe that the common drawback of these methods is that they cannot *focus* on a small set of pertinent program regions that trigger the bug. Such focusing is not easy: a static analysis tool encounters plenty of code irrelevant to a particular bug, but such code is not obviously irrelevant before it is analyzed. Furthermore, the tool may repeatedly re-analyze such irrelevant code, thus wasting resources without finding a witness.

Consider a few examples illustrating the need of focusing. (a) Suppose a *goal* function with a potential null dereference makes a virtual call with 100 possible targets, none of which are relevant to the bug. Exploring all these targets is wasteful, and therefore it is necessary to restrain the forward search to only a subset of callees. (b) Alternatively, consider a goal function g invoked in a large number of call contexts (exponential in the depth of call graph, in the worst case). If the analysis begins from *main* procedure, it is likely that many irrelevant program fragments will be encountered and analyzed before reaching g. Therefore, a goal-driven backward search is necessary for focusing.

Based on above observations, we may conceive of a potentially effective technique that performs backward expansion from a goal function g in a *small scope* centered around g. Effective discovery of such a scope in practice is non-trivial: previous work [16] employed a strategy based on breadth-first expansion from the goal function, but this may be inefficient if callers or callees far away from the goal need to be explored.

In this paper, we propose a new focused method to perform inter-procedural analysis for detecting bugs. The strategy performs a systematic search around the goal function g with the aim of either inferring a small scope which can trigger the bug or, in some cases, proving the absence of it. Note that finding a witness path to an error location in g requires finding a feasible *call context* for g. This call context consists not only of a set of transitive (backward) callers of g, but also (forward) callees invoked by g on the path to the error function. Based on this observation, our method *alternates* between forward and backward exploration in the call graph to detect a violation and backtracks whenever it fails to find a feasible call context. During alternation, forward expansion takes priority over backward expansion. This is crucial because forward expansion proves infeasibility of the error at the current caller level, and avoids further backward expansion into irrelevant program fragments, thus discovering small program scopes in practice.

The alternating expansion method, despite being lazy, may revisit several irrelevant program regions (e.g., error-free call contexts), re-analyze them and perform wasteful backtracks. Such unfocused exploration clearly reduces the efficiency of the analyzer. Therefore, to improve focus, we propose to *learn*, on-demand from exploration failures, *caller/callee* invariants that over-approximate the caller/callee data values respectively. These invariants contain specific facts which induced the failure and help avoid similar failures later by not re-exploring irrelevant callers/callees.

The proposed method may be viewed as an instance of the general DPLL paradigm, *explore-fail-learn-backtrack*, applied directly to the program call graph representation instead of operating at a fine-grained inter-procedural control flow graph level [18]. Because there may be large number of call contexts to a particular procedure, the backward search tries to efficiently explore the set of call contexts in a depth-first manner, backtracks from failures, and exploits caller/callee invariants inferred from failures to prune future search. The forward expansion assists the backward search to infer early failures, akin to how theory propagation assists in finding conflicts during DPLL search.

In our preliminary experiments with industrial Java benchmarks, we found that alternating scope expansion is crucial to get some benchmarks to finish in a reasonable time. Learning reduced the number of call graph edges visited, but this reduction is not always able to compensate for the overhead of computing invariants.

The key contributions of the paper are as follows:

- A scalable bug detection method ALTER that performs alternating backward and forward search (Sec. 4) to lazily infer a small scope around the goal function, sufficient to detect a witness. A symbolic intra-procedural *local* summary for each procedure (Sec. 3) forms the basis of efficient inter-procedural alternating expansion.
- A systematic technique to learn a program scope pertinent for bug-detection by inferring caller and callee invariants for procedures from failed explorations (Section 5).
- An experimental evaluation (Sec. 6) that shows the effectiveness of our techniques.

2 Motivating Examples and Overview

2.1 Alternating Scope Expansion

Consider the program `App1` in Fig. 1: here, the goal function is `A.init`, where a potential null dereference may occur at line 11 because the class A's local field `this.srcs` (non-null) is shadowed by the local parameter variable `srcs`.

```
1  class A implements C {
2    List srcs;
3    A(List srcs, Rect b) {
4      init (srcs, b);
5    }
6    void init(List srcs, Rect b) {
7      this.srcs = new Vector ();
8      if (srcs != null) {
9        this.srcs.addAll (srcs);
10     }
11     if (srcs.size() != 0) {...}
12   }
13 }
14 class T extends A {
15   T(List srcs, Rect b) {
16     if (srcs.isEmpty()) return;
17     init(srcs, b);
18   }
19 }
```

```
1  class M extends A {
2    M(C src, C alpha) {
3      List srcs; Rect b;
4      srcs = makeList(src,alpha);
5      b = makeBounds(src,alpha);
6      super(srcs, b);
7    }
8    List makeList(C s1, C s2) {
9      List ret = new ArrayList (2);
10     ret.add(s1);
11     ret.add(s2);
12     return ret;
13   }
14 }
15 class N {
16   void foo(C src, C alpha) {
17       C m = new M(src,alpha);
18       ...
19   }
20 }
```

Fig. 1: `App1` example, based on a fragment of the `batik` open-source benchmark.

ALTER first computes the local error condition for the goal at line 11 in `A.init`: ϕ := $(srcs_{A.init} = null)$, where $srcs_{A.init}$ refers to the `srcs` parameter of `A.init` (the extra constraints arising from the conditional at line 8-10 are simplified away). Now, ALTER must examine callers of `A.init`, namely `T.T` and `A.A`. Carrying out a *backward expansion* for `T.T`, ALTER composes the local path condition for calling `A.init` inside `T.T`, with ϕ. This composition yields false, because the `srcs` parameter of `T.T` must be non-null for execution to pass line 16 of `T.T`. Next, ALTER carries out backward expansion to include `A.A`, and another backward expansion to include `M.M`, which is a caller of `A.A`. At this point, it carries out a *forward expansion* to bring `M.makelist` in scope. Now, the side effect summary of `M.makelist` can prove —the return value of `M.makelist` cannot be null— that the call context `M.M` → `A.A` → `A.init` cannot lead to error. Thus, ALTER is able to show the absence of null dereference in `A.init` by alternating backward/forward expansion starting from the goal location in `A.init`.

Focused Exploration. Note how ALTER performs a focused search by avoiding exploration of irrelevant program regions which are in the nearby scope, i.e., functions `makeBounds` in `M.M`, `isEmpty` in `T.T`, `addAll` in `A.init`, `add` in `M.makeList` and other callers of `M.M` and `T.T`. See Figure 2. The method names in bold are the only ones visited in this process. In particular, note how forward expansion of `M.makeList` ensures early backtrack and avoids further backward expansion from `M.M`. Without alternating forward and backward expansion, the analysis would expand backward to callers of `M.M`, such as `N.foo` and its callers in Figure 2, which are irrelevant for the goal.

2.2 Learning Pertinent Scopes

In Fig. 3, the function `bar` contains a potential null dereference if the parameter `c` is `null`; `bar` is called by `foo` at two sites, which in turn, is called by `runA` and `runB` with newly allocated objects. Let us denote the local parameter `c` of `foo` by c_{foo}, and of `bar` by c_{bar}. ALTER begins analysis by building the local *error condition* for `bar`, i.e.,

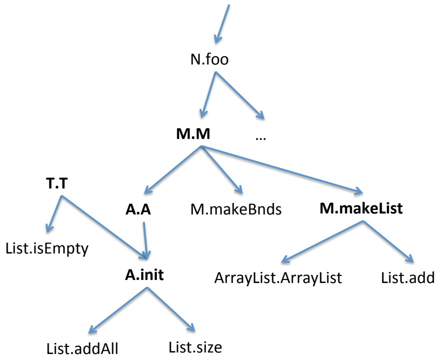

Fig. 2: Call Graph of `App1`.

```
1  class App2 {
2    void runA ()
3      {... foo(new A()); ...)
4    void runB ()
5      {... foo(new B()); ...}
6    //classes A, B extend class C
7    int foo (C c) {
8      if (*) return bar(c,1);
9      else return bar(c,2);
10   }
11   int bar (C c, int i) {
12     return c.compute(i);
13   }
14 }
```

Fig. 3: `App2` example.

$\phi := (c_{bar} = null)$, which is satisfiable if the parameter `c` gets the $null$ value under some call context to `bar`. To find such a context, ALTER performs backward search in a depth-first manner among callers of `bar`. The two call sites in `foo` for `bar` are analyzed individually; suppose the first call site foo_1 at line 8 is analyzed first. ALTER propagates ϕ backward, resulting in $\phi' := (c_{foo} = null)$. Here, c_{foo} is substituted by the actual called value, which is a heap-allocated object represented as $alloc(A)$; so, $\phi'' = (alloc(A) = null)$, which is unsatisfiable. Because the current context $runA \rightarrow foo_1 \rightarrow bar$ fails to find a witness, ALTER backtracks and tries the other caller `runB` for `foo`. Again, it fails, and backtracks further to try a different call site for `bar`: (site foo_2 at line 9). ALTER continues to try callers `runA` and `runB` again; however, no witness is found and the search terminates.

Focused Exploration. Note, however, that exploring `runA` and `runB` for the second call site foo_2 to `bar` in `foo` is redundant because we already know from exploring the first call site foo_1 that $(c_{foo} \neq null)$ for all callers to `foo` and hence no witness is possible via the callers of `foo`. A naive exploration technique may therefore explore the same callers redundantly without success because it does not *learn* from failed search attempts. The proposed algorithm therefore incorporates *learning* from failed exploration (Sec. 5): the learned information helps prune away the irrelevant program scope and focus search towards relevant regions.

3 Preliminaries and Intra-procedural analysis

We refer to the program statement with the violation, e.g., a null dereference, as the *goal location*. Also, the procedure having the goal location is called as the *goal procedure*. We say that a procedure f in an application is an *entrypoint* for the application if f is a public method. An entry point is *relevant* if it may call the goal procedure g transitively. Given a set of relevant entrypoints E, our analysis tries to find a (inter-procedural) feasible path, called *witness*, to the goal location from some entrypoint in E. We refer to such a path as a *global witness*. In contrast, any feasible path which terminates at the goal location but does not begin at a relevant entrypoint is said to be a *local witness*.

For a procedure f, the *input* (*output*) variables consist of the non-local variables and fields read (written) by some statement in f; the output variables also include two special variables ret and exc denoting the returned data and exception values from f respectively. A *symbolic state* $s = (\psi, \sigma)$ at a location l is a tuple consisting of a *reachability* predicate ψ and a map σ from scalar variables, fields and arrays to their symbolic values (terms). The predicate ψ represents the condition under which l can be reached via a given set of paths terminating at l. The map σ represents the symbolic values of variables obtained under the same set of paths. Both fields and arrays are modeled as mathematical maps from object references (integers) to their values. We do not distinguish between fields and arrays in our presentation; we use the term fields to refer to both. Loops are transformed to tail-recursive functions.

Local Summary for a Procedure. Classical inter-procedural program analysis [20, 19] intertwines procedure summary computation with summary composition: the (global) summary G_f for a function f is obtained after composing f's local behaviors with the summaries of all the callees of f. Such close coupling of summary computation and composition makes it hard to selectively explore the callees for a given goal location in f. For selective exploration, our approach decouples summary computation with composition: we analyze a procedure f in isolation and compute a *local* summary L_f for f *independent* of its callers and callees (referred to as the *environment* of f). The local summary L_f over-approximates the effect of both the callers and the callees of f and has two benefits: (a) we need not re-analyze f for different call contexts, and (b) we can utilize summaries from the environment of f to improve the precision of L_f in a lazy, goal-driven manner, and obtain G_f in the limit. To analyze f independent of its callees, we resort to *structural abstraction* [23, 1]: all outputs of each potential callee g of f are modeled using fresh symbolic variables (Skolem constants) denoting arbitrary values that the call to g may return. These Skolem constants (skolems, in short) over-approximate the output values of g and hence allow us to conservatively incorporate g's behavior in the summary of f.

Formally, the local summary L_f consists of three components: a *side effect* summary, a set of *call site* summaries and a set of *error conditions* (ECs). The *side effect* summary of f is a map from the outputs of f to their symbolic values in terms of inputs of f and captures the data flow from inputs to outputs along all possible paths of f. Let en_f denote the entry location of f. For each call site f_j in f, we compute a *call site* summary at f_j denoted by a symbolic state $s = (\psi, \sigma)$, where ψ denotes the all-path reachability condition of f_j from en_f and the state σ contains the symbolic values of variables and fields obtained along each path to f_j and expressed in terms of inputs of f. Finally, for each goal location l in f, the error condition (EC) predicate ϕ is obtained by conjoining the all-paths reachability condition from en_f to l with the violation condition, e.g., the null dereference predicate $(v = null)$ for a variable v.

If f has no callees or all the callees are inlined into f, then all the components of L_f are precise, i.e., L_f contains the precise symbolic values of outputs along each path and precise reachability conditions for each error location from en_f. However, if we employ structural abstraction to decouple the callees of f, L_f becomes over-approximate. In particular, an EC ϕ may now contain skolems and satisfiability of ϕ no longer implies that an actual local witness to l exists. Note that ϕ may also contain input variables to f and hence a local witness may not extend to any global witness. Both these sources of imprecision in L_f are removed on-demand during the inter-procedural exploration phase (cf. Sec. 4) for finding a global witness.

Summary Computation. We compute the summary for a function f by a forward all-path analysis algorithm which propagates the symbolic state along all paths of f

precisely starting from en_f. We use program expressions to represent symbolic states precisely and propagate states by employing precise transformers for each statement in f (structural abstraction is applied at each call location). To avoid path explosion as well as maintain precision, the algorithm merges symbolic states at join nodes by guarding the incoming symbolic value along each edge by the corresponding path condition and representing the merged state using an *if-then-else* (ite) term compactly. The details of Java statement transformers can be found in [4] and merge operation in [13, 21] and are omitted in the interest of space. During propagation, we compute the ECs at each goal location, the call site summaries at each call location and the side effect summary at the exit location of f.

```
int p(int x){
  if(x < 10)
    error();
  return x - 10;
}

int q(int y){
  if(y > 6){
    int z = t(y);     (1)
    int a = p(z);     (2)
    int b = r(y, z); (3)
    return (a + b);
  }
  return 0;
}
```

```
int r(int u, int v){
  if(u > v)
    return p(u);  (1)
  else
    return p(v);  (2)
}

int s(int c){
  return r(c, 10);(1)
}

int t(int d){
  return d * 2;
}
```

Fig. 4: Program P.

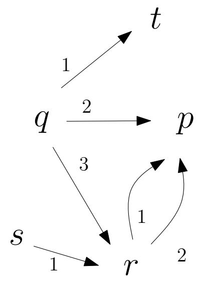

Fig. 5: Call Graph of Program P

Example. Consider program P in Fig. 4. The summary for the return value of r is $ite((u > v), sk_1^p, sk_2^p)$ where $u(v)$ is the initial value for parameter u(v) in r and $sk_1^p(sk_2^p)$ is the return value of p at call site $r_1(r_2)$. The call site summary for call site r_1 in r is (ψ, σ), where, $\psi := (u > v)$ and $\sigma := [\mathtt{u} \to u, \mathtt{v} \to v]$. Then, the error condition ϕ for the violation (function $error()$ in p) in p is $\phi := (x < 10)$.

4 Backward, Forward and Alternating Expansions

Recall that an EC ϕ local to f is imprecise because it contains inputs of f and skolems from callees of f, both of which are unconstrained. Hence even if ϕ is satisfiable, neither a local nor a global witness may exist. To search for a global witness, we perform interprocedural analysis by expanding the scope around the goal function iteratively using a combination of *forward* and *backward* expansion. Forward expansion replaces skolems in ϕ with the actual return values of callees while backward expansion substitutes the inputs with the actual input values for a calling context of f.

Backward. Consider an EC ϕ at entry of a procedure f such that the satisfiability of ϕ implies a local witness to the goal location from f. To propagate back ϕ into a particular caller h of f at site h_k, we use the call site summary (ψ, σ) at h_k. This summary allows us to express the inputs in ϕ directly in terms of inputs of h without reanalyzing h. For every input i in ϕ, let $Val(i, h_k)$ denote the value of i in the symbolic state σ before the call at h_k. Backward propagation is achieved by computing $\phi' := (\phi \wedge CC(h_k))$:

$$CC(h_k) := (\bigwedge_{i \in in_\phi} (i = Val(i, h_k)) \wedge \psi)$$

where $CC(h_k)$ consists of constraints expressing each input i in set of inputs in_ϕ of ϕ, in terms of actual symbolic values at the call site and the all-path reachability condition ψ from entry of h to h_k[3]. The procedure EXPANDBWD$(h_k \to f, \phi)$ computes $CC(h_k)$.

Forward. Suppose we want to expand a skolem sk at a call site f_j in f, where sk corresponds to an output variable, say ret, in a callee g. We first obtain the summary expression sum_{ret} for ret from the side-effect summary of g and then substitute the inputs in sum_{ret} with the actual values obtained from the call site summary at f_j. More precisely, the forward expansion constraint for sk is $SC(sk) := SC_1(sk) \wedge SC_2(sk)$. Here, $SC_1(sk)$ contains the summary expression, i.e., $SC_1(sk) := (sk = sum_{ret})$. Note that sum_{ret} depends on the set of inputs In of g and skolems Sk corresponding to callees of g. So, we *raise* the inputs In to the caller by using the call site values $Val(i, f_j)$ (defined above) from call site f_j, i.e., $SC_2(sk) := \bigwedge_{i \in In}(i = Val(i, f_j))$. In sum_{ret}, we also replace each $sk \in Sk$ by a fresh value sk' using a *contextualization* scheme which records the fact that sk' corresponds to the call from f_j to g. The scheme is discussed in Appendix. Note that sk' may be expanded forward in a similar way as sk. Let the procedure EXPANDFWD(f, ϕ) compute the skolem constraints SC (recursively, if required) for the set of skolems Sk' in ϕ, i.e., $SC := \bigwedge_{sk \in Sk'} SC(sk)$.

Example. In Fig. 4, the initial EC in `p` is $\phi_1 := (x < 10)$. Suppose, we need to propagate EC ϕ_1 back to caller `q` at call site q_2. We start by computing $CC(q_2) := (y > 6 \wedge x = sk^t)$ where sk^t is skolem for call to `t` at site q_1. On *backward* propagation and simplification, EC becomes $\phi_2 := (\phi_1 \wedge CC(q_2)) \equiv (sk^t < 10 \wedge y > 6)$. Now, we expand *forward* the skolem sk^t in ϕ_2 using $SC := (sk^t = d * 2) \wedge (d = y)$. Finally, the EC is $\phi_3 := \phi_2 \wedge SC \equiv (y * 2 < 10 \wedge y > 6)$, which is unsatisfiable.

In practice, instead of conjoining constraints, we substitute the actual values for inputs and summaries for skolems in the error condition ϕ. This assists simplification before invoking a constraint solver to check for satisfiability of ϕ. Note that iterative forward or backward expansion may not terminate due to recursive function calls. Therefore, we impose fixed bounds to terminate expansion under recursion. Similarly, we cannot expand a skolem (expand backward) if the source code of the corresponding callee (caller) is not available to the analyzer.

4.1 Alternating Expansion

Alg. 1 describes the alternating expansion algorithm. The main procedure ALTER takes the goal function g and an EC ϕ from summary of g as input and performs a backtracking based search over the program call graph. In a particular iteration with EC ϕ local to function f, ALTER proceeds as follows. First, ALTER expands the skolems in ϕ using EXPANDFWD to obtain the corresponding summary constraints SC. If $\phi \wedge SC$ is satisfiable, ALTER expands all the callers of f (CALLERS(f)) using EXPANDBWD in a depth-first manner iteratively. Given a caller h with call site h_k, EXPANDBWD returns the call context constraints $CC(h_k)$ for h_k, which express the inputs of f in terms of inputs of h. ALTER then recursively proceeds to analyze h with the new error condi-

[3] Similar to [4], we also add constraints for handling virtual calls; described in Appendix

tion $\phi' := (\phi \wedge SC \wedge CC(h_k))$ obtained by conjoining both forward and backward expansion constraints with the previous ϕ.

If the EC ϕ at any moment during alternating expansion is infeasible (UNSAT), it indicates an *exploration failure*, i.e., no further backward/forward search will yield a global witness. In this case, ALTER *backtracks* to the previous callee c on the recursion stack and pursues the next caller of c for backward expansion. Backtracking may occur on obtaining infeasibility after either (a) forward expansion (on conjoining with SC) or (b) backward expansion (on conjoining with $CC(h_k)$). As we will see in Sec. 5, ALTER *learns* facts responsible for the current failure and uses them to avoid similar failures during future exploration.

ALTER may terminate with either (a) witness (WIT) or (b) no witness (NOWIT) or (c) an inconclusive (UNKNOWN) result. During backward exploration, if ALTER encounters an entrypoint procedure (ENTRYPOINT(f)) and the current ϕ is feasible, then a potential witness exists. If ϕ is skolem-free, ALTER concludes that a witness exists and returns the corresponding call context. Otherwise, ϕ may still contain skolems which cannot be expanded further, e.g., due to recursion bounds. Consequently, there may exist skipped callees which affect the feasibility of ϕ, thus making the witness spurious. In this case, ALTER returns an UNKNOWN value. Finally, if ALTER finishes exploring all callers without finding an actual witness or an UNKNOWN result, then ALTER concludes that no witness to the goal location exists. Note that obtaining an UNKNOWN value for some call context does not imply that the search is inconclusive; ALTER may go on to find an actual witness along a different call context. However, ALTER cannot infer no-witness if an UNKNOWN value is obtained for some context during exploration.

```
ALTER($f$, $\phi$)
if $\phi$ is UNSAT then
    return NOWIT
/* Forward Expansion */
$SC$ := EXPANDFWD($f$, $\phi$)
if ($\phi \wedge SC$) is UNSAT then
    return NOWIT
if ENTRYPOINT($f$) then
    if ($\phi \wedge SC$) has no skolems then
        return (WIT, $nil$)
    else
        return UNKNOWN
$inconcl$ := false
foreach $h_k \in$ CALLERS($f$) do
    /* Backward Expansion */
    $CC(h_k)$ := EXPANDBWD($h_k \rightarrow f$, $\phi \wedge SC$)
    $ans$ := ALTER($h$, $\phi \wedge SC \wedge CC(h_k)$)
    if $ans$ = (WIT, $l$) then
        return (WIT, $[h_k, l]$)
    if $ans$ = UNKNOWN then
        $inconcl$ := true
if $inconcl$ then
    return UNKNOWN
return NOWIT
```

Algorithm 1: Alternating Expansion Algorithm for Bug Detection

Example. Let us see how ALTER analyzes the program `App1` in Fig. 1. The goal function is `A.init`, where a potential null dereference may occur at line 11 because the class A's local field `this.srcs` (non-null) is shadowed by the local parameter variable `srcs`. `A.init` has two callers: `T.T` and `A.A` where `A.A` is, in turn, called by `M.M`.

1. First ALTER computes a local EC ϕ for `A.init`. This $\phi := \phi_1 \wedge \phi_2$ where $\phi_1 := ((srcs_{A.init} \neq null) \wedge (this.srcs \neq null)) \vee (srcs_{A.init} = null)$ and $\phi_2 := (srcs_{A.init} = null)$ and $srcs_{A.init}$ refers to the value of parameter `srcs` of `A.init`.

On simplifying ϕ_1 with ϕ_2, we get $\phi := (srcs_{A.init} = null)$. Because ϕ does not contain any skolems, ALTER proceeds with backward expansion along some caller, say `T.T`.

2. ALTER computes the local summary for `T.T` and employs the call site component, (ψ, σ) for backward expansion, where the reachability condition $\psi := (srcs_{T.T} \neq null \wedge \neg sk^{ie})$ and value map σ = (`srcs` $\rightarrow srcs_{T.T}$), where $srcs_{T.T}$ refers to the value of parameter `srcs` in `T.T` and sk^{ie} corresponds to return value of `isEmpty` function. In ψ, $(srcs_{T.T} \neq null)$ appears because otherwise the previous call to `isEmpty` will throw an exception. After expansion, we obtain $\phi := (\psi \wedge (srcs_{T.T} = null))$, which simplifies to $false$, implying search failure along `T.T`. ALTER now backtracks to try the next caller `A.A` for `A.init`.
3. For `A.A`, the call site summary is $(true, \sigma')$ where σ' := (`srcs` $\rightarrow srcs_{A.A}$, `b` $\rightarrow b_{A.A}$). On propagation, $\phi := (srcs_{A.A} = null)$, which remains satisfiable. So, ALTER expands further backwards along caller `M.M`.
4. The call site summary for `M.M` is $(true, \sigma'')$ where σ'' := (`srcs` $\rightarrow sk^{ml}$, `b` $\rightarrow sk^{mb}$) where sk^{ml} and sk^{mb} denote the skolems corresponding to the return values of calls to `makeList` and `makeBounds`. Now, $\phi := (sk^{ml} = null)$, which leads ALTER to perform forward expansion to compute the return value of `makeList`.
5. The side-effect summary for `makeList` is computed next: the summary value for the returned variable (SC_1) is $ret^{ml} := alloc(ArrayList, 2)$. Because $sk^{ml} = ret^{ml}$, we get $\phi := (alloc(ArrayList, 2) = null)$ which again simplifies to $false$.

Thus, ALTER is able to show the absence of null dereference in `A.init` by a combination of backward and forward expansion starting from the goal location in `A.init`. Note how it avoids exploration of irrelevant program regions which are in the nearby scope, i.e., functions `makeBounds` in `M.M`, `addAll` in `A.init`, `isEmpty` in `T.T`, `add` in `M.makeList` and other callers of `M.M` and `T.T`. Also, note how forward expansion of `M.makeList` ensures early backtrack and avoids further backward expansion from `M.M`. The following theorem proves the correctness of ALTER.

Theorem 1. *Given a goal location l, (a) if* ALTER *returns a witness (*WIT*) result then there must exist a global witness for l, and (b) if* ALTER *returns no-witness (*NOWIT*) then no global witness exists.*[4]

5 Learning for Efficient Expansion

Naïve alternating expansion (Sec. 4.1) may perform redundant analysis by revisiting the same callers and callees and fail repeatedly. We now present an improved ALTER algorithm for efficient exploration based on learning *caller* and *callee* invariants and employing them to prune future search. The *caller invariant* $\Omega(f)$ for a procedure f over-approximates the incoming data values from the callers of f, while the *callee invariant* $\Theta(f)$ over-approximates the return values (side-effects, in general) of the callees in f. Both these invariants are learned from expansion failures, i.e., when the constraints added due to an expansion lead to infeasibility of the error condition. Alg. 2 shows the ALTER algorithm combined with failure-driven learning of *caller* and *callee* invariants.

ALTER initializes $\Omega(f)$ and $\Theta(f)$ for all procedures f to $true$ and strengthens them during exploration iteratively. The caller invariant $\Omega(f)$ is computed as disjunction of

[4] .

```
INITIALLY,
$\forall(h_k \to f), \omega(h_k \to f) := true$
$\forall f, \Omega(f) := true, \Theta(f) := true$

ALWAYS,
$\Omega(f) := \bigvee(\omega(h_k \to f) \mid h_k \in \text{CALLERS}(f))$

LEARN$\omega(h_k \to f, a, b)$
begin
    $I$ := INTERPOLANT $(a, b)$
    $\omega(h_k \to f) := \omega(h_k \to f) \wedge I$

LEARN$\Theta(f, a, b)$
begin
    $I$ := INTERPOLANT $(a, b)$
    $\Theta(f) := \Theta(f) \wedge I$

ALTER$(f, \phi)$
[S] if $\phi$ is UNSAT then
    return (NOWIT, $true$)

[C1] if $\phi \wedge \Theta(f) \wedge \Omega(f)$ is UNSAT then
    return (NOWIT, $\Theta(f) \wedge \Omega(f)$)

/* Forward Expansion */
$SC$ := EXPANDFWD$(f, \phi)$

[F1] if $\phi \wedge SC \wedge \Omega(f)$ is UNSAT goto [L1]
if ENTRYPOINT$(f)$ then
    if $(\phi \wedge SC)$ has no skolems then
        return (WIT, $nil$)
    else
        return UNKNOWN

$inconcl := false$
foreach $h_k \in$ CALLERS$(f)$ do
    [C2] if $\phi \wedge SC \wedge \omega(h_k \to f)$ = UNSAT then
        continue
    /* Backward Expansion */
    $CC(h_k)$ := EXPANDBWD$(h_k \to f, \phi \wedge SC)$
    $ans$ := ALTER$(h, \phi \wedge SC \wedge CC(h_k))$
    if $ans$ = (WIT, $l$) then
        return (WIT, $[h_k, l]$)
    if $ans$ = UNKNOWN then
        $inconcl := true$
    [F2] if $ans$ = (NOWIT, $Inv_h$) then
        [L2] LEARN$\omega(h_k \to f, CC(h_k) \wedge Inv_h,$
        $\phi \wedge SC)$

if $inconcl$ then
    return UNKNOWN
[L1] LEARN$\Theta(f, SC, \phi \wedge \Omega(f))$
[E] return (NOWIT, $\Theta(f) \wedge \Omega(f)$)
```

Algorithm 2: ALTER with learning caller Ω and callee Θ invariants.

call edge invariants (ω) which label each incoming call edge to f. When backward expansion from f to a caller h at a call site h_k fails, i.e., ans = (NOWIT, Inv_h) at location **F2** in Alg. 2, then ALTER learns a call edge invariant ω (**L2**) along the edge $h_k \to f$ using the procedure LEARNω. To this end, it splits the EC into caller- and callee-specific parts, A and B respectively, where $A \wedge B$ is infeasible. The caller-specific part, A consists of call context constraints $CC(h_k)$ and invariants Inv_h of h (usually, $\Omega(h) \wedge \Theta(h)$) which cause infeasibility. The callee-specific part, B consists of the original ϕ in f together with forward constraints SC. Note that A and B only share the input variables of f. LEARNω now computes an *interpolant* I of A and B over the common variables of A and B such that $A \Rightarrow I$ and $I \wedge B$ is infeasible. I.e., I is an expression over input variables of f such that it over-approximates the caller constraints and is still infeasible with the error condition in f. LEARNω now strengthens $\omega(h_k \to f)$ with I by conjoining I with the previous value of $\omega(h_k \to f)$. Then, ALTER backtracks and explores a different caller of f. Note that $\Omega(f)$ is updated when any of the call edge invariants change.

Similarly, ALTER computes (and updates) the callee invariant for f using LEARNΘ when forward expansion of ϕ from f fails (**F1**). In this case, the constraints are partitioned (**L1**) again into callee-specific (SC) and caller-specific ($\phi \wedge \Omega(f)$) parts, and an interpolant I of the two formulae is computed which over-approximates SC. The callee invariant $\Theta(f)$ is then strengthened by conjoining it with the new invariant I.

Note how both Ω and Θ are employed during exploration. Before forward expansion at location **C1**, ALTER first checks the current ϕ against the conjunction of both the invariants of f. Note that the invariants over-approximate the values from callers and callees of f. Hence, if the check with invariants is infeasible, no witness is possible on further expansion, and ALTER backtracks with NOWIT. Similarly, before backward expansion along $h_k \to f$ at location **C2**, ALTER checks ϕ against call edge invariants $\omega(h_k \to f)$, and backtracks if the check is infeasible. Lemma 1 and Theorem 2 prove the correctness of caller/callee invariants computed by Alg. 2.

Lemma 1. *The following invariants hold in Alg. 2. (a)* $(\Omega(h) \wedge \Theta(h)) \Rightarrow Inv_h$ *(b)* *(*$CC(h_k) \wedge Inv_h \wedge \phi \wedge SC$*) is unsatisfiable at* **L2***,* $SC \wedge \phi \wedge \Omega(f)$ *is unsatisfiable at* **L1***. (c)* $(\Omega(h) \wedge \Theta(h) \wedge CC(h_k)) \Rightarrow \omega(h_k \to f)$

Theorem 2. *Given a procedure* f*, (a) the caller invariant* $\Omega(f)$ *over-approximates the incoming data values from all the callers of* f *and (b) the callee invariant* $\Theta(f)$ *over-approximates the side-effects of the callees of* f*.*

Proofs of Non-Violation. If the analysis returns NOWIT, then the set of caller and callee invariants constitute a proof for absence of violation in the goal function g. In other words, we can conclude that null dereference is not possible at the goal location by using the caller $\Omega(g)$ and callee $\Theta(g)$ invariants for g. These invariants are obtained, in turn, from the invariants of other functions in the scope of the analysis. The undecidability of program analysis implies we cannot always obtain such a proof; however, in practice, we obtain proofs for absence of null dereference in several of our benchmarks. Note that the learned facts can be reused to improve search when checking multiple goals in the same application (cf. Sec. 6). Further, they are useful for re-validation across upgrades of an application; we leave investigating the usefulness of learned facts during incremental verification to a future work.

5.1 Examples illustrating the Learning Algorithm

Example 1. Consider the program and its call graph in Fig. 4. Suppose the functions `q` and `s` are the entry points and the call to $error()$ in `p` is a null dereference. Fig. 6 shows the ECs and invariants computed by ALTER on this program, starting with $true$ for all caller and callee invariants. The initial EC is $\phi_0 := (x < 10)$ in `p`.

1. ALTER first propagates ϕ_0 to caller `q` at site q_2 to get ϕ_1 (cf. Fig. 6(a)). Then, it expands forward sk^t in ϕ_1 to obtain ϕ_2, which is infeasible. ALTER learns the callee invariant $\Theta(q)$ from this failure (location **[L1]** in Algo. 2): it splits ϕ_2 into $A := SC_q \equiv (sk^t = y * 2)$ and $B := \phi_1 \wedge \Omega(q) \equiv (y > 6 \wedge sk^t < 10) \wedge (true)$ and computes interpolant $\Theta_0 = (sk^t \geq y * 2)$ (cf. Fig. 6 (b)). Then, it updates $\Theta(q) := \Theta_0$ and backtracks to `p`.

2. In `p`, ALTER now continues to learn a call edge invariant $\omega(q_2 \to p)$ (**[L2]** in Alg. 2) based on the previous failure. It partitions ϕ_2 into $A := \Omega(q) \wedge \Theta(q) \wedge CC(q_2) \equiv (true) \wedge (sk^t \geq y * 2) \wedge (y > 6 \wedge x = sk^t)$ and $B := \phi_0$, computes interpolant $\omega_1 := (x \geq 14)$ and updates $\omega(q_2 \to p) := \omega_1$ (Fig. 6 (b)). Now, ALTER propagates ϕ_0 back to next caller `r` of `p` at call site r_1 as ϕ_3 and then to `s` at s_1 as ϕ_4. Here, ϕ_4 is infeasible. Thus, ALTER backtracks to `r` and learns $\omega(s_1 \to r) = (v \geq 10)$.

3. Now, it propagates ϕ_3 to `q` from `r` and obtains ϕ_5 which is satisfiable. However, when ϕ_5 is conjoined with $\Theta(q)$, it becomes infeasible **[C1]**. Therefore, ALTER uses $\Theta(q)$ learned from previous failure in `q` to backtrack to `r` and avoid multiple forward

ϕ_0	INITIAL EC	$(x < 10)$	SAT	
ϕ_1	EXPANDBWD(ϕ_0, q_2)	$(y > 6 \wedge sk^t < 10)$	SAT	
ϕ_2	EXPANDFWD(ϕ_1, q)	$(y > 6 \wedge y * 2 < 10)$	UNSAT	$\varTheta_0, \omega_1$
ϕ_3	EXPANDBWD(ϕ_0, r_1)	$(u < 10 \wedge u > v)$	SAT	
ϕ_4	EXPANDBWD(ϕ_3, s_1)	$(c < 10 \wedge c > 10)$	UNSAT	ω_2
ϕ_5	EXPANDBWD(ϕ_3, q_3)	$(y > 6 \wedge y < 10 \wedge y > sk^t)$	SAT	
ϕ_6	CHK$(\phi_5, \varTheta(q))$	$(y > 6 \wedge y < 10 \wedge y > sk^t) \wedge (sk^t \geq y * 2)$	UNSAT	ω_3, ω_4
ϕ_7	EXPANDBWD(ϕ_0, r_2)	$(v < 10 \wedge u \leq v)$	SAT	
ϕ_8	CHK$(\phi_7, \varOmega(r))$	$(v < 10 \wedge u \leq v) \wedge (u \leq v - 7 \vee v \geq 10)$	SAT	
ϕ_9	CHK$(\phi_7, \omega(s_1 \rightarrow r))$	$(v < 10 \wedge u \leq v) \wedge (v \geq 10)$	UNSAT	-
ϕ_{10}	CHK$(\phi_7, \omega(q_3 \rightarrow r))$	$(v < 10 \wedge u \leq v) \wedge (u \leq v - 7)$	SAT	
ϕ_{11}	EXPANDBWD(ϕ_7, q_3)	$(y > 6 \wedge sk^t < 10 \wedge y \leq sk^t)$	SAT	
ϕ_{12}	CHK$(\phi_{11}, \varTheta(q))$	$(y > 6 \wedge sk^t < 10 \wedge y \leq sk^t) \wedge (sk^t \geq y * 2)$	UNSAT	ω_5, ω_6

(a)

	INV	A	B	INTERPOLANT
$\varTheta_0$	$\varTheta(q)$	$(sk^t = y * 2)$	$(y > 6 \wedge sk^t < 10)$	$sk^t \geq y * 2$
ω_1	$\omega(q_2 \rightarrow p)$	$(sk^t \geq y * 2) \wedge (y > 6 \wedge x = sk^t)$	$(x < 10)$	$x \geq 14$
ω_2	$\omega(s_1 \rightarrow r)$	$(u = c \wedge v = 10)$	$(u < 10 \wedge u > v)$	$v \geq 10$
ω_3	$\omega(q_3 \rightarrow r)$	$(sk^t \geq y * 2) \wedge (y > 6 \wedge u = y \wedge v = sk^t)$	$(u < 10 \wedge u > v)$	$u \leq v - 7$
ω_4	$\omega(r_1 \rightarrow p)$	$(u \leq v - 7 \vee v \geq 10) \wedge (u > v \wedge x = u)$	$(x < 10)$	$x \geq 11$
ω_5	$\omega(q_3 \rightarrow r)$	$(sk^t \geq y * 2) \wedge (y > 6 \wedge u = y \wedge v = sk^t)$	$(v < 10 \wedge u \leq v)$	$v \geq 14$
ω_6	$\omega(r_2 \rightarrow p)$	$((u \leq v - 7 \wedge v \geq 14) \vee (v \geq 10)) \wedge (u \leq v \wedge x = v)$	$(x < 10)$	$x \geq 10$

(b)

Fig. 6: Illustration of the Learning Algorithm for Program P in Fig. 4

expansions of t in q. On backtracking, it learns $\omega(q_3 \rightarrow r) := (u \leq v - 7)$ and updates $\varOmega(r) := \omega_2 \vee \omega_3 \equiv (u \leq v - 7) \vee (v \geq 10)$.

4. As ALTER failed on all callers of r, it backtracks to p and learns $\omega(r_1 \rightarrow p) := \omega_4 \equiv (x \geq 11)$. ALTER now tries the next caller r_2 of p to obtain ϕ_7, which is feasible. Next, all callers of r are tried: ALTER first checks ϕ_7 against current call edge invariant value $\omega(s_1 \rightarrow r)$, which is infeasible; it next tries $\omega(q_3 \rightarrow r)$, which is feasible. So, ϕ_7 propagates back to q as ϕ_{11}. In q, however, ϕ_{11} becomes unsatisfiable with $\varTheta(q)$, forcing backtrack to r while updating $\omega(q_3 \rightarrow r) := \omega_3 \wedge \omega_5$ and $\varOmega(r) := \omega_2 \vee (\omega_3 \wedge \omega_5)$. Because all callers of r are explored, ALTER further backtracks to p while updating $\omega(r_2 \rightarrow p) := \omega_6$. Finally, no feasible paths to error in p exist; ALTER returns NOWIT.

Example 2. Recall the example in Fig. 3 where ALTER redundantly explores callers runA and runB multiple times. Learning solves this problem: after failing with context $runA \rightarrow foo_1 \rightarrow bar$, ALTER labels edge $runA \rightarrow foo$ with predicate $\omega_1 = (c_{foo} \neq null)$. Similarly, edge $runB \rightarrow foo$ is labeled with ω_1. Because both callers of foo have been explored, ALTER now computes a call invariant $\varOmega_1 = (c_{foo} \neq null)$ for foo by disjoining the incoming edge invariants. This invariant helps to prune backward search in the second iteration: the EC $(c_{foo} = null)$ for context $foo_2 \rightarrow bar$ is unsatisfiable immediately on conjoining with $\varOmega_1$. Hence, ALTER avoids the redundant exploration of runA and runB for the second call to bar.

6 Evaluation

We implemented the ALTER algorithm using the WALA framework for analyzing Java programs and applied it to validate the null dereference warnings produced by FindBugs [10], in a manner similar to the earlier Snugglebug work [4], where these benchmarks were validated using weakest precondition computation. We considered three open-source Java benchmarks, *apache-ant*(v1.7), *batik*(v1.6) and *tomcat*(v6.0.16), having LoC 88k, 157k and 163k, respectively.

Our analysis finds global witnesses with respect to a set of given entrypoints; we initialized the set of entrypoints to all public methods without any callers. Procedure summarization is done on-demand during forward/backward expansion. We used the CVC3 solver [2] to check the satisfiability of ECs and the MathSAT5 solver [7] to compute interpolants. A coarse mod-ref analysis is performed on the call graph in the beginning to compute side-effects. Extensive formula simplification is performed in ALTER using a pre-defined set of rewrite rules [4]. Forward expansion involves recursive expansion of skolems as the predominant strategy, with feedback driven expansion for virtual call skolems [4] (cf. Appendix). We also tried lazy expansion strategies [1]; however, recursive forward expansion outperforms lazy expansion in most cases.

We designed a set of experiments: First, we compare ALTER with a non-alternating version NOALT which performs forward expansion only after backward expansion terminates at an entrypoint. Next, we evaluate the impact of learning. Finally, since we consider Snugglebug (SB) to be an ancestor of ALTER (they do share significant amount of code), we also compare the end-to-end performance of ALTER with SB.

Fig. 7 shows the ALTER results on a set of dereference checks for above benchmarks (each check corresponds to a single warning reported by FindBugs). All the benchmarks contain a combination of witness and no-witness instances. [5] We show only the actual analysis run times; the initial call graph and mod-ref computation times are excluded. The results also show the number of functions summarized by ALTER and the maximum error depth for the checks: the alternating expansion by ALTER succeeds in finding a witness or showing its absence by analyzing a small set of functions around the goal.

ALTER outperforms NOALT on most benchmarks: although NOALT performs similar to ALTER for bugs where entrypoints are closer to the goal function, it times-out on deeper goal functions. For example, NOALT performs poorly on `tomcat14` because it redundantly explores a much longer call context that does not lead to an error, and wastes resources performing many redundant forward expansions. In contrast, ALTER finds a call context of depth 6 that leads to a witness. This shows the advantage of alternating expansion clearly: expanding forward before backward avoids exploring long redundant contexts and helps obtain smaller scopes on our benchmarks.

Fig. 8 shows the impact of learning on alternating expansion, both in terms of the run-time and the edges explored during backward expansion: our experiments primarily focused on learning and reusing caller invariants. Instead of analyzing a single goal, we collect multiple null dereferences from the goal function and analyze them in sequence. This allows the successive runs to take advantage of previously learned invariants. The results show that learning invariants indeed reduces the number of call graph edges re-explored (Edge(L) vs Edge(NL)) by reusing invariants learned earlier. In some cases, e.g., `tomcat10`, the number of edges explored reduces by almost two-thirds. In contrast, the run-time benefits depend on how effectively the invariants are reused: if there is plenty of reuse, the ALTER run-time is lower. However, if the overhead of computing invariants is much larger than the reduction due to reuse, ALTER is slower with learning. For example, although ALTER explores much fewer edges in `tomcat10`, the time taken for interpolant generation is also large (3.203 seconds), which annulls the benefits of learning. However, in such cases, learning provides proofs (at a small cost) which we believe amounts to long term benefits, e.g. during regression testing across upgrades.

[5] The table excludes Snugglebug benchmarks on which either ALTER reported inconclusive (due to recursion), did not finish or the run times of both the tools were very small.

Benchmark	WIT?	T(SB)	#FS(SB)	T(NOALT)	MaxD(NOALT)	#FS(NOALT)	T(ALTER)	MaxD(ALTER)	#FS(ALTER)
ant3	Y	>300	>154	0.6	0	1	0.6	0	1
ant4	N	4.17	102	1.9	1	2	1.21	1	2
ant5	N	2.7	66	0.8	0	1	0.87	0	1
batik2	Y	7.6	33	1.0	2	4	0.9	2	4
batik5	Y	11.5	25	18.3	23	91	5.1	9	26
batik7	N	3.5	37	> 300	> 38	> 100	1.3	3	6
batik8	N	4.5	30	2.4	1	3	2.5	1	3
batik9	Y	48.7	89	6.79	3	21	5.6	2	14
batik10	Y	3.8	88	1.7	2	4	1.8	2	4
tomcat9	N	114	26	> 300	> 16	> 74	2.8	0	7
tomcat10	Y	4.9	26	4.1	4	17	3.7	4	17
tomcat11	Y	19.64	7	0.8	0	3	0.86	0	3
tomcat12	N	> 300	>50	0.94	1	2	0.9	0	2
tomcat14	Y	6.1	26	> 300	> 17	> 55	1.778	6	7

Fig. 7: Comparison of Snugglebug (SB), NOALT and ALTER on Java benchmarks. WIT? = witness or not. All times in seconds. MaxD denotes the length of longest call context to the goal function during exploration, #FS denotes the total number of functions summarized during each analysis.

Benchmark	#Goals	Time(NL)	Time(L)	Time(Itp)	Edge(NL)	Edge(L)	LrnReUse	LrnEdge	LrnUpdts
ant3	9	4.013	3.996	0	8	8	0	3	3
ant4	6	1.377	1.527	0.214	3	3	0	2	3
ant5	7	1.302	1.36	0	0	0	0	0	0
batik2	20	1.349	1.589	0.183	4	4	0	1	2
batik7	23	9.319	9.529	0.546	50	41	9	6	7
batik8	24	9.113	9.179	0.461	9	9	0	2	3
batik9	32	8.508	9.879	0.931	31	31	0	8	8
batik10	20	2.558	2.45	0.306	13	9	2	2	3
tomcat9	54	24.511	26.736	0.209	68	68	0	2	2
tomcat10	33	9.193	10.542	3.203	105	39	3	23	23
tomcat11	16	2.519	2.573	0	0	0	0	0	0
tomcat12	18	4.771	4.949	0	16	16	0	0	0
tomcat14	4	2.24	1.934	0.23	17	9	4	4	4

Fig. 8: Evaluation of learning in ALTER on Java benchmarks. Time : Time for analysis. L-learning, NL-No learning, Itp : Interpolant generation during learning. Edge : Number of edges explored in callgraph. LrnReUse : Number of times previous learning helped in backtracking. LrnEdges : Number of edges with learning. LrnUpdt : Total number of learning updates. *batik5* (multiple goals) does not finish because of bugs in our tool.

We believe the results will improve further by employing a single solver for both checking infeasibility and interpolant generation (we used two solvers because we wanted to reuse our existing stable interface to CVC3) and compute interpolants in-memory.

Finally, Fig 7 also shows that ALTER consistently finishes faster than SB. In particular, on `ant3` and `tomcat12`, ALTER finishes quickly while Snugglebug times out (5 minutes). ALTER and SB are architecturally very different and it is difficult to narrow down the cause for the large performance difference to a single factor. One factor is that ALTER computes and reuses local summaries as opposed to SB which may re-analyze procedures for different call contexts. Another factor is that intraprocedurally, ALTER merges symbolic states at join points, whereas SB does not, due to which it needs to propagate a large number of different formulae through a control-flow graph. Finally, SB does not implement alternating scope expansion or learning.

7 Related Work

Loginov et. al. [16] present a closely-related analysis that expands the scope around the goal function in a breadth-first fashion, iteratively analyzing larger scopes until it finds a witness. Breadth-first expansion was also used in the work of Ma et al. [17], which combines forward and backward exploration for testing. In some cases, a strict breadth-first strategy may lead to excessive analysis of irrelevant code, e.g., when the goal function has many callees irrelevant to the property. ALTER uses a more sophisticated alternating search strategy to avoid analyzing such irrelevant code. The probabilistic analysis of Gulwani and Jojic also combines forward and backward exploration [8], but their work does not focus on handling of procedure calls in large programs.

Scope-bounded analysis in DC2 [11] bounds the program scope and computes environment (caller) constraints and (callee) function stubs for the procedures outside the scope using a light-weight whole program analysis. However, scope bounding is performed manually, without automatic scope expansion. ALTER could also be extended to exploit separately-computed caller and callee invariants. Snugglebug (SB) [4] tries to detect bugs by performing backward weakest precondition computation on the interprocedural control flow graph. Unlike ALTER, SB may re-analyze functions for different postconditions, and it does not learn facts from failed backward propagation.

Structural abstraction techniques [1, 23, 22] focus on heuristics for lazy forward expansion. CORRAL [15] performs efficient forward expansion in a stratified manner (a variant of structural abstraction/refinement) together with selective variable abstraction. CORRAL also uses separately-computed invariants to improve search. Unlike ALTER, these techniques have no backward expansion, helpful for deep goal functions, and no automated invariant learning to avoid redundant re-analysis.

Our learning technique is influenced by the DPLL paradigm, in general, and by *lazy annotation* [18], in particular. The latter learns program annotations from failed explorations during path-enumeration-based analysis but starts from the *main* routine, which may make it hard to locate bugs in deep callees. Also, it performs basic block-level expansion and fine-grained learning at the intra-procedural level, which may aggravate path explosion when finding long inter-procedural witnesses. In contrast, ALTER employs local procedure summaries, which avoid re-analysis of procedures as well as both intra- and inter-procedural path explosion. By expanding a whole procedure in one step and learning constraints at procedure interfaces, ALTER is able to focus on inter-procedural exploration without being distracted by repeated intra-procedural analysis.

The SMASH tool [5] employs a combination of *may* and *must* summaries obtained from predicate abstraction and directed symbolic execution, respectively, to avoid redundant re-analysis. Both these summaries are approximations (over- and under-, respectively) of callee side-effects and are useful for forward expansion. Here, we propose to compute caller invariants to improve backward expansion besides employing callee invariants for forward search. Call invariants proposed by Lahiri and Qadeer [14] may be seen as a restricted form of callee invariants which capture the memory footprint unchanged by a procedure.

More broadly, many recent systems for verification and bug detection have been based on predicate abstraction (e.g., BLAST [9] and CPACHECKER [3]). Predicate-abstraction approaches suffer from expensive predicate image computation and, typically, cannot recover from irrelevant refinements. In contrast, ALTER performs a sort of lazy annotation [18] at procedure boundaries, which is able to generalize from invariants specific to a particular call context. Also, while predicate abstraction has worked well on certain kinds of programs (e.g. programs arising from the device-driver domain), it has not been shown to work well on general object-oriented programs. A key challenge with OO programs is heavy use of heap structures, which makes the predicate space that can adequately abstract a program difficult to identify.

8 Conclusions

We proposed a new scalable method to detect inter-procedural bugs using a focused, alternating backward and forward expansion strategy, starting from the goal function. The method iteratively explores the call contexts of the goal function and the callees thereof in an alternating manner, backtracks from infeasible contexts, and learns caller/-callee invariants from failed explorations to prune future search. We demonstrated the effectiveness of our method on large open-source Java programs in terms of faster run times and lesser analysis scopes. In future, we will investigate better forward expansion strategies and improve reuse and management of learned facts.

References

1. D. Babic and A. J. Hu. Structural abstraction of software verification conditions. In *CAV*, pages 366–378, 2007.
2. C. Barrett and C. Tinelli. CVC3. In *CAV*, 2007.
3. D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, 2011.
4. S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.
5. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.
6. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.
7. Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, January 2012.
8. S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *POPL*, 2007.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 2002*.
10. D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, 2004.
11. F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *ASE*, pages 133–142, 2011.
12. Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.
13. A. Kölbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *IJPP*, 33(6):645–666, 2005.
14. S. K. Lahiri and S. Qadeer. Call invariants. In *NASA Formal Methods*, pages 237–251, 2011.
15. A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. In *CAV*, 2012.
16. A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA*, pages 213–224, 2008.
17. K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, 2011.
18. Kenneth L. McMillan. Lazy annotation for program testing and verification. In *CAV*, 2010.
19. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, NY, USA, 1995. ACM.
20. M. Sharir and A. Pnueli. Two approaches to interprocedureal data flow analysis. In *Program Flow Analysis: Theory and Applications*, volume 5, pages 189–234. Prentice Hall, 1981.
21. N. Sinha. Symbolic program analysis using term rewriting, generalization. In *FMCAD*, 2008.
22. N. Sinha. Modular bug detection with inertial refinement. In *FMCAD*, 2010.
23. M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.

Appendix

9 Contextualization

Contextualization. Note that during alternating expansion, the value of both inputs and skolems may be different based on the particular call context they correspond to. For example, in Fig. 3, skolem $sk1$ representing the return value of `bar` may assume different values depending on whether the caller is `runA` or `runB`. To succinctly capture this dependence, we *contextualize* both inputs and skolems by representing them as uninterpreted functions (UFs) from call context to (symbolic) values. Intuitively, this creates skolem *clones* for different call contexts, which expand to different summary values depending on the context. We model a call context symbolically as a tuple $\langle func, l\rangle$ where $func$ represents the last called function and l is a list of call sites. For example, the above skolem is contextualized as $\widehat{sk1}(\langle foo, c_f\rangle)$ where c_f is a variable of list type denoting the call context of `foo`. For an input or skolem isk and context $c = \langle func, l\rangle$, let $\widehat{isk}(c)$ denote the contextualized version of isk. Before we start expanding a local EC ϕ we replace the inputs and skolems by their contextualized versions; all the local summaries are also contextualized appropriately. The backward and forward expansion with contextualization are as follows.

Backward. Given an EC $\phi(in_f, \langle f, c_f\rangle)$ local to f, where in_f represents the inputs of f and c_f denotes the call context of f, the formula obtained after backward propagation into caller h at h_k is $\phi' := \phi(in_f, \langle f, c_f\rangle) \wedge CC(h_k)$ where

$$CC(h_k) := (Ty(h_k) \wedge \psi(h_k) \wedge c_f = [h_k, c_h] \bigwedge_{i \in in_f} \widehat{i}(\langle f, [h_k, c_h]\rangle) = \widehat{Val(i, h_k)}(\langle h, c_h\rangle))$$

Here, c_h is the context variable for h.

Forward. To expand a skolem sk at a call site f_j in f, where sk corresponds to side-effect ret in a callee g. The expansion constraint for sk is $SC := SC_1 \wedge SC_2$. Here, SC_1 contains the summary expressions contextualized by the callee's context.

$$SC_1 := (sk(\langle f, c_f\rangle) = \widehat{sum_{ret}}(\langle g, [f_j, c_f]\rangle))$$

The expression sum_{ret} depends on inputs in_g and skolems sk_g in g. So, we *raise* the inputs in_g to the caller by using the call context values from call site f_j, i.e.,

$$SC_2 := \bigwedge_{i \in in_g} (\widehat{i}(\langle g, (f_j, c_f)\rangle)) = \widehat{Val(i, f_j)}(\langle f, c_f\rangle))$$

Because sk_g are contextualized by c_f initially, SC_1 ensures that now sk_g are contextualized by $[f_j, c_f]$.

We illustrate contextualization using the same example presented in Sec. 4.

Example. In Fig. 4, the initial EC in `p` is $\phi_0 := (x < 10)$. After contextualization, EC is $\phi_1 := \widehat{\phi}(\langle p, c_p\rangle) = (\widehat{x}(\langle p, c_p\rangle) < 10)$. Suppose, we propagate EC ϕ_1 to caller `q` at call site q_2.

$$CC(q_2) := (\widehat{y}(\langle q, c_q\rangle) > 6) \bigwedge (c_p = [q_2, c_q]) \bigwedge (\widehat{x}(\langle p, [q_2, c_q]\rangle) = \widehat{sk_t}(\langle q, c_q\rangle)$$

sk_t is skolem for call to `t` in `q`. On backward propagation and simplification, EC becomes

$$\phi_2 := (\phi_1 \wedge CC(q_2)) \equiv (\widehat{sk_t}(\langle q, c_q\rangle) < 10 \bigwedge \widehat{y}(\langle q, c_q\rangle) > 6)$$

Now, we expand the skolems $\{sk_t\}$ in EC ϕ_2.

$$SC := (\widehat{sk_t}(\langle q, c_q\rangle) = \widehat{d}(\langle t, [q_1, c_q]\rangle) * 2) \bigwedge (\widehat{d}(\langle t, [q_1, c_q]\rangle) = \widehat{y}(\langle q, c_q\rangle))$$

On forward expansion of skolems, new EC is

$$\phi_3 := \phi_2 \wedge SC \equiv (\widehat{y}(\langle q, c_q\rangle) * 2 < 10 \bigwedge \widehat{y}(\langle q, c_q\rangle) > 6)$$

10 Forward Expansion Heuristics

ALTER also employs the following heuristics for efficient forward expansion.

Recursive Skolem Expansion. Repeated forward expansion of skolems in a lazy manner [1] is costly in most cases because the input variables in summary constraints from deep skolems need to be *lifted* up independently through different call contexts multiple times without being merged. This is inefficient, in particular, with call contexts which share common prefixes. In contrast, bottom-up summary composition can merge different summary constraints at join points in the call graph, during backward propagation. To simulate bottom-up composition by forward expansion, we *recursively* expand the skolem before lifting the inputs up, i.e., if the summary constraints $sum(sk)$ for a skolem sk contains other skolems S, then S is expanded before the inputs in $sum(sk)$ are lifted to the current function. Once we obtain the constraints for S, we substitute them in $sum(sk)$ and then lift the inputs up. Thus, deeper skolem summaries are lifted up only once during expansion. Note that as opposed to conventional bottom-up composition of summaries, recursive skolem expansion is goal-directed and restricted only to the callees of the actual call contexts explored during backward expansion by ALTER.

Feedback-driven expansion. The number of potential targets for a virtual function call is quite large in many cases, many of which are irrelevant. We postpone expansion of the skolems for virtual calls (virtual skolems), until the current call context implies a unique target for the call, i.e., the receiver object type at the call location is concrete. If the error condition still contains virtual skolems when the backward search reaches an entrypoint, we expand all potential targets for the skolem. The backward search, in this case, assists the forward search to focus better. Also, during backward propagation from f to h, if h calls f via a receiver object $recv$ (`recv.f()`) at h_k, then we add to $CC(h_k)$ the type constraint subtype$(recv, T)$ where T is the class containing f.

Utilize callee invariants. Before expanding skolems in the current error condition ϕ, we check if ϕ is falsified by conjoining with the callee invariants involving the skolems. In that case, ALTER avoids repeated skolem expansions and backtracks more efficiently.

11 Proofs

Theorem 1. *Given a goal location l, (a) if* ALTER *returns a witness (Wit) result then there must exist a global witness for l, and (b) if* ALTER *returns no-witness ($NoWit$) then no global witness exists.*

Proof Sketch. **(a)** If ALTER returns a witness, it follows from the description that there is a call context ctx from an entrypoint to the goal function g. We reason inductively over the length of this call context. If $length(ctx) = 0$, and the error condition $\phi' := (\phi \wedge SC)$ is satisfiable, then (a) g is an entrypoint and (b) there exists a path p from entry of g to goal location l involving callees whose summary constraints are in SC. Because ϕ' contains no skolems and SC, by construction, contains precise summary constraints for the callees, the path p does not skip over any callee and corresponds to an actual path through the callees. In the inductive case, let $length(ctx) = k$ and $ctx = (e_k, ctx')$ for some call site e_k in an entrypoint function e and a context suffix ctx'. By inductive hypothesis, a local witness p exists corresponding to suffix ctx'. Let

error condition ϕ correspond to ctx'. Now because $\phi' = CC(e_k) \wedge SC(e) \wedge \phi$ is satisfiable, there exists a feasible path p' through function e and its callees which terminates at call site e_k. Because e is an entrypoint, by concatenating p' with p, we get an actual global witness.
(b) Because the error condition ϕ precisely captures all paths in a particular calling context leading to the goal location, if ϕ is infeasible (ALTER returns UNSAT), then there exist no witness path in that calling context. Also, ALTER return UNSAT for a function f only if all its callers return UNSAT. Therefore, ALTER returns UNSAT for the goal function g only if ALTER returns UNSAT for all call contexts to g, i.e., no call context to g contains a witness.

Lemma 1. *The following invariants hold in Alg. 2. (a)* $(\Omega(h) \wedge \Theta(h)) \Rightarrow Inv_h$ *(b)* $(CC(h_k) \wedge Inv_h \wedge \phi \wedge SC)$ *is unsatisfiable at* **L2***,* $SC \wedge \phi \wedge \Omega(f)$ *is unsatisfiable at* **L1***. (c) After* **L2***,* $(\Omega(h) \wedge \Theta(h) \wedge CC(h_k)) \Rightarrow \omega(h_k \rightarrow f)$*.*

Proof. In this proof, we assume that ENTRYPOINT(f) contains the set of functions with no callers. This restriction can be removed by conservatively changing ALTER to return UNKNOWN if a no-caller is not an entrypoint.

(a) The value of Inv_h is obtained after return from the locations **S, C1, E** in code, when UNSAT is detected. At location **C1, E**, $Inv_h = (\Omega(h) \wedge \Theta(h))$. Hence proved. At **S**, $Inv_h = true$, hence the lemma holds trivially.

(b) (i) We first prove that $\Lambda_1 := (CC(h_k) \wedge Inv_h \wedge \phi_f \wedge SC_f)$ is always unsatisfiable at **L2**. Because, Inv_h is returned from calling ALTER for h_k, we consider value returned from each location in **S, C1, E** individually. For labels **S** and **C1**, it follows from the respective UNSAT checks preceding the labels that Λ_1 is also UNSAT. For **E**, note that after update in LEARNΘ for function h, we know that $(\Theta(h) \wedge b) \equiv (\Theta(h) \wedge \phi_h \wedge \Omega(h)) \equiv (\phi_h \wedge Inv_h)$ is UNSAT. But $\phi_h = \phi_f \wedge SC_f \wedge CC(h_k)$. Hence Λ_1 is UNSAT.
(ii) We now prove that $\Lambda_2 := (\Omega(f) \wedge \phi \wedge SC)$ is UNSAT at **L1**. When **L1** is reached directly from **F1**, it follows trivially from the unsat check at **F1**. In the other case, to reach **L1**, all predecessor call sites h_k of f must return UNSAT. Hence it follows from **(b)** (i) and call to LEARNω at **L2** that $\forall h_k.\ \omega(h_k \rightarrow f) \wedge \phi \wedge SC$ is UNSAT. Because, $\Omega(f) = \bigvee_{h_k \in h_k(f)} (\omega(h_k \rightarrow f))$, hence Λ_2 is also UNSAT.
(c) We prove by induction over the number of updates to $\omega(h_k \rightarrow f)$ (ω, in short). Initially, $\omega = true$, which is updated to $\omega' = I$, where I is the interpolant from LEARNω. By property of interpolants, $CC(h_k) \wedge Inv_h \Rightarrow I$. From **(a)**, $CC(h_k) \wedge \Omega(h) \wedge \Theta(h) \Rightarrow I$. Hence the base case is proved. For the inductive case, let $\omega = \omega_0(\omega_1)$, $\Omega(h) = \Omega_0(\Omega_1)$ and $\Theta(h) = \Theta_0(\Theta_1)$ be the previous (new) values of invariants after LEARNω executes. Therefore,
$CC(h_k) \wedge \Omega_0 \wedge \Theta_0 \Rightarrow \omega_0$ (1)
After the current LEARNω update, we have
$CC(h_k) \wedge \Omega_1 \wedge \Theta_1 \Rightarrow I.$ (2)
We need to prove that $CC(h_k) \wedge \Omega_1 \wedge \Theta_1 \Rightarrow \omega_1$ (3)
where $\omega_1 = \omega_0 \wedge I$ and I is the interpolant. To show (3), it is sufficient to show that (i) $\Omega_1 \Rightarrow \Omega_0$ and (ii) $\Theta_1 \Rightarrow \Theta_0$. Because LEARN$\Theta$ updates Θ by only conjoining, (ii) is proved. Let $\Omega_1 = \bigvee_i \omega_1^i$ and $\Omega_0 = \bigvee_i \omega_0^i$. Because LEARN$\omega$ updates ω by only conjoining, $\omega_1^i \Rightarrow \omega_0^i$. Hence (i) is proved.

Theorem 2. *In Alg. 2, for all procedures f, (a) the callee invariant $\Theta(f)$ over-approximates the side-effects of the callees of f, and, (b) the caller invariant $\Omega(f)$ over-approximates the incoming data values from all the callers of f*

Proof.

(a) Let $Fsum(f)$ denote the summary constraints for all the skolems S that may appear in the summary of f. Note that the skolems in S correspond to all possible side-effects that callees of f may have and skolems which appear in any ϕ are a subset of S. To show that $\Theta(f)$ over-approximates the side-effects of callees of f, we must prove that $Fsum(f) \Rightarrow \Theta(f)$. We proceed by induction over number of updates to $\Theta(f)$. Initially, $\Theta(f) = true$ and after update at **L1**, $\Theta(f) = I$ where the $SC \Rightarrow I$. Because SC contains summary constraints for a subset of S, so $Fsum(f) \Rightarrow SC$. Hence $Fsum(f) \Rightarrow SC \Rightarrow I \equiv \Theta(f)$. For the inductive case, assume $\Theta(f) := \Theta_0$ and $Fsum(f) \Rightarrow \Theta_0$. After update, $\Theta(f) := \Theta_0 \wedge I$ where $Fsum(f) \Rightarrow SC \Rightarrow I$. Hence, $Fsum(f) \Rightarrow \Theta(f)$.

(b) Let $I(f)$ denote the set of all possible input vectors to function f as a formula. We need to prove that $I(f) \Rightarrow \Omega(f)$. We proceed by induction over the depth of f from a given entrypoint (the argument holds for a set of entrypoints too). When f is an entrypoint $\Omega(f) = true$ and hence $I(f) \Rightarrow \Omega(f)$. In the inductive case, assume f is called from n call sites $c_1, \ldots c_n$ and for $1 \leq i \leq n$, $I(fc_i) \Rightarrow \Omega(fc_i)$, where fc_i denotes the caller method having call site c_i. Now, by definition,
$I(f) = \bigvee_i (I(fc_i) \wedge CC(c_i) \wedge Fsum(fc_i))$,
$\Rightarrow \bigvee_i (I(fc_i) \wedge CC(c_i) \wedge \Theta(fc_i))$ from Theorem2(a),
$\Rightarrow \bigvee_i (\Omega(fc_i) \wedge CC(c_i) \wedge \Theta(fc_i))$ by inductive hypothesis,
$\Rightarrow \bigvee_i \omega(c_i \rightarrow f)$ from Lemma1(c),
$\equiv \Omega(f)$. Hence proved.

Theorem 3. ALTER *augmented with learning (Alg. 2) returns witness or no-witness results correctly.*

Proof Sketch. It follows from Theorem 1 and 2 that if ALTER backtracks due to check **C1** or **C2** in function f, then either (a) no actual inputs to f may give rise to a global witness to the goal location or (b) there exists no feasible path through the callees of f which may produce a global witness. Hence proved.

12 The Full ALTER algorithm

The algorithm presented earlier in Alg. 2 does not account for *late* skolem expansion, i.e., expansion of skolem sk local to a function f, in one of the transitive callers of f. Alg. 3 presents the complete algorithm and performs learning in presence of late skolem expansion. To this goal, the algorithm, on returning to f, collects late skolem expansions in transitive callers of f (SC' at **L3**). Then it combines SC' with the skolems expanded early in f (SC at **L1**). The combined set is then split into two parts: SC_f, containing skolem constraints local to f, and $SC_{f'}$: skolem expansions for skolems local to the transitive callees of f (which were *late* expanded in f or its callers). This split allows the algorithm to learn $\Theta(f)$ on skolems local to f using SC_f. Finally, it returns $SC_{f'}$ back to the previous callee method on the recursion stack. The proof of the full algorithm is similar to the earlier algorithm with the main differences given by the following lemma.

Lemma 2. *The following invariants hold in Alg. 3. (a)* $(\Omega(h) \wedge \Theta(h)) \Rightarrow Inv_h$ *(b) (i)* $(CC(h_k) \wedge Inv_h \wedge \phi \wedge SC \wedge SC_{h'})$ *is unsatisfiable at* **L2***, (ii)* $SC' \wedge \phi \wedge \Omega(f)$ *is unsatisfiable at* **L3**. *(c) After* **L2***,* $(\Omega(h) \wedge \Theta(h) \wedge CC(h_k)) \Rightarrow \omega(h_k \rightarrow f)$.

Proof. **(a)**, **(b)**.(i) and **(c)** can proved in a similar manner as **(a)**, **(b)**.(i) and **(c)** in Lemma 1.
(b)(ii) We prove that $\Lambda_0 := SC' \wedge \phi \wedge \Omega(f)$ is unsatisfiable at **L3**.
In the same way as Lemma 1.**(b)**.(ii), we can show that $\forall h_k.\, \omega(h_k \rightarrow f) \wedge \phi \wedge SC \wedge SC_{h'}$ is unsatisfiable. Now, $SC' = SC \wedge \bigwedge_i (SC_{h'_i} \mid h_i \in \text{CALLERS}(f))$. Therefore, $\forall h_k.\, \omega(h_k \rightarrow f) \wedge \phi \wedge SC'$ is unsatisfiable.
As $\Omega(f) = \bigvee_{h_k \in h_k(f)} (\omega(h_k \rightarrow f))$, Λ_0 is is unsatisfiable.

Theorem 4. *In Alg. 3, for all procedures* f*, (a) the callee invariant* $\Theta(f)$ *over-approximates the side-effects of the callees of* f*, and, (b) the caller invariant* $\Omega(f)$ *over-approximates the incoming data values from all the callers of* f

Proof Sketch. **(a)** First, from the properties of SPLIT procedure, we can infer that, $Fsum(f) \Rightarrow SC_f$ at **L1** and **L3**. Now, following the proof of Theorem 2(a), we can prove that $Fsum(f) \Rightarrow \Omega(f)$.
(b) Similar to the proof of Theorem 2(b).

```
Initially,
$\forall (h_k \rightarrow f), \omega(h_k \rightarrow f) := true$
$\forall f, \Omega(f) := true, \Theta(f) := true$

Always,
$\Omega(f) := \bigvee(\omega(h_k \rightarrow f) \mid h_k \in$ callers$(f))$

Learn$\omega(h_k \rightarrow f, a, b)$
$I :=$ interpolant $(a, b)$
$\omega(h_k \rightarrow f) := \omega(h_k \rightarrow f) \wedge I$

Learn$\Theta(f, a, b)$
$I :=$ interpolant $(a, b)$
$\Theta(f) := \Theta(f) \wedge i$

interpolant$(a, b)$
return $I$, so that $a \Rightarrow I$ and $I \wedge b$ is unsat

split$(SC, \phi, f)$
let $SC := \bigwedge_i (sk_i = Sum(sk_i))$
$SC_f := true$
$SkoSet_f :=$ getSkolems $(\phi, \langle f, c_f \rangle)$
while $SkoSet_f$ changes do
    foreach $(sk_i = Sum(sk_i)) \in SC$ do
        if $sk_i \in SkoSet_f$ then
            $SC_f := SC_f \wedge (sk_i = Sum(sk_i))$
            $SkoSet_f = SkoSet_f \cup$getSkolems$(Sum(sk_i)))$
$SC_{f'} := \bigwedge_i (sk_i = Sum(sk_i) \in SC \mid sk_i \notin SkoSet_f)$
return $(SC_f, SC_{f'})$

getSkolems$(\phi, c)$
return $\{sk \in$ getSkolems$(\phi) \mid$ call context of $sk = c\}$
```

```
Alter$(f, \phi)$
[S] if $\phi$ is unsat then
    return (NoWit, $(true, true)$)
[C1] if $\phi \wedge \Omega(f) \wedge \Theta(f)$ is unsat then
    return (NoWit, $(\Omega(f) \wedge \Theta(f), true)$)
/* Forward Expansion */
$SC :=$ expandFwd$(f, \phi)$
[F1] if $\phi \wedge SC \wedge \Omega(f)$ is unsat then
    $(SC_f, SC_{f'}) :=$ split$(SC, \phi, f)$
    [L1] Learn$\Theta(f, SC_f, \phi \wedge SC_{f'} \wedge \Omega(f))$
    return (NoWit, $(\Theta(f) \wedge \Omega(f), SC_{f'})$)
if entrypoint$(f)$ then
    if $\phi \wedge SC$ has no skolems then
        return (Wit, $nil$)
    else
        return unknown
$inconcl := false$
$SC' := SC$
foreach $h_k \in$ callers$(f)$ do
    [C2] if $\phi \wedge SC \wedge \omega(h_k \rightarrow f)$ = unsat then
        continue
    /* Backward Expansion */
    $CC_{h_k} :=$ expandBwd$(h_k \rightarrow f, \phi \wedge SC)$
    $ans :=$ Alter$(h, \phi \wedge SC \wedge CC_{h_k})$
    if $ans$ = (Wit, $l$) then
        return (Wit, $[h_k, l]$)
    if $ans$ = unknown then
        $inconcl := true$
    [F2] if $ans$ = (NoWit, $(Inv_h, SC_{h'})$) then
        [L2] Learn$\omega(h_k \rightarrow f, CC_{h_k} \wedge Inv_h, SC_{h'} \wedge \phi \wedge SC)$
        $SC' := SC' \wedge SC_{h'}$
if $inconcl$ then
    return unknown
$(SC_f, SC_{f'}) :=$ split$(SC', \phi, f)$
[L3] Learn$\Theta(f, SC_f, \phi \wedge SC_{f'} \wedge \Omega(f))$
[E] return (NoWit, $(\Theta(f) \wedge \Omega(f), SC_{f'})$)
```

Algorithm 3: Complete Alter algorithm for learning caller invariants Ω and callee invariants Θ with late skolem expansion.