Comprehensive Failure Characterization

***Abstract*—There is often more than one way to trigger a fault. Standard static and dynamic approaches focus on exhibiting a single witness for a failing execution. In this paper, we study the problem of computing a comprehensive characterization which safely bounds all failing program behavior while exhibiting a diversity of witnesses for those failures. This information can be used to facilitate software engineering tasks ranging from fault localization and repair to quantitative program analysis for reliability.**

Our approach combines the results of overapproximating and underapproximating static analyses in an alternating iterative framework to produce upper and lower bounds on the failing input space of a program, which we call a comprehensive failure characterization (CFC). We evaluated a prototype implementation of this alternating framework on a set of 168 C programs from the SVCOMP benchmarks, and the data indicate that it is possible to efficiently, accurately, and safely characterize failure spaces.

I. Introduction

Significant effort in software development is directed at determining whether program's behave as intended. While a test engineer might consider the discovery of evidence of a program failure as the end of the story, in reality it is just the beginning. For example, a development team may follow up by triaging and understanding the failure, then repairing the fault that led to the failure, and finally assessing the validating its correctness and deploying the fix. While it is possible to perform this work manually, there has been significant research into automating these facets of software development, e.g.,[39], [33], [32], [36], [37], [49].

All of these development steps could benefit from expanding the description of the failure from a single input vector to a more comprehensive characterization. Triaging could potentially identify duplicate reports more easily, understanding could be improved by identifying multiple ways that the failure could be exhibited, repairs could be made more robust by covering the full failing input space, and validation could be focused on the space of inputs indicated by such a failure characterization.

In this paper, we explore methods that begin with a single indication of a program failure and construct a rich characterization of the program behavior that may exhibit that failure. Our aim is that this failure characterization be both *comprehensive*, in that it characterizes all failing behavior, and *constructive*, in that it definitively characterizes failing behavior. Moreover, we seek to render the characterization in a form that can be exploited by both automated and manual methods. Towards this end, we define a *comprehensive failure characterization* (CFC) as a pair of logical formulae that bound the failing input space of a program.

Imagine a failure report that includes the test input $(12, 7, "test")$. We seek to compute a pair of formula that define an upper bound, e.g., $I_1 > 0 \wedge I_2 < I_1$, and a lower bound, e.g., $I_1 > 0 \wedge I_2 < I_1 \wedge I_2 < 10$, on the failing input space; I_i represents the ith input. The upper bound defines the space of inputs on which the program may fail, whereas the lower bound defines the space of inputs on which it must fail. We conjecture that this richer failure information can be leveraged in manual and automated development processes. For example, in the example the upper bound indicates that the third input is not implicated in the failure, which may simplify program understanding. Whereas the lower bound establishes a minimal set of inputs for regression testing, the upper bound establishes a maximal set since inputs outside of that bound are guaranteed to be failure-free. We discuss selected applications of CFCs in Sec. VI.

We explore the combination of over and underapproximating static program analyses to compute CFCs. Overapproximating analysis tools, such as AbsInt Astrée [16], Facebook Infer [11], and MathWorks Polyspace [40], have the ability to prove the absence of certain types of program failures. This benefit comes with the downside that reports of failures from these tools may be spurious—they may not correspond to executable program behavior. Underapproximating analysis tools, such as Microsoft SAGE [26], KLEE [10], and Mayhem [12], have the advantage that they never report a spurious failure. This benefit comes with the downside that they may miss failures and, thus, cannot prove their absence.

For more than a decade researchers have understood that there are advantages in using these approaches in combination, e.g., [14], [3], [51], [27]. Conceptually, these techniques alternate the application of over and underapproximating analyses with the output of each driving the other toward convergence. Rather than develop a bespoke alternating analysis, we develop a framework to extract and combine information from existing static analysis tools to achieve the *efficient*, *accurate*, and *safe* computation of CFCs.

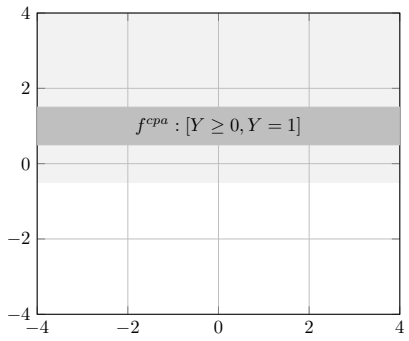
The resulting framework for *alternating characterization of failures* (ACF) leverages the strengths of existing state-of-the-art analysis techniques and tools, while tolerating their limitations. We illustrate the ACF framework in the next section and describe it in more detail in Sec. III. We describe an instance the framework for C programs that incorporates state-of-the-art C analyzers in Sec. IV. We report on an evaluation that assesses the efficiency, accuracy and safety of ACF it to a corpus of 168 C programs in Sec. V.

II. Overview

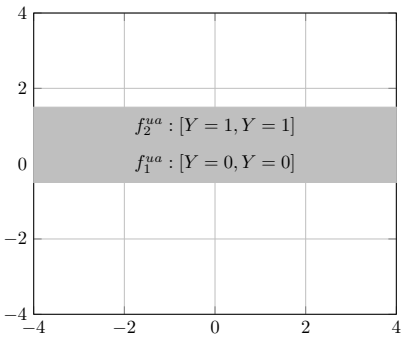
In this section, we describe the concept of a CFC and our proposed analysis framework to compute failure characterizations (Fig. 2) using the small simple example in Fig. 1a.

```
1  int main(int argc, char** argv) {
2   int x = atoi(argv[1]);
3   int y = atoi(argv[2]);
4   int tmp = 0;
5   if (y >= 0) {
6     if (x < 0)
7       x = 0 - x;
8     for (int i=0; i<y; i++)
9       tmp += i;
10  } else {
11    tmp = y*y;
12  }
13  assert(tmp > 0);
14 }
```

(a) Example C program



(b) CFC calculated with CPA



(c) CFC calculated with UA

Fig. 1: Example program with calculated CFCs

a) Detecting Failures: There are a variety of methods for detecting program failures, but we focus here on the use of state-of-the-art static program analyzers. These tools vary in their cost, in their ability to detect failures, and in the guarantees that they provide about failure reports.

For example, CPAchecker [6], [17] (CPA) is the analyzer that finished second in the 2016 SV-COMP competition [45]. When run on this example with the "-svcomp-16" option, the tool reports a failure when the false branch is taken at line 5. This is, however, a spurious failure report that arises because CPA is designed to overapproximate program behavior so as to soundly prove the absence of failures. The abstraction of the program state used by CPA leads to this inaccurate failure report—a problem that is inherent with overapproximation.

Different analyzers may report different failures and with different fidelity. For example, UltimateAutomizer [30], [47] (UA), the overapproximating analyzer that won the 2016 and 2017 SV-COMP competition [46], reports a failure when the true branch at line 5 is taken, then the false branch at 6 and the loop header at 8. Executing this path leads to a violation of the assertion at line 13 and, thus, represents a definite failure of the program.

Unlike CPA and UA, CIVL [42], [13] is an underapproximating analyzer for C programs. It also reports the failure detected by UA and, moreover, it is able to compute that the failure corresponds to running the program with the second parameter equal to 0, i.e., it computes a constraint $Y = 0$ that characterizes the failing input sub-space. For clarity in the presentation we use Y to model the value of variable `y`.

b) Alternating Characterizion of Failures: Our proposed framework seeks to compute a efficient, accurate, and safe characterization of program failures. Our insight is that this can be achieved by alternating the application of over and underapproximating static analyses to leverage their strengths while accomodating their weaknesses. Fig. 2 sketches the architecture of the ACF framework.

The analysis accepts a program, p, and a reachability property, φ, as input; `assert` statements are a natural means of expressing φ. It computes a comprehensive failure characterization, F, which consists of an *upper bound* that is guaranteed to subsume all failing behavior and a *lower bound* that is guaranteed to be subsumed by all failing behavior. F is constructed incrementally by combining characterizations of failure sub-spaces, f, that are computed by iterations of the ACF framework.

The framework relies on the notion of *conditional* program analysis [5] where information is computed about partial analysis results and then used to direct subsequent analysis, for instance, to avoid repeatedly analyzing program behavior.

There are five major phases of the framework, numbered in the black circles, depicted in the Figure along with the data and control flows between them. We elide some detail here, but present a complete explanation in Sec. III and in Alg. 1.

The POSSIBLE failure ❶ phase seeks to detect a new potential program failure using an overapproximating analysis $\overline{A}$. It uses the upper bound of the most recently computed failure sub-space, $f.up$, to condition the analysis to explore new behavior. If $\overline{A}$ is unable to detect a new failure then the conditioning, c, is *generalized* and the analysis repeated. When generalization is required we say that the conditioning was *ineffective*. This iterative process will lead to the detection of a new failure or to proving the absence of additional failures, then the final effective conditioning is used to define the failure sub-space, $[c, f.lo]$, to MERGE with F (abbreviated M in the figure). The phase returns evidence of a failure or validity, as e_o where validity is denoted $\overline{e}_\varphi$, and the effective conditioning, c, that allowed $\overline{A}$ to compute that result.

The DEFINITE failure ❷ phase seeks to confirm a potential failure, or demonstrate that it is spurious, using an underapproximating analysis $\underline{A}$. It uses the overapproximating evidence of failure, e_o, to condition the analysis to explore behavior in the neighborhood of the potential failure. This phase also uses an iterative refinement of conditioning to compute a characterization of the failure, $\underline{e}_{\neg\varphi}$, which is returned as e_u. In this case, however, the conditioning is *specialized* to further restrict the analysis. If the refinement process fails, then $\underline{A}$ is applied to characterize a region of valid behavior, by detecting violations of $\neg\varphi$, under the conditioning of e_o—this is used to block subsequent spurious failure reports.

These phases are complementary. For example, the first

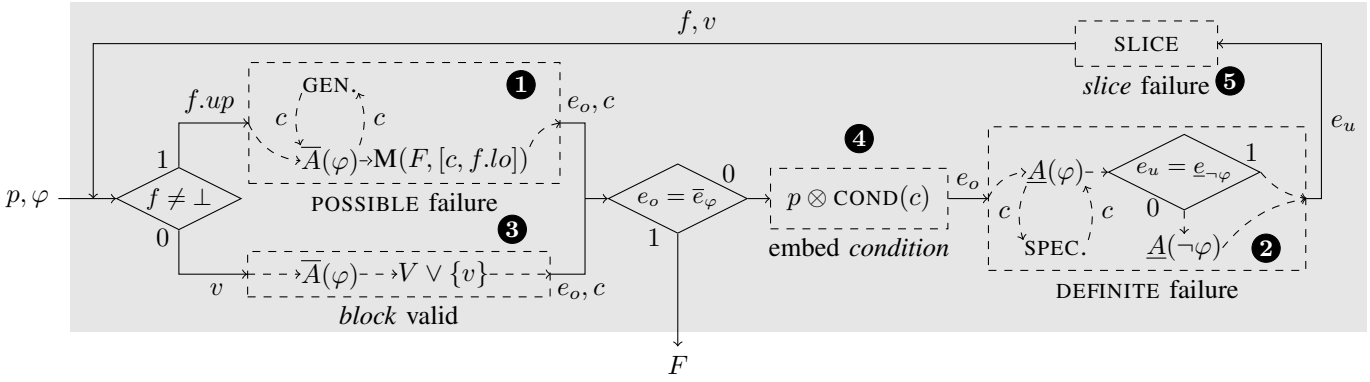


Fig. 2: Alternating Characterization of Program Failure

exploits abstraction approaches to efficiently analyze looping program behavior, whereas the second exploits symbolic representations to accurately analyze complex branching along program paths. The alternation of these phases use conditioning in complementary ways. Conditioning directs overapproximating analyses away from failure sub-spaces that have been already analyzed and directs underapproximating analyses towards a sub-space that harbors a potential failure.

c) An Example: Consider a configuration of the framework where $\overline{A}$ is CPA and $\underline{A}$ is CIVL. The analysis begins, with $f \neq \bot$, by running phase ❶ using CPA with $f.up = false$, i.e., no behavior is excluded from the conditioned analysis. As explained above, CPA reports a failure along the path $\ldots, 5, 11, 13$ which is encoded in e_o and c is initially false. Since evidence of a failure is found the comparison with $\overline{e}_\varphi$ is false and ACF continues to the right in the Figure—in this case the conditioning is trivial so the program is unchanged. Phase ❷ uses e_o to condition CIVL by directing it to analyze the failing subspace of behavior reported by CPA. CIVL is unable to find an error, so it analyzes the complementary property to compute $Y < 0$ as evidence of valid behavior which is returned as e_u and which is fed back as v to begin a second iteration.

On this iteration, $f = \bot$ and $v = Y < 0$ which activates phase ❸ to block CPA analysis from considering that path. This succeeds in avoiding the spurious failure and CPA reports a new failure $e_o = \ldots, 5, 6, 7, 8, 9, 8, 13$ with $c = v = Y < 0$. With non-trivial conditioning phase ❹ embeds $Y < 0$ into the program, using an `assume` statement, to ensure that no subsequent analysis considers it. In the second exection of phase ❷, CIVL confirms this failure and computes $e_u = Y \geq 0 \wedge X < 0 \wedge Y = 1$. Phase ❺ is able to determine that input X is independent of the failure, through a form of program slicing, and calculates a safe failure characterization of $f = [Y \geq 0 \wedge Y = 1, Y \geq 0 \wedge Y = 1]$ which is fed back to begin a third iteration.

CPA attempts to restrict analysis to exclude $f.up$, but its abstractions are too imprecise and it finds the same error as before. This triggers the generalization of the upper bound of f to be $Y \geq 0$ which is effective in blocking further failure reports by CPA and the algorithm terminates with $F = \{[Y \geq 0, Y = 1]\}$. Fig. 1b illustrates the bounding characterization of the failure space that is computed—failures may be exhibited when $Y \geq 0$, the light gray space, and they must be exhibited when $Y = 1$, the darker gray space.

This execution of the ACF framework required 3 iterations involving 4 conditioned runs of CPA, 3 conditioned runs of CIVL (2 for φ and 1 for $\neg\varphi$), generalization of 1 failure characterization, blocking of 1 set of valid behaviors, slicing of 1 clause (and variable) from the characterization, and it resulted in a CFC whose upper bound substantially overapproximates the lower bound.

Instantiating the framework with UA instead of CPA and running on the same program requires just 2 conditioned runs of UA and CIVL and involves no generalization. Slicing is applied to each element of the computed CFC to remove a clause and the resulting upper and lower bounds coincide as illustrated in Figure 1c.

III. An Alternating CFC Framework

Let P be the domain of programs and Φ the domain of reachability properties. A program analysis, A, targets a pair of elements $\langle p, \varphi \rangle$, $p \in P$ and $\varphi \in \Phi$, in order to provide information about whether the executions of p conform to φ, i.e., whether $p \models \varphi$.

Most modern program analysis frameworks produce some form of evidence about their claims of property validity or violation. Evidence of a violation can be as complete as a trace of program execution leading to a potential failure or as partial as designating a single statement at which the failure may occur. A general model for such evidence, termed an *error automaton*, has been developed by the organizers of SV-COMP [7]. These automata would be used as evidence of failure for an overapproximating analysis, e.g., e_o on output from ❶ or ❸. Error automata have a start state that coincides with the initial program state and transitions that are labeled by control flow branches. The language of an error automaton is a set of program executions, e.g., the set of all executions for an automaton that accepts immediately, or a single execution which has a single enabled transition at each state.

Let E be the domain of evidence produced by a program analysis. This would include evidence of property validity, e_φ, or violation, $e_{\neg\varphi}$. An evidence producing program analysis is $A : P \times \Phi \rightarrow E$ All 32 analyzers in the 2017 SV-COMP

competition are evidence producing including: CPA, UA, and CIVL.

An over-approximating analysis $\overline{A} \in A$ is one where $\overline{A}(p, \varphi) = e_\varphi \implies p \models \varphi$. An under-approximating analysis $\underline{A} \in A$ is one where $\underline{A}(p, \varphi) = e_{\neg\varphi} \implies p \not\models \varphi$. Conceptually, $\overline{A}$ is capable of proving property validity and $\underline{A}$ is capable of proving property violation.

We assume that a program, $p \in P$, has a set of input statements, $i \in I$, that return well-typed values, where $dom(i)$ is the domain of i's type. To simplify the presentation of ACF, in this paper we formalize the analysis for programs that read from each input statement a single time, thus the input domain of p is $D = \prod_{i \in I} dom(i)$. Given this D is finite. The framework, and the implementation described in Sec. IV, work more generally by modeling the kth execution of an input statement by the pair (i, k), and the input domain as the union of execution-specific input domains, $\prod_{1 \leq j \leq n} dom(i_j)$, where the inputs on an execution are $\langle (i_1, 1), \ldots, (i_n, n) \rangle$. The implementation can enforce a bound on the length of sequence to restrict analysis to a finite D; in doing this it may lose accuracy by characterizing all inputs beyond the given length as potentially failing.

We consider sequential programs in this work; leaving extensions to concurrency for future work.

A. Failure Characterization

We characterize failures using logical formulae that encode regions of D on which failures are detected. Let $F_{p,\varphi}$ be an exact characterization of the input space on which the program fails relative to φ.

Definition 1. *A comprehensive failure characterization (CFC), $[\overline{C}, \underline{C}]_\varphi$, is a pair of logical formulae that semantically bound the program's failure space, $F_{p,\varphi}$, such that $\underline{C} \implies F_\varphi \implies \overline{C}$.*

CFCs lie between two extremes: an *exact* CFC, where the bounds coincide, i.e., $\underline{C} = \overline{C} = F_\varphi$, and a *trivial* CFC, where the bounds are non-informative, i.e., $[false, true]$. We work with a particularly advantageous form of CFC that permit algorithms to operate on elements of the CFC encoding.

Definition 2. *A disjoint CFC is a set of pairs of formulae F, where $f \in F$ can be written $f = [f.up, f.lo]$, and such that the upper bounds are disjoint, $\forall f, f' \in F : f \neq f' \implies f.up \wedge f'.up = false$, and the upper bounds subsume the lower bounds, $\forall f \in F : f.op \wedge \neg f.lo = false$. We write $upper(F) = \bigcup_{f \in F} f.up$ and $lower(F) = \bigcup_{f \in F} f.lo$.*

In a disjoint CFC, there may be elements of the upper bound, for which the associated lower bound cannot be characterized precisely. In such a case the lower bound is $false$. While this lower bound is non-informative, its use is logically consistent given the disjunctive nature of the lower bound.

Ideally F encodes a small failing subspace of D. In the worst-case, $upper(F) = D$ and all of the input space is implicated. Even here ACF is able to isolate the imprecision to one element of the partition that will have as its upper bound the complement of the disjunction of all of the other upper bounds. The remaining partitions provide useful information about failures, especially in their lower bounds.

For example, on the program Ackermann02_false-unreach-call_true-no-overflow_true-termination.c, UA directs CIVL to the failure, and after blocking this exact characterization, UA declares the remaining program failure-free. This is the ideal case. At the other extreme, on the program pc_sfifo_2_false-unreach-call_false-termination.cil.c, UA finds directs CIVL to a failure on the first iteration. After this failure is blocked, the overapproximators can neither find a new failure nor declare the program safe, and the upper bound moves to $true$.

B. Conditioning Program Analysis

The goal of conditioning is to restrict the program behavior that is subjected to analysis. For example, to restrict the propagation of abstract states across a branch in $\overline{A}$ or prune the exploration of a sub-tree in $\underline{A}$.

There are many possible ways to define conditioning, and in this work we employ two different approaches. First, given e_o as an error automaton, the automaton structure is used to direct the state-space search performed by $\underline{A}$. Any branches that are inconsistent with e_o go unexplored. Second, given e_u as a logical formula defined over D, we instrument the program with the statement $\texttt{assume}(\neg e_u)$ to inform $\overline{A}$ that it need not consider that behavior. Regardless of the approach used, we denote the conditioning of p's behavior for analysis as $p \otimes COND(c)$, for conditioning c.

While conditioning $\overline{A}$ aims to restrict the analysis to avoid previously analyzed failing sub-space, it does not guarantee that this will be effective. For example, if the expression e in $\texttt{assume}(e)@l$ cannot be precisely modeled by $\overline{A}$'s abstract domain then the semantics of the assume will be overapproximated. Generalization is used to address this issue.

Dually, conditioning $\underline{A}$ aims to restrict the analysis to a failing space of the program behavior in order to confirm the failure and characterize it. Effective conditioning for $\underline{A}$ means that the failures that are characterized are guaranteed to be executable. When such guarantees are not computed, e.g., due to timeouts or overapproximation in underlying constraint solvers, specialization further restricts the conditioning.

Within phases ❶ and ❷ the use of conditioning is speculative in that we are seeking to determine if the conditioning is effective. In the overall flow of the ACF algorithm, phase ❷ generates candidate conditioning, phase ❶ confirms that it is effective or generalizes it until it is, and phase ❹ embeds it into the program model for use in all subsequent analyses.

C. An Alternating CFC Algorithm

The structure of the algorithm is depicted in Figure 2. Algorithm 1 provides addition details of the computation of CFCs. The ACF algorithm depends on certain properties of the algorithms that it combines. Specifically, it assumes that over and underapproximating program analyses are typed:

$$\overline{A} : \mathcal{P} \times \Phi \to \{\overline{e}_\varphi, \overline{e}_{\neg\varphi}, \bot\} \quad (1)$$

$$\underline{A} : \mathcal{P} \times \Phi \to \{\underline{e}_{\neg\varphi}, \bot\} \quad (2)$$

where $\bot$ encodes the inability of the analysis to compute a result, e.g., due to a timeout, encountering an unsupported language feature, or unsoundness relative to the nature of the analysis' approximation.

In addition, it assumes the strict monotinicity and anti-monotonicity of generalization and specifialization with respect to a finite order on formulae.

Function POSSIBLE, lines 1-9, along with lines 32-33 realize phase ❶. The additional detail in the algorithm is in the use of a cache of previously computed analysis results, *seen*, and the logic that continues GENERALIZATION as long as no new analysis result is seen or the conditioned analysis is ineffective. Note here that the generalization process will converge to $true$, due to monotinicity. Line 33 will MERGE the new component of the CFC into F ensuring that the result is a disjoint CFC.

Function DEFINITE, lines 10-22, along with line 46 realize phase ❷. Here the original conditioning, c, is recorded, in c', for later use should the analysis not be able to detect a failure. As above, the iterative SPECIALIZATION of conditioning will terminate when the conditioned analysis is effective or when the specialization reaches the limiting $false$ value, due to anti-monotonicity. If a sound underestimating characterization of the failure, $\underline{e}_{\neg\varphi}$, is found it is returned. Otherwise the original conditioning is used to analyze the negation of the property, i.e., to find a definitive characterization of program behavior that is consistent with φ. This may be ineffective, in which case evidence of valid behavior is $false$ which will trigger termination on the next iteration in phase ❸.

The main ACF algorithm initializes the data structures, lines 24-29, and then begins the alternating iterative computation of a CFC. Note that the algorithm uses p' as the version of p that accumulates the effective conditioning across iterations.

The pair f, v drive the behavior of each iteration. One or the other is well-defined, the other has value $\bot$, and it is used to select the phase to execute.

Once completed if the phase has determined, at line 42, that there are no additional failures or that $\underline{A}$ was ineffective, then the accumulated CFC, F, is returned. Otherwise, effective conditioning was computed and it is embedded into the program at line 45.

Line 47 applies slicing to the program using the evidence of failure, e_u, which can be thought of as encoding a program trace. A dynamic slicing algorithm can be used to eliminate branches from the trace that are independent of the failure.

Finally, lines 48-52 determine whether the results of phase ❷ computed evidence of failure or evidence of validity and update the f, v pair accordingly.

Theorem 1 (Termination)**.** *Algorithm 1 terminates if $\overline{A}$ and $\underline{A}$ terminate and* GENERALIZE *and* SPECIALIZE *are strictly monotone and anti-monotone relative to a finite ordering on formulae, respectively.*

Proof. There are three loops in the algorithm: lines 1-6, lines 13-16, and lines 30-53.

Algorithm 1 Alternating Characterization of Failures

```
1:  function POSSIBLE(p, c, φ)                          ▷ Detect possible failure
2:      e ← Ā(p ⊗ COND(c), φ)
3:      while e ∈ seen ∨ e = ⊥ ∨ c ≠ true do
4:          c ← GENERALIZE(c)
5:          e ← Ā(p ⊗ COND(c), φ)
6:      end while
7:      seen ← seen ∪ {e}
8:      return e, c                          ▷ evidence, effective conditioning
9:  end function
10: function DEFINITE(p, c, φ)                          ▷ Detect definite failure
11:     c' ← c
12:     e ← A̲(p ⊗ COND(c), φ)
13:     while e = ⊥ ∨ c' ≠ false do
14:         c' ← SPECIALIZE(c')
15:         e ← A̲(p ⊗ COND(c'), φ)
16:     end while
17:     if e = e̲_¬φ then
18:         return e                                          ▷ failure evidence
19:     else
20:         return A̲(p ⊗ COND(c), ¬φ)                ▷ validity evidence
21:     end if
22: end function
23: function ACF(p, φ)
24:     F ← ∅
25:     V ← ∅
26:     f ← [false, false]
27:     v ← false
28:     seen ← ∅
29:     p' ← p
30:     loop
31:         if f ≠ ⊥ then
32:             e_o, c ← POSSIBLE(p', f.up, φ)
33:             F ← MERGE(F, {[c, f.lo]})
34:         else
35:             e_o, c ← Ā(p' ⊗ COND(s), φ), v
36:             if e_o = ⊥ ∨ c = false then
37:                 F ← F ∪ {[(⋀_{f∈F} ¬f.up) ∧ ⋀_{v∈V} ¬v, false]}
38:             else
39:                 V ← V ∪ {v}
40:             end if
41:         end if
42:         if e_o = ē_φ ∨ e_o = ⊥ then
43:             return F
44:         end if
45:         p' ← p' ⊗ c                  ▷ embed conditioning henceforth
46:         e_u ← DEFINITE(p', e_o, φ)
47:         e_s ← SLICE(p', e_u)
48:         if e_u = e̲_¬φ then
49:             f, v ← [e_s, e_s], ⊥
50:         else
51:             f, v ← ⊥, e_s
52:         end if
53:     end loop
54: end function
```

Termination depends on the existence of an ordering on the formulae that are used to encode conditioning. The framework can be instantiated with any finite ordering. For example, containment ordering on sets of conjuncts from a CNF encoding of a formulae.

An execution of lines 1-6 (13-16) is guaranteed to terminate if every call to $\overline{A}$ ($\underline{A}$) terminates and if GENERALIZE (SPECIALIZE) produces a result that is greater (lesser) in the ordering due to the finite bound on chains in the ordering.

The outer most loop executes once for each element of F and V that is computed. New elements are only computed if the conditioning is effective or in two special cases: the loop at line 3 exits because $c = true$ in which case $e = \bot$ which triggers termination at line 42, or line 37 is executed which is also followed by termination at line 42. There can be only finitely many elements of F and V, since the upper bounds of $f \in F$ and the elements of V are disjoint and their union must be a subset of D. □

Theorem 2 (Safety)**.** *Algorithm 1 terminates with F such that the failure space of p relative to φ, $F_{p,\varphi}$, is bounded by F, $lower(F) \implies F_{p,\varphi} \implies upper(F)$.*

Proof. For all elements of F, the lower bounds are computed by $\underline{A}$ and then sliced. By definition $\underline{A}$ computes a safe underapproximation and thus any failure detected is guaranteed to be feasible. If no failure is detected, then $false$ is used as the lower bound which is guaranteed to be safe. Slicing only eliminates sub-formulae that are provably independent from the failure characterized by the partition. Thus, while the sliced lower bound may subsume the original error detected by $\overline{A}$ it is guaranteed to underapproximate the space of all errors that are equivalent up to execution of independent branches.

For any iteration of the ACF algorithm, the upper bounds from all prior iterations are conditioned into the program, by line 45, and the upper bound from the current element of f is conditioned in POSSIBLE. The algorithm terminates only if: (1) the conditioning of all upper bounds permit $\overline{A}$ to prove the program free of failures, or (2) a final run of $\overline{A}$ results in $\bot$ in which case line 37 adds an element to F that guarantees that $upper(F) = D$. The first case guarantees that the upper bounds of F are safe, since $\overline{A}$ computes a safe overapproximation, and the second case is trivially safe. □

IV. A Prototype ACF Implementation

To explore ACF we implemented Alg. 1 along with incorporating or adapting other analysis tools to define the following components: analyses, slicing, conditioning, generalization, and specialization.

Our implementation consists of 2770 non-comment source lines (NSLOC) of Ruby. We chose Ruby because it has facilities for easily processing text-based files which plays an important role in integrating external tools, e.g., parsing tool output. These facilities also enabled us to implement the source-level instrumentation that is required by conditioning.

We plan to share our prototype implementation with the broader community later in 2017. This will include sharing both the standalone ACF components and going through the pull-request approval process for external tools.

Analyses We focused on analysis tools that participated in the annual International Competition on Software Verification (SV-COMP). This allowed us to take advantage of the already-existing corpus of hundreds of benchmarks of C programs submitted by the community. The competition also requires that these tool report possible failures in a specified language, so there was a common interface that we could build on.

Our implementation permits multiple overapproximators to be used in a sequential portfolio, i.e., we run one after the other in sequence; we plan to replace this with a parallel portfolio to reduce analysis time. We incorporate UA and CPA, since both performed well in the two most recent SV-COMP instances. In addition to the standard sequential configuration of CPA, we also used two other CPA configurations that participated in the competition—one that combines a value analysis, a predicate analysis, and a technique for caching analyzed blocks [24], and another that implements a variant of k-induction [4].

UA and three different configurations of CPA constitute our portfolio of overapproximators. The prototype also uses a portfolio of underapproximators, but we have only populated it with CIVL. The portfolio management implementation establishes an execution time bound ensuring that all analysis runs terminate.

Our prototype can accommodate any number of analyzers, or configurations, from the 32 that participated in SV-COMP 2017 and we will explore the benefits of a broader portfolio once we can execute them in parallel.

Slicing To determine which branches were relevant in triggering the failure, we implemented a version of Xin and Zhang's dynamic slicing algorithm [48], on the CIVL representation of the replayed failure trace. This algorithm detects branches on the failing trace that are independent from the failure and, consequently, can be removed. Because this slicing algorithm detects only *dynamic* control dependence, to ensure a safe underapproximation—one whose logical formula encodes definite failure traces—we also run an inexpensive static analysis.

This static analysis seeks to determine that branches not taken will not impact the detected failure. To do this, for each branch identified as independent we run a depth-first search (DFS) on sub-control flow graph rooted at the branch not taken in the trace; the DFS backtracks when it rejoins the failure trace. If the DFS encounters a "suspicious" statement, then we assume that there may be a dependence and revert the classification of the branch as independent. We define a suspicious statement as one of: assignment to a variable that is data-dependent on the failure, a `goto` statement, a function call, and any statement involving pointers (including arrays). Slicing is implemented as a configuration option, `-sliceAnalysis`, within CIVL's error "replay" feature. The implementation consists of 1851 NSLOC of Java code.

Conditioning There are two kinds of conditioning in an ACF. The first is a standard application of directed symbolic execution, where the execution engine is guided down specified

branches, instead of exploring all possible branches. This kind of conditioning only applies to the underapproximators.

The directions at each branch come from the witness graphml files produced by the overapproximator when a possible failure is reported. If a branch is not specified in the witness file, then we inject no instrumentation at this branch, and the underapproximator will do its standard exploration of both branches at that point. This is implemented as a configuration option, `-direct`, within CIVL's "verify" feature. The implementation consists of 338 NSLOC of Java code.

In our experience, both UA and CPA produce witnesses that give full direction, though our implementation of directed symbolic execution can handle the general case of partial direction.

The other kind of conditioning is seen by both the over and under approximators. This is the injection of `assume` statements, intended to direct the tools away from exploring already-analyzed subspaces. Within the `assume` statements, we place the negated formulae of the upper characterization. We place the `assume` statements just before the points at which a failure is asserted. This was an implementation decision that guaranteed that all inputs have been read before assuming constraints on their values. In the future, we would like to explore placing the `assume` statements at the top of the program to immediately prune the state space that has yet to be explored. Overapproximator conditioning is implemented in our Ruby code base.

Generalization Initially ACF attempts to condition $f.up$ (line 32 in Alg. 1) to avoid previously detected failures. If that is unsuccesful, generalization is applied to compute effective conditioning.

The prototype implements a structural generalization of $f.up$, which is always a conjunctive formula. To generalize a conjunction of n clauses, we first construct the powerset lattice over the set of conjuncts; this lattice has height n, $f.up$ as the top element (at height n), and $true$ as the bottom element (at height 0). Having failed to demonstrate that the element at height n constitutes effective conditioning, we determine whether the elements at height 1, i.e., the singleton clauses, are effective. For those that are effective we compute the least-upper-bound and determine if it is effective; if k singletons are effective then the lub will be at height k. We refer to this as a "round" of a binary search on the conditioning sub-lattice. Each round successively narrows that sub-lattice, by raising the height of the bottom and lowering the height of the top. This process defines a bounded finite order and thus, generalization is guaranteed to terminate. Generalization returns the effective condition that is lowest in the lattice. If none are effective, then $true$ is returned.

Generalization is implemented in our Ruby code base. For very large formulae, we control runtime by specifying an upper bound on the number of rounds of generalization that can be computed.

Specialization In our prototype ACF, there is no implementation of specialization. The need for specialization—the underapproximator returning a failure report that cannot be verified with certainty—did not arise in the 168 programs analyzed.

V. An Exploratory Study of ACF

We conducted a study to explore the cost and effectiveness of ACF for computing CFCs of C programs. Our goal is to provide information about the efficiency, accuracy, and safety of the ACF framework.

RQ1: *How does the total ACF runtime and the runtime of ACF components vary across programs?*

RQ2: *How does the accuracy of the computed CFCs vary across programs?*

RQ3: *How do the ACF components that ensure safety influence the efficiency of ACF?*

A. Subject Selection

Our study uses a selection of the SV-COMP benchmarks [44]. In particular, we started with the set of 3624 C programs in the benchmark that have failures; these are a combination of real failures and seeded failures. Our goal is to explore the cost-effectiveness ACF so we limited ourselves to programs on which at least one overapproximator completed within the timeout of 900 seconds. This left 718 C programs with failures. SV-COMP has both failing and non-failing variants of C programs and in ACF we expect to condition analyses so that they can ultimately verify a program failure-free. For this reason, we removed 128 benchmarks for which the non-failing variant could not be proven so by an overapproximator within the timeout. A similar filtering removed 244 benchmarks based on the ability of the underapproximator to complete its analysis. This left 346 C programs for our study.

During the course of our study we detected 104 SV-COMP benchmarks that either read no input or read no input on failing behaviors. We do not think that these are representative of real programs failures, since all program runs lead to failure, and keeping these in our study would inappropriately, albeit positively, bias our results. We established a maximum runtime for ACF of 13 hours, which led us to drop another 74 programs.

This selection process resulted in a diverse set of 168 failing C programs that span the gamut of categories in the SV-COMP benchmark.

B. Experimental Setup

Our implementation is purely sequentially and was executed on a Opteron 6376 processor (2.3 Ghz) with 192 GB of RAM running Scientific Linux. Our analysis portfolio used configurations of: CPA version 1.6.1, UA version 0.1.8, and CIVL version 1.7.3, with a 15 minute timeout.

C. Results and Discussion

We report results of running our ACF implementation on the 168 SV-COMP C programs both in aggregated data and through a series of plots that depict the variability of informative metrics across the programs. All of the underlying data from our study are available at http://www.mediafire.com/file/9biz38d74icd831/analysis_logs.zip.

Component	Avg.	Max.	Min.	% Total
POSSIBLE	4419	24766	22	43.6%
DEFINITE	86	392	2	0.8%
GENERALIZE	4381	38042	0	43.2%
SLICE	90	430	2	0.9%
Total	10137	42481	29	

TABLE I: ACF runtime (rounded to the nearest second)

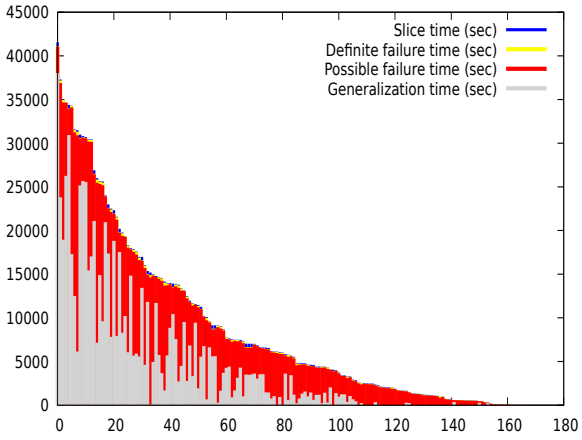


Fig. 3: Total and Component ACF Runtime

Each plot uses a single "impulse" to record the metric for the run of ACF on a program. The impulses are plotted from high to low moving left to right on the horizontal axis. These plots are effective at depicting the trend across the set of subject programs. Note that the ith impulse in a pair of plots may not correspond to the same program.

RQ1 (efficiency): Table I reports the average, maximum, and minimum times, in seconds, to run ACF in our study. The average time to compute a CFC is 2.8 hours. This runtime is dominated by the cost of running overapproximators and generalization; each taking on average just over 43% of the runtime. Not listed in the table is the cost of the core ACF implementation which, for example, integrates the components, performs conditioning, and merges disjoint partitions. This comprises 11.5% of the analysis time on average.

Fig. 3 shows substantial variability in runtime across the 168 programs. This is a stacked impulse plot with generalization on the lowest part of the impulse, then possible failure time, then definite failure, and slice time at the top. Definite failure and slice times are visible when zooming in on the plot.

Fig. 4 shows the variability in the number of iterations of ACF needed to reach convergence. This ranges from 1 to 27 across the data set, but more than a third of the programs required 5 or more iterations. This is due to, at least in part, to the fact that for more than 50 programs the overapproximators detected spurious failure reports which had to be blocked to reach convergence.

While the reported runtime of ACF is significant, the study has revealed two optimizations that promise to significantly reduce that time.

First, replacing our sequential portfolio with parallel execution of instances of $\overline{A}$ will reduce POSSIBLE by up to a factor

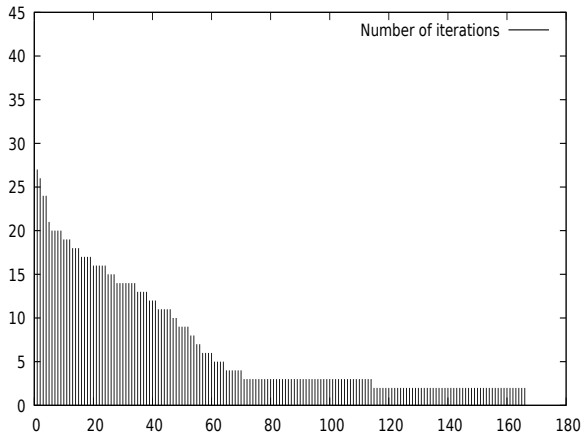


Fig. 4: ACF Iterations to Convergence

of 4. While this best case is unlikely to be observed, more than half of the benchmarks encountered timeouts, as many as 19 for CPA and 27 for UA in a single run, which suggests significant room for improvement.

Second, each round of generalization analyzes two "layers" of the powerset lattice of conjuncts for a given $f.up$. The first round involves a single run of $\overline{A}$ conditioning all conjuncts, i.e., the full formula, and $|f.up|$ runs each conditioning a single conjunct. We limited our experiments to a single round and across the 279 generalization instances in the study we observed $1 \leq |f.up| \leq 72$, with an average of 13. We use the sequential portfolio to solve these, which means that parallelizing generalization could reduce runtime by a factor of 4-288 and a factor of 52 on average.

RQ2 (accuracy): The accuracy of a CFC should be judged based on the the input space it describes, i.e., how many failing inputs are not characterized by the lower bound, how many non-failing inputs are characterized by the upper bound. This presupposes we know the exact failure space of the program, which is hard to determine. In this study, we use two proxy measures to provide information on accuracy. First, we know that any CFC partition that is exact is completely accurate—the upper and lower bounds coincide. Second, we know that any CFC partitions that have $true$ as an upper bound are inaccurate—since we removed programs from the study that failed on all inputs. Intuitively, the greater the number of exact partitions and the fewer the number of $true$-partitions the more accurate the analysis.

The CFCs for all 168 programs in the study were comprised of 532 partitions. 265 (49.8%) of the partitions were exact and 5 had $true$ values as upper bounds.

Fig. 5(middle) plots the variability across the examples in terms of the number of partitions computed for the CFCs and whether those partitions were exact or inexact. The significant number of inexact partitions corresponds to the need for generalization across the study subjects.

Generalization influences the accuracy of upper bounds, but slicing influences the accuracy of lower bounds, i.e., they are generalized yet they remain underapproximations of failure. Fig. 5(left) plots the effect of slicing across the study. For

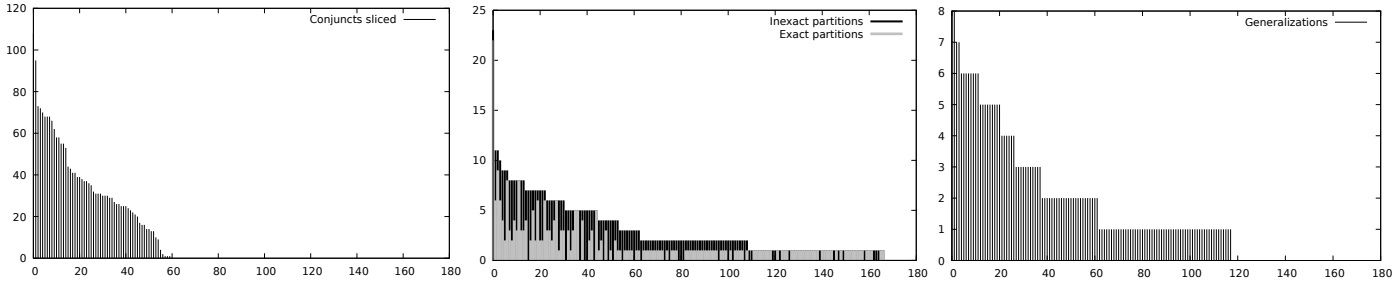


Fig. 5: Data on ACF accuracy and safety

more than a third of the programs, some slicing is performed indicating its value. Slicing has an effect on performance as well, since reducing the size of $f.up$ reduces the cost of generalization.

Our prototype ACF computed exact CFCs for 49 programs. For 55 programs, the gap in bounds across all partitions in the computed CFC was a single conjunct—generalization moved just one step up the lattice. The maximum gap in our study, as measured by conjuncts, was 7. In total, CFCs for 163 of the 168 programs had non-$true$ upper-bounds, indicating that they characterized a strict subset of the input space as failing.

Our study does not quantify accuracy in absolute terms, nor does it quantify the partition gaps in terms of their semantics, i.e., the size of the partitions input sub-space. We plan to explore this in future work. The study does provide evidence that even when ACF is configured with a limited form of generalization it can converge to CFCs that bound the failure space to a strict subset of the input space.

RQ3 (safety): The safety of CFCs computed by ACF is based on the soundness of $\overline{A}$ and $\underline{A}$ and the use of generalization. We take the former as a given and present data on the use of generalization in Fig. 5(right).

More than two-thirds of the subject programs require at least some generalization. This is a clear indication of the necessity of generalization for computing safe CFCs. The discussion above describes the runtime cost of generalization and while we have suggested ways that cost may be reduced, that cost is unavoidable in computing safe CFCs.

D. Validity and Generalizability of Findings

With regard to internal validity, we have conducted extensive testing and post-analysis of the computed CFC data to ensure that it is safe and have only used static analyses that have proven to be robust in the SV-COMP competitions [45], [46]. The SV-COMP benchmark suite, while limited, is updated annually to reflect the challenges to static analyses found in the broader population of C programs. While we do not claim generalizability of our results to all C programs, using SV-COMP programs constitutes a common convenience sample used in evaluating C static analysis tools. Moreover, its use promotes the replicability and reproduction of our study.

VI. Applying CFCs

We conjecture that CFCs have a wide range of software engineering applications. We discuss two such applications here.

a) Fault Repair and Fix Completeness: The literature has demonstrated the challenges of fixing a bug completely, i.e., fixing all manifestations of a fault in a code base. Recent studies of a range of code bases have reported the occurrence of bug fixes that are incomplete at rates of 9% [28], 12% [31], and more than 14% [50]. Analysis of this incompleteness stems from multiple sources, but all of these studies find that a failure to fully understand the failing program behavior is a significant contributing factor. A key step in repair, whether manual or automated, clearly involves a complete characterization of how a failure may be exhibited. CFCs compute exactly that.

To understand how this might be used, consider the recent automated fault repair technique Angelix [37]. Like most automated repair techniques, Angelix relies on a test suite both as the definition of the failing input space and as an encoding of the repair correctness criteria. Computing CFCs for failing tests would produce a safe characterization of the failing input space that can be used to direct Angelix's symbolic execution based repair method to produce more robust repairs.

b) Quantitative Program Analysis: Recent years have witnessed the development of quantitative program analysis techniques built on a combination of symbolic execution and model counting [9], [8], [23], [22], [34], [25]. These have been shown to be useful in computing quantitative information about program behavior ranging from system reliability estimates [23] to secure information flow and side-channel attacks [38].

These techniques work by exploring the symbolic state space of a program, formulating path conditions for reachable states, invoking exact or approximate model counting techniques to estimate the size of the input space satisfying those conditions, and then computing the appropriate quantitative metric, e.g., probability of failure, probability of information leakage, etc. Unfortunately, these techniques suffer from the limitations of all symbolic execution techniques—their abstractions are not well-suited for characterizing loops. This leads them to designate regions of program behavior that are not analyzed (e.g., because of bounds on the symbolic execution) as uncertain (e.g., "gray" nodes in [23]). ACF offers a means of addressing this uncertainty since $\overline{A}$ are designed to analyze looping behavior.

Moreover, a CFC, F, computed by ACF is well-suited to subsequent quantitative analysis. First, F is made up of disjoint elements, each of which can be counted independently

and the resultant counts can be summed. Given the complexity of model counting, this can offer exponential reductions in cost. Second, for a given element, $f \in F$, when $f.up = f.lo$ one need count only a single formula. Moreover, the coincidence of upper and lower bounds provides more precise information for the input sub-space characterized by such elements. Third and most importantly, a CFC guarantees that inputs in $\neg upper(F)$ do not lead to program failures, thus, the uncertainty of quantitative analysis results is due only to the gap in F's bounds, i.e., $\#(upper(F) \wedge \neg lower(F))$, which again can be computed on a per partition basis.

VII. Related Research

There has been significant recent interest in program analyses that mix may and must analyses, e.g., [43], [29], [2], [1], [18], [27], but these ideas go back more than two decades, e.g., [41], [20], [14], to seminal work on abstraction-based model checking [21], [19]. Perhaps the best known, and most influential, alternating analysis framework is *counterexample-guided abstraction refinement* (CEGAR) [14]. CEGAR performs a forward analysis using overapproximating abstractions of program states and when a potential error is detected, it switches to a backward underapproximating analysis whose results are used to refine the overapproximation for the next round. The ACF algorithm shares the basic alternation approach, but it only indirectly influences the abstractions used in an analysis through conditioning, e.g., predicate abstraction may key off `assume` statements. The analyses described above seek to either prove a program free of failures or produce a failure witness. In contrast, ACF accumulates and generalizes the set of witnesses until it can prove remaining program behavior free of failure, at which point it returns a CFC.

Our work builds on an increasingly sophisticated and cost-effective corpus of static analysis and verification tools. The 2017 Competition on Software Verification [46] involved 32 C static analysis tools. SV-COMP defines a rich error witness interchange format [7] which facilitates the combination of analyses in ACF. Our selection of CPA and UA was based largely on the robustness of those tools and their performance in the two most recent SV-COMP instances, but the tools in the competition represent an enormous range of technical approaches, and one could easily constitute an instance of ACF out of a much larger portfolio of these techniques.

A key aspect of ACF is the use of assumptions to block $\overline{A}$ from analyzing portions of the program state space. This is inspired by the conditional model checking approach developed by CPA [5]. In essence, the iterations of ACF build an increasingly general condition that blocks potentially failing behavior until the remaining behavior is failure-free. Unlike conditional model checking, ACF uses generalization (specialization) techniques to broaden (restrict) the upper (lower) bound of potentially failing program sub-spaces so as to compute safe CFCs.

We use the CIVL symbolic execution system for C [42] as $\underline{A}$ since it too participated in the two most recent SV-COMP competitions. Unlike the overapproximating analyses we used, we had to modify CIVL to incorporate it into ACF. This was primarily due to the need to direct the symbolic execution to a potentially failing sub-space of behavior. We note that this simply implements the idea of directed symbolic execution from the work of Ma et al [35]. We could have incorporated other underapproximating analyses into ACF, such as CBMC [15] or its successors, with appropriate support for direction.

From the CFC perspective, the most closely related work is from Kim et al. [31] and Gu et al. [28]. Kim et al. introduce the notion of "bug neighborhood" which is conceptually similar to the upper bound of a CFC, but it is formulated only in the case of null-reference failures, it does not characterize the failure space constructively in terms of the program input space, and it does not provide a definitive lower bound. Gue et al. introduce the notion of the "coverage of a fix", i.e., the extent to which a fix handles all inputs that trigger a bug. CFCs seek to accurately and safely characterize the failure to provide an upper bound on the required coverage for a fix. While Gu et al. rely on a bounded series of successively general under-approximations to maximize coverage, ACF uses alternation. This allows ACF to exploit the power of overapproximating abstraction to efficiently summarize non-failing program sub-spaces and, ultimately, converge with a safe upper bound.

VIII. Conclusions

This paper introduces the concept of a comprehensive failure characterization which provides a constructive formulation of the failing input space of a program. It also presents a general algorithmic framework for computing CFCs that exploits the power of alternating over and under-approximating static analyses along with refinment approaches. The ACF framework is guaranteed to compute safe CFCs and it incorporates several mechanisms to improve their accuracy.

Data from an exploratory study on a prototype implementation of ACF for C programs reveals that it is possible to compute CFCs with a good degree of accuracy in a matter of hours. Moreover, the study reveals multiple opportunities for optimization that offer the chance to significantly reduce ACF runtime and thereby enable more thorough generalization (by running more than one round) which will boost accuracy.

The ACF framework can incorporate a wide-variety of analyses and strategies and our future work will explore that space. We plan to enrich the analysis portfolio to include all of the SV-COMP competitors. We plan to explore additional generalization strategies including those that are based on semantics using, for instance, logical interpolation. We plan to apply ACF to compute CFCs for larger C subject programs with faults, such as those in the SIR repository (http://sir.unl.edu), and to integrate ACF as a front-end analysis to improve the performance and solution quality of synthesis-based program repair and quantitative program analysis.

References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From under-approximations to over-approximations and back. In *Proceedings of the 18th International Conference on Tools and Algorithms for the*

Construction and Analysis of Systems, TACAS'12, pages 157–172, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] A. Albarghouthi, A. Gurfinkel, O. Wei, and M. Chechik. Abstract analysis of symbolic executions. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 495–510, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 3–14, New York, NY, USA, 2008. ACM.

[4] D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *International Conference on Computer Aided Verification*, pages 622–640. Springer, 2015.

[5] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: a technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 57. ACM, 2012.

[6] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.

[7] D. Beyer and P. Wendler. Reuse of verification results. In *Model Checking Software*, pages 1–17. Springer, 2013.

[8] M. Borges, A. Filieri, M. d'Amorim, and C. S. Păsăreanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2015. ACM, 2015.

[9] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. *SIGPLAN Not.*, 49(6):123–132, June 2014.

[10] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[11] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. *Moving Fast with Software Verification*, pages 3–11. Springer International Publishing, Cham, 2015.

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[13] CIVL: Concurrency Intermediate Verification Language. https://vsl.cis.udel.edu/civl, Accessed May 1, 2017.

[14] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Counterexample-Guided Abstraction Refinement*, pages 154–169. Springer Berlin Heidelberg, 2000.

[15] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, pages 168–176. Springer, 2004.

[16] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. *The ASTREÉ Analyzer*, pages 21–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[17] CPACHECKER tool. https://github.com/dbeyer/cpachecker, Accessed Nov. 10, 2015.

[18] P. Daca, A. Gupta, and T. A. Henzinger. Abstraction-driven concolic testing. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 328–347, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[19] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, Mar. 1997.

[20] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[21] D. L. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 409–422, London, UK, UK, 1995. Springer-Verlag.

[22] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. Statistical symbolic execution with informed sampling. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 437–448. ACM, 2014.

[23] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in Symbolic Pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.

[24] K. Friedberger. Cpa-bam: Block-abstraction memoization with value analysis and predicate analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 912–915. Springer, 2016.

[25] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 166–176, New York, NY, USA, 2012. ACM.

[26] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[27] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 43–56, New York, NY, USA, 2010. ACM.

[28] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.

[29] W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 304–319, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] M. Heizmann, J. Hoenicke, and A. Podelski. *Software Model Checking for People Who Love Automata*, pages 36–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[31] M. Kim, S. Sinha, C. Görg, H. Shah, M. J. Harrold, and M. G. Nanda. Automated bug neighborhood analysis for identifying incomplete bug fixes. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 383–392, Washington, DC, USA, 2010. IEEE Computer Society.

[32] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[33] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[34] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Eng ineering*, ASE '14, pages 575–586, New York, NY, USA, 2014. ACM.

[35] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] P. D. Marinescu and C. Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[37] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[38] C. S. Pasareanu, Q. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 387–400, 2016.

[39] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 465–475, Washington, DC, USA, 2003. IEEE Computer Society.

[40] Polyspace code prover. https://www.mathworks.com/products/polyspace-code-prover.html, Accessed May. 1, 2017.

[41] D. A. Schmidt. A calculus of logical relations for over- and underapproximating static analyses. *Sci. Comput. Program.*, 64(1):29–53, Jan. 2007.

[42] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers. CIVL: The Concurrency Intermediate Verification Language. In *SC15: International Conference*

for High Performance Computing, Networking, Storage and Analysis, Proceedings, SC '15, Piscataway, NJ, USA, Nov 2015. IEEE Press. To appear.

[43] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. Alternate and learn: Finding witnesses without looking all over. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 599–615, Berlin, Heidelberg, 2012. Springer-Verlag.

[44] SV-COMP benchmarks. https://github.com/sosy-lab/sv-benchmarks, Accessed May 1, 2017.

[45] SV-COMP 2016 results. https://sv-comp.sosy-lab.org/2016/results/, Accessed May 1, 2016.

[46] SV-COMP 2017 results. https://sv-comp.sosy-lab.org/2017/results/, Accessed May 1, 2017.

[47] ULTIMATEAUTOMIZER tool. https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer, Accessed May 1, 2017.

[48] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 185–195. ACM, 2007.

[49] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Softw. Eng.*, 43(1):34–55, Jan. 2017.

[50] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.

[51] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: Better together! In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 145–156, New York, NY, USA, 2006. ACM.