Conditional Quantitative Program Analysis

Mitchell Gerrard, Mateus Borges, Matthew B. Dwyer, Antonio Filieri

Abstract—Standards for certifying safety-critical systems have evolved to permit the inclusion of evidence generated by program analysis and verification techniques. The past decade has witnessed the development of several program analyses that are capable of computing guarantees on bounds for the probability of failure. This paper develops a novel program analysis framework, CQA, that combines evidence from different underlying analyses to compute bounds on failure probability. It reports on an evaluation of different CQA-enabled analyses and implementations of state-of-the-art quantitative analyses to evaluate their relative strengths and weaknesses. To conduct this evaluation, we filter an existing verification benchmark to reflect certification evidence generation challenges. Our evaluation across the resulting set of 136 C programs, totaling more than 385k SLOC, each with a probability of failure below 10^{-4}, demonstrates how CQA extends the state-of-the-art. The CQA infrastructure, including tools, subjects, and generated data is publicly available at bitbucket.org/mgerrard/cqa.

Index Terms—program analysis, model counting, symbolic execution, conditional analysis, software reliability, software certification

1 Introduction

Modern safety-critical systems are software-intensive. While such systems undergo traditional verification and validation processes to detect and remove faults, they also go through a certification process that aims to demonstrate their absence. International standards for such systems establish requirements for certifying the software's contribution to overall system safety across a range of domains including: avionics [1], industrial robotics [2], personal care robotics [3], railway [4], automotive [5], and medical software [6]. Meeting these standards is essential, but they present substantial verification and validation challenges above and beyond those of traditional software [7].

Safety certification standards vary, but all represent a complex undertaking that includes, for example, demonstration of bi-directional traceability between requirements and implementation elements and achieving rigorous forms of implementation coverage. It comes as no surprise that the primary means of demonstrating that an implementation meets a safety requirement is achieved through testing. In fact, testing is used in myriad ways across the breadth of application domains and associated standards—Nair et al. [8] identify 13 different forms of testing evidence that can be incorporated into safety arguments. For example, structural coverage evidence, such as MC/DC that is required for avionics software [1], robustness evidence, such as that which is achieved using fault-injection to meet automotive standards [5], and reliability evidence, such as that which is required to certify functions to IEC 61508 safety integrity levels (SIL) [9].

The increasing cost-effectiveness of automated formal methods and static analyses has led certification standards, e.g., DO-333 [10], and researchers to explore the types of evidence they can contribute to safety arguments to complement evidence from testing, e.g., [11]. In the context of a safety argument, such methods tend to provide *all or nothing* evidence—they can prove a property, e.g., through sound overapproximating model checking [12], or they cannot.

In this paper, we investigate combinations of static analysis methods that can provide a more gradual, quantitative form of evidence that can contribute to safety arguments. Our work is motivated by Ladkin and Littlewood's call for the increasing use of statistical evaluation in the certification of critical software [13], [14]. Their perspective is motivated by the fact that IEC 61508 defines SIL levels in statistical terms, e.g., a SIL level 4 function has an average probability of failure of less than 10^{-4} per invocation,[1] yet few cost-effective test methods exist to directly provide such evidence.

The challenges of testing ultra-reliable systems have long been known. Butler and Finelli [15] observed that achieving confidence in a very low probability of failure requires an exorbitant amount of testing. This challenge has been mitigated to an extent by advances in underlying technologies, e.g., high-fidelity simulation systems that can run in faster than real time and that can be executed in parallel [16], yet testing for high, much less ultra, reliability remains a significant obstacle.

Our insight is that two complementary forms of static analysis, when combined synergistically, yield a cost-effective method for demonstrating that functions achieve extremely low probability of failure. For completing subjects across the study, one instantiation of this technique computes a mean and median probability of failure below 10^{-10} and 10^{-38}, respectively. This would be sufficient to easily discharge the evidentiary requirements for a low demand SIL 4 function.

- *M. Gerrard and M.B. Dwyer are with the Department of Computer Science, University of Virginia, Charlottesville, VA, 22904. E-mail: {mitchelljgerrard, matthewbdwyer}@virginia.edu.*
- *M. Borges and A. Filieri are with Department of Computing, Imperial College London, London SW7 2AZ, UK. E-mail: {m.borges, a.filieri}@imperial.ac.uk*

1. This is for functions that are invoked relatively infrequently which is referred to as *low demand* in the standard; functions invoked frequently or continuously frame requirements in terms of the number of failure-free hours of operation.

The first analysis targets the fact that a key component of the cost of reliability testing comes from the need to *resample* equivalent program behavior. It is not obvious that two inputs will lead to equivalent behavior from a black box perspective, but when testing is permitted to observe the internal behavior of software, equivalence can be detected. This is precisely what symbolic execution techniques do [17] and reliability-focused extensions to symbolic execution can quantify the probability mass of a set of equivalent inputs [18], [19], [20]. This allows a single non-failing test input to accumulate all of the probability mass associated with its equivalent behaviors, which can greatly accelerate the process of reaching a reliability threshold.

The second analysis targets the fact that when systems enter the certification process they have already been thoroughly validated [7]. Our insight is that in this setting one can formulate a sound static analysis to partition the program input space into two subspaces—one that *may lead to failure* and one that *definitely does not lead to failure* [21], [22]. The latter of these can be skipped entirely when performing the above reliability analysis and the former can be used to *condition* the application of the reliability analysis, allowing it to focus on a smaller region of program behavior to maximize its cost-effectiveness.

In this paper, we study how these two analyses can be blended to create a new form of quantitative static analyses that can produce *guaranteed bounds on the probability of violating a safety property*. Unlike statistical methods [23], [24], which can only produce a probabilistic *confidence* on the soundness of the results based on statistics on the outcome of many test runs of the program,[2] the static analyses we focus on in this paper provide mathematically sound *guarantees* on the bounds for the probability of violations, and thus meet the strict evidentiary requirements for the above standards, e.g., [10].

Quantitative static program analysis has been studied for more than two decades, e.g., [27], [28], but only recently have fully automated techniques been developed that can scale to non-trivial code bases. Researchers have built on developments in increasingly scalable path-sensitive analyses, e.g., [29], [30], [31], [32], and increasingly scalable techniques for model counting of logical formulae, e.g., [33], [34], [35], [36], [37], [38], [39], to produce several families of techniques which we term *probabilistic symbolic execution* (PSE) [18], [19] and *statistical symbolic execution* (SSE) [20].

These techniques hint at the potential of combining non-quantitative program analyses, like symbolic execution, with quantitative analysis techniques, like model counting. We take this a step further in presenting a novel algorithmic framework for *conditional quantitative program analysis* (CQA) that blends evidence from multiple static analyses to extend the scalability, accuracy, and applicability of quantitative program analysis.

The history of combining non-quantitative static analyses to improve cost-effectiveness dates back at least three decades, e.g., [40]. In recent work, the open-source ALPACA framework [22], [41] implements an alternating conditional analysis (ACA) that combines 9 different C static analyzers to precisely characterize the regions of a program's execution space that always satisfy (or always violate) a given property. Whereas individual analyzers may be limited in their ability to cope with aspects of a program or state-space structure, ACA harvests and blends their partial results to produce a comprehensive description of program behavior. The key to CQA is the insight that ACA-computed descriptions—rendered as logical constraints formulated over program input variables—can be leveraged to focus the application of different forms of quantitative analyses, which has the potential to make them more efficient and more accurate.

Understanding the potential improvements that the algorithmic variants of the CQA framework offer relative to existing state-of-the-art quantitative static analyses, such as [18], [19], [20], requires empirical evaluation. Unfortunately, no benchmarks exist that focus on the specific challenges in evidence generation for certification of safety-critical software systems.

Developing a broad and representative benchmark for this class of problems is a worthwhile pursuit, but in this work we only take a modest first step by customizing an existing verification benchmark—SV-COMP [42]. The benchmark is designed to stress automated static analysis and verification tools, but to reflect certification challenges, benchmark programs should exhibit low-probability property violations—like those that might slip through development into a certification process. In §4 we describe the systematic selection of 136 C programs, comprising more than 385,000 SLOC, for which the probability of a property violation is less than 10^{-4}. This threshold was chosen because it corresponds to the failure probability threshold required to meet IEC 61508's SIL 4 standard. As our evaluation reveals, CQA is capable of establishing a much lower probability of failure than the SIL 4 requirement and can produce probability guarantees that were previously thought to be completely infeasible to achieve [15], e.g., less than 10^{-35} in under 15 minutes on `Problem10_label48`.

The next section presents background and the prior work on quantitative and conditional program analysis on which we build. §3 presents the foundations of the CQA framework. §4 presents an evaluation that explores the algorithmic tradeoffs between CQA and existing approaches and demonstrates that CQA extends the state-of-the-art. We discuss related work in §5 and future work in §6.

2 Background

2.1 Reachability Guarantees

A program's semantics can be formalized as a transition system $\langle Q, q_0, \rightarrow \rangle$, where Q is a set of states (mappings from variables to values), $q_0 \in Q$ is the initial state, and $\rightarrow \subseteq Q \times Q$ is a transition relation between states and their possible successors. The *state space* is the set of all possible configurations in this transition system. The reachability problem is that of determining whether there exists a path in a program's transition system from the initial state to a target state that satisfies property ψ; we will call these target states ψ-states. Most program analysis questions can be framed as reachability problems [43].

2. For a property ψ, a probabilistic guarantee is of the form $Pr(p_{\neg\psi} \in [a, b]) \geq \delta$, where $p_{\neg\psi}$ estimates the probability of violating ψ and $\delta < 1$ is a *confidence* value bounding the probability of the produced interval being incorrect (e.g., [24], [25], [26]).

In this paper we will refer to two kinds of reachability guarantees using different terms that refer to these distinct kinds; we will group and italicize the related terms in this paragraph. The first kind of guarantee is based on an *overapproximation* of the program state space, and gives a proof that a ψ-state is unreachable. We refer to this as a *safety* proof, produced by a *may* analysis that reasons about the *necessary* conditions on reaching a ψ-state. The second kind of guarantee is based on an *underapproximation* of the state space, and gives a proof that a ψ-state is in fact reachable. This is a *soundness* proof, produced by a *must* analysis that reasons about the *sufficient* conditions on reaching a ψ-state. More formally, given a program p, the first kind of guarantee is equivalent to the statement $p \models \neg\psi$, while the second to $p \not\models \neg\psi$.[3]

While the classical reachability problem is given as an existential query (does a path exist or not?), this work considers its generalization: find *all* paths from q_0—the start state—that could reach a ψ-state. Because there may be many program paths that lead to a ψ-state, we will characterize these sets of paths in terms of *intervals*, defined in the following subsection.

2.2 Logical Intervals

Any given program can contain large state spaces, such that portions of it cannot be exactly characterized in an efficient manner. Overapproximating the state space can simplify the model of complex state spaces in ways that make this more amenable to efficient reasoning [45], [46]. In general, program analyzers can approximate a program's ψ-state reachability within over- and underapproximate bounds, which we define as a logical interval.[4]

Informally, a logical interval I is a characterization of the input subdomain that exhibits behaviors reaching ψ-states. This characterization consists of a *lower bound* ($\underline{I}$) guaranteed to be subsumed by a program's true ψ-state reachability, R_ψ, and an *upper bound* ($\overline{I}$) guaranteed to subsume R_ψ. A sufficient condition on p's inputs reaching a ψ-state is given by $\underline{I}$, while $\overline{I}$ comprises a necessary condition for the same. In the remainder of the text, we will use the term *interval* and logical interval interchangeably.

Definition 1 (Logical Interval)**.** *A logical interval, $I_\psi = [\underline{I}_\psi, \overline{I}_\psi]$, is a pair of logical predicates on the inputs that semantically bound a program's ψ-state reachability, R_ψ, such that $\underline{I}_\psi \Rightarrow R_\psi \Rightarrow \overline{I}_\psi$.*

This interval lies between two extremes: $\underline{I}_\psi = \overline{I}_\psi = R_\psi$ (most informative, exact characterization) and $[\mathit{false}, \mathit{true}]$ (non-informative).

A partition on I_ψ induces a set of *disjoint intervals*. For the remainder of the text, the ψ subscript will denote the logical interval over the entire program space, and those without the ψ subscript will denote disjoint intervals.

3. A reachability property is a property stating that a particular state *can* be reached, while a safety property states that a "bad" state is *never* reached; so reachability properties can be seen as the negation of a safety property [44]. Because the focus of this work is on characterizing reachable states, safety is discussed in a negated sense in relation to ψ.

4. A logical interval that semantically bounds program behavior is discussed in [21], where it is referred to as a *comprehensive failure characterization*.

Definition 2 (Disjointness)**.** *Two intervals are considered disjoint when their upper bounds are disjoint, i.e., $\overline{I}_i \wedge \overline{I}_j \equiv \emptyset$; because upper bounds subsume their lower bounds, all lower bounds will also be disjoint.*

Note that the upper bound of each individual disjoint interval is not a necessary condition on reaching a ψ-state over the entire program space: there may exist other intervals, so the negation of one upper bound would by definition include the other intervals. However, it *is* a necessary condition on reaching a ψ-state within its respective partition of the input domain. The lower bound of a disjoint interval, in contrast, is a sufficient condition in reaching a ψ-state both in the entire program space as well as within its respective partition of the input domain.

2.3 Conditional Program Analysis

The idea of conditional quantitative analysis (CQA) builds on the principle of conditional verification [47]. Conditional verification combines the strengths of multiple verification procedures by excluding the portion of state space that one procedure has verified as safe from the search space of the others. This allows each application of a verification procedure to focus only on parts of the state space which have not yet been covered by any of the others. Because most verification procedures are specialized (or optimized) on specific program features (e.g., via specific abstractions or memory models), composing their partial results may enable each to leverage their respective strengths on portions of a program exhibiting specific features, leaving simpler residual problems after each application.

2.4 Basic Probability Definitions

The possible outcomes of an experiment are called *elementary events*. For example, flipping a coin can produce one of two elementary events: heads or tails. Elementary events are mutually exclusive, and the set of all elementary events is called the *sample space*. An *event* is a set of elementary events.

Definition 3 (Probability distribution)**.** *Let Ω be the sample space of an experiment. A* probability distribution *on Ω is a function associating to each subset of Ω a real value between 0 and 1: $Pr : 2^\Omega \to [0, 1]$ that satisfies the Kolmogorov's probability axioms [48]:*

- *$Pr(e) \geq 0$ for every elementary event e*
- *$Pr(\Omega) = 1$*
- *$Pr(A \cup B) = Pr(A) + Pr(B)$ for all events A, B where $A \cap B = \emptyset$*

(Ω, Pr) is called the probability space*.*

Definition 4 (Conditional probability)**.** *Let (Ω, Pr) be a probability space. Let A and B be events with $Pr(B) > 0$. The* conditional probability *of A given B (i.e., the probability of A assuming B has occurred) is defined as $Pr(A \mid B) = \frac{Pr(A \cap B)}{Pr(B)}$.*

Definition 5 (Law of total probability)**.** *Let (Ω, Pr) be a probability space and $\{E_i \mid i = 1, 2, 3, \ldots, n\}$ be a finite partition of Ω, where $\forall i. Pr(E_i) > 0$. Then, for any event A, $Pr(A) = \sum_{i=1}^{n} Pr(A \mid E_i) \cdot Pr(E_i)$.*

The probability mass function yields the probability that a discrete random variable is equal to some value. The

probability mass of a set of values is the summation of the probability mass function applied to its elements. A logical formula is the characteristic function of the set of its models, thus the probability mass of a logical formula is the probability mass of each of its models.

2.5 Quantifying Logical Formulae

Given a logical formula and a probability distribution over the free variables in the formula, there are a growing number of cost-effective methods to estimate the probability mass contained in the formula. Some of these estimates are exact, e.g., when the formula lies in the domain of linear integer arithmetic [33]; in other cases the accuracy of estimates are probabilistically bounded [49], [50].

3 Conditional Quantitative Analysis

The problem this paper addresses is determining *how likely* it is that a ψ-state is reached within some program. Unlike the classical formulation of reachability, where either a path to a ψ-state exists or not, quantifying the probability of reaching a ψ-state requires considering many paths, in general.

One approach to solving the problem of how to quantify the probability mass of inputs reaching a ψ-state is via brute force, i.e., enumerate all program paths and sum the mass of those reaching a ψ-state, as proposed in [18]. Another approach is to fuzz the input space to get a statistical bound on the probability of reaching a ψ-state [51], [52], [53]. The first approach suffers when the state space is large, while the latter suffers when the probability of reaching a ψ-state is exceedingly rare.

The solution advocated in this paper is to first determine *which* regions of the input space can lead to a ψ-state, and only quantify this reduced portion of the program state space. We call this a *conditional quantitative analysis*.

Algorithm 1 defines the conditional quantitative analysis algorithm using the specified internal functions. CQA takes as input a program, a reachability property (ψ), and a probability distribution over the program's input variables; and outputs a quantitative characterization that bounds the input probability mass reaching ψ. The "lower" quantity (l) provides a sound lower bound on ψ-reaching inputs, i.e., l quantifies the sufficient conditions on inputs reaching ψ, while the "upper" quantity (u) provides a safe upper bound on ψ-reaching inputs, i.e., u quantifies the necessary conditions.

CQA begins by initializing the lower and upper quantifications to zero in line 2. The function *generate_intervals* on line 3 takes a program and a reachability property and returns a nonempty, finite set of intervals that describe the portions of the state space that *may*/*must* reach a ψ-state. The intervals must satisfy the safety and disjointness properties of Definition 1 and Definition 2, respectively. A trivial implementation of *generate_intervals* would return the set $\{[\mathit{false}, \mathit{true}]\}$, which contains a single interval that implies *all* program behavior—thus safely but trivially bounding ψ-state reachability. A more informative implementation of *generate_intervals* could, for instance, return the set $\{[\alpha \wedge \beta, \alpha], [\neg\alpha \wedge \gamma, \neg\alpha \wedge \gamma]\}$, which contains two intervals: the first denotes that a ψ-state *must* be reached when the program inputs satisfy $\alpha \wedge \beta$ (the interval's lower bound), and that a ψ-state *may* be reached when the program inputs satisfy α (the interval's upper bound); the second denotes an interval whose lower and upper bound coincide—this means that a ψ-state *must* be reached when the inputs satisfy $\neg\alpha \wedge \gamma$.

Algorithm 1 Conditional Quantitative Analysis

Input: Program P, reach. property ψ, prob. distribution X
Output: Lower/upper quant. of ψ-reaching inputs $[l, u]$

```
1:  procedure CQA(P, ψ, X)
2:      [l, u] ← [0, 0]
3:      I ← generate_intervals(P, ψ)
4:      for each I ∈ I do
5:          if I_ ≡ Ī then
6:              e ← estimate(Ī, X)
7:              [l, u] += [e, e]
8:          else
9:              [l, u] += quantify_in_bounds(P, ψ, I, X)
10:     return [l, u]
```

Specifications for CQA Functions

generate_intervals (P, ψ)
Input: Program P, reachability property ψ
Output: Set of disjoint logical intervals (see Defs. 1–2)

estimate (f, X)
Input: Logical formula f, probability distribution X
Output: Estimate of $Pr(f)$

quantify_in_bounds (P, ψ, I, X)
Input: Prog. P, reach. prop. ψ, interval I, prob. dist. X
Output: [Overappr. of $Pr(\underline{I})$, Underappr. of $Pr(\overline{I})$]

Lines 4–9 use these computed intervals to focus quantification efforts within the state space delineated by a given interval. There are two cases to consider when deciding how to quantify ψ-state reachability within an interval. In one case—line 5—, the lower and upper bounds coincide, so we can directly quantify the formula given by its upper (or equivalent lower) bound. This is done by the function *estimate*, which takes as input a logical formula and a probability distribution, and computes either an exact or a (probabilistically bounded) approximate estimate—e—of the probability mass that satisfies the given formula. In the case of coinciding bounds, the computed probability mass e is necessary *and* sufficient, so e is added to both the lower and upper quantifications in line 7.

The other possibility—line 8—is that an interval's bounds do not coincide, in which case we must explore the region of the state space between the lower and upper bounds in order to quantify the ψ-reaching probability mass contained within the interval. The function in line 9—*quantify_in_bounds*—takes as input a program, a reachability property, an interval, and a probability distribution, and returns a pair whose first part quantifies a safe overapproximation of the probability mass reaching $\underline{I}$—the lower bound of I, and whose second part quantifies a safe underapproximation of the probability mass reaching $\overline{I}$—the upper bound of interval I. The output in line 10 gives the lower and upper bounds on the probability mass of reaching a ψ-state.

Theorem 1 (Termination)**.** *Algorithm 1 terminates if generate_intervals, estimate and quantify_in_bounds terminate.*

Proof. The loop in lines 4–9 will run a bounded number of times because $\mathcal{I}$ is a finite set, so if each function terminates, Algorithm 1 will terminate. All functions called within CQA are required to terminate due to both time and space bounds, guaranteeing that CQA terminates. □

Theorem 2 (Correctness)**.** *Algorithm 1 terminates with l providing a sound lower bound and u a safe upper bound on the probability mass of a program reaching a ψ-state, given some input probability distribution.*

Proof. The correctness of CQA's output follows from four observations: (1) the function *generate_intervals* requires all program behavior reaching a ψ-state to be contained within $\mathcal{I}$, implying all probability mass of reaching a ψ-state is also in $\mathcal{I}$, (2) the same function requires the intervals of $\mathcal{I}$ to be disjoint, so no probability mass is quantified twice, (3) the functions *estimate* and *quantify_in_bounds* are required to yield a sound underapproximation and a safe overapproximation on the probability of reaching a ψ-state within the state space bounded by an interval, and (4) the estimates on each interval's probability mass are accumulated in l and u in either lines 7 or 9. So upon termination of the loop in line 10, l and u correctly provide lower and upper bounds on the probability mass of reaching a ψ-state. □

The remainder of this section discusses some of the possible instantiations of the functions used within Algorithm 1. These instantiations are used in the evaluation of CQA, discussed in §4.

3.1 Instantiation of *generate_intervals*

One non-trivial instantiation of *generate_intervals* is given by the framework of an *alternating conditional analysis* (ACA), which computes a sound characterization of all the ways a program either may or must satisfy some property. This is computed by alternating between over- and underapproximate analyses, conditioning analyses to ignore portions of the program that have already been analyzed, and combining the results of state-of-the-art analysis tools in a portfolio run in parallel. ACA is based on the framework introduced in [21], which was generalized in [22].

We will present the general idea of ACA by way of example. Given the program in Listing 1, ACA begins by running a portfolio of static analysis tools that search for a path reaching a call to `psi()`. Suppose a tool provides evidence of a path to `psi()`, then an underapproximate analyzer uses this evidence to characterize the path as either valid or spurious. This characterized space is now accounted for and does not need to be analyzed again.

Suppose the given evidence describes constraints on x that drive execution to one of the calls to `psi()`: $x < 0$. ACA now checks if there are other paths to `psi()`. To do so, blocking instrumentation is injected into the initial program—via `assume` statements—to condition analyzers to avoid this already-covered space.

```
main()
{
  x = read();
  y = read();

  if (x < 0)
    psi();
  elif ((x > 9) &&
        (x < y*y))
    psi();
}
```

Listing 1: Initial program

```
main()
{
  x = read();
  y = read();
  assume(!(x < 0));
  if (x < 0)
    psi();
  elif ((x > 9) &&
        (x < y*y))
    psi();
}
```

Listing 2: Conditioning 1

```
main()
{
  x = read();
  y = read();
  assume(!(x < 0));
  assume(!((x > 9) &&
           (x < y*y)));
  if (x < 0)
    psi();
  elif ((x > 9) &&
        (x < y*y))
    psi();
}
```

Listing 3: Conditioning 2

```
main()
{
  x = read();
  y = read();
  assume(!(x < 0));
  assume(!(x > 9));

  if (x < 0)
    psi();
  elif ((x > 9) &&
        (x < y*y))
    psi();
}
```

Listing 4: Conditioning 3

ACA runs the portfolio of analysis tools on the instrumented program of Listing 2. This time an overapproximate analyzer claims that a different path to `psi()` is reachable, with the given evidence of $x > 9 \wedge x < y * y$. A second `assume` statement is injected into the program, and the tool portfolio is run on the program in Listing 3.

This time around, an overapproximate analyzer—upon encountering the relational and nonlinear expression in the second `assume` statement—overapproximates the state space and declares that the call to `psi()` within the `elif` block is still reachable. An underapproximator deems this evidence spurious. In order to reach a fixed point, ACA now generalizes the second blocking clause by relaxing the constraint of $x > 9 \wedge x < y * y$ to be $x > 9$; this relaxed conditioning is shown in the second `assume` statement in Listing 4.

The tool portfolio is run on the program of Listing 4, and an overapproximate analyzer declares that `psi()` is unreachable given the conditioned program. Because the safety proof comes from an overapproximate analysis, it is assumed correct, and ACA terminates with two intervals describing inputs constraints leading to `psi()`. One interval has coinciding lower and upper bounds that exactly describe input constraints leading to `psi()`: $\underline{I}_1 \equiv \overline{I}_1 \equiv (x < 0)$, and another has noncoinciding lower and upper bounds: $\underline{I}_2 \equiv (x > 9 \wedge x < y * y) \Rightarrow \overline{I}_2 \equiv (x > 9)$. Note that, while $\underline{I}_2$ defines constraints on inputs that *must* reach `psi()`, this is not the case for $\overline{I}_2$, which includes concrete inputs that do not reach `psi()`, e.g., $x \equiv 10 \wedge y \equiv 3$. This simple example does not cover all cases of ACA; see [21] for an exhaustive case analysis.

The intervals computed by ACA satisfy the requirements

of *generate_intervals*, in that its output consists of a *lower bound* that is guaranteed to be subsumed by all reachable paths (the *must* information), and an *upper bound* that is guaranteed to subsume all reachable paths (the *may* information).

Across the 136 subjects in this study, the instantiation of *generate_intervals* returns intervals with noncoinciding upper and lower bounds on 129 subjects; 15 of these subjects are composed of multiple intervals. The upper and lower bounds coincide on the remaining 7 subjects, one of which is composed of multiple intervals.

3.2 Instantiation of *estimate*

Inputs can be assumed distributed uniformly over their domains or according to a given input distribution called a *usage profile* [19]. For simplicity, a uniform distribution over the input domains will be assumed throughout the paper; extension to arbitrary usage profiles is orthogonal to our contributions and can be straightforwardly implemented as in [19] and [54].

For a finite input domain D, computing the probabilities $Pr(c)$ of a constraint c can be reduced to computing the ratio between the number of solutions of $\sharp(c \wedge D)$ and the size of the domain $\sharp(D)$. Model counting procedures may in general be intractably complex [55]. Nonetheless, as with constraint solving problems, several algorithms are available for the efficient solution of specific fragments of the problem. Linear integer constraints can be efficiently and exactly solved using Barvinok's algorithm [33] (with off-the-shelf implementations including Latte [34] and Barvinok [36]). Nonlinear constraints over numerical variables can rely on progress in convex analysis [56], interval constraint paving [35], [57], and the approximate methods developed in both program analysis [54], [58], [59] and statistical machine learning [60]. Model counting over string domains includes exact counters for regular languages [38], exact bound computation [37], and mixed string/numerical counters [61].

More general—though usually more expensive—♯SAT and ♯SMT solvers also exist for model counting over mixed theories (e.g., [62], [63], [64]). The growing research interest in model counting for program analysis and artificial intelligence is driving a substantial research effort discovering new fragments of theories where efficient solutions are possible (e.g., [65], [66]) and are expected to directly benefit quantitative program analysis in the coming years.

As model counting is an orthogonal concern for CQA (equally impacting all the existing quantitative analysis techniques), for the implementations reproduced in this paper we will focus on linear integer constraints.

3.3 Instantiations of *quantify_in_bounds*

Following the principle of conditional program analysis, because all behaviors outside the logical interval of the entire program space $I_\psi = [\underline{I}_\psi, \overline{I}_\psi]$ have already been analyzed by *generate_intervals*,[5] quantification techniques can focus only on the residual behaviors, i.e., program paths satisfying the *assumption* $\alpha \equiv \neg\underline{I}_\psi \wedge \overline{I}_\psi$.

In probabilistic terms, this can be formalized as computing the conditional probability $Pr(I_\psi|\,\alpha)$, instead of $Pr(I_\psi)$, as the analysis is restricted to the subspace of the sample space that encloses all inputs satisfying α. Recalling Definition 4 (conditional probability), for each disjoint interval I_i and its corresponding assumption α_i we obtain:

$$Pr(I_\psi|\,\alpha_i) = \frac{Pr(I_\psi \wedge \alpha_i)}{Pr(\alpha_i)}\ . \tag{1}$$

The total probability $Pr(I_\psi)$ is the result of summing over the conditional probabilities multiplied by the respective $Pr(\alpha_i)$, as in Definition 5.

We now discuss three possible instantiations of *quantify_in_bounds*; the first being a direct application of model counting and the last two based on symbolic execution. Each satisfies the requirements of *quantify_in_bounds* in that it: (1) safely overapproximates its lower bound $\underline{I}$ and safely underapproximates an interval's upper bound $\overline{I}$, and (2) is amenable to conditioning. The second requirement is fulfilled simply by each technique respecting the semantics of `assume` statements. As the intervals of $\mathcal{I}$ are guaranteed to be disjoint, the conditioning comes for free, because each technique will reason only about the state space encoded by the disjoint formulae.

We will refer to CQA whose *quantify_in_bounds* has been instantiated with model counting, probabilistic symbolic execution, and statistical symbolic execution, as CQA$_\#$, CQA$_{pse}$, and CQA$_{sse}$, respectively.

3.3.1 Counting lower and upper bounds

The set of logical intervals $\mathcal{I}$ can be passed to a model counting procedure that converts its bounds into a numerical interval describing the contributions to the probability mass, i.e., by summing over the counts of the lower bounds and the counts of the upper bounds.

Applying model counting to an interval's lower bound and upper bound yields quantifications of these formulae that are either exact or (probabilistically bounded) over- and underapproximate estimates on the probability mass defined by $\underline{I}$ and $\overline{I}$, respectively; this satisfies the postcondition of *quantify_in_bounds*.

This instantiation of *quantify_in_bounds* is straightforward but can be very imprecise depending on the precision of the bounds. The potential imprecision can be improved upon by focusing underapproximate analyses within the lower and upper bounds. Any behavior that is analyzed within the interval is guaranteed to improve the bounds, e.g., if ψ is found, then the lower bound raises, and if $\neg\psi$ is found, the upper bound drops. Below we discuss two techniques that offer this kind of improved precision.

5. Recall that *generate_intervals* characterizes behaviors that *must* reach a ψ-state, i.e., $\underline{I}_\psi$; and upon termination guarantees that all inputs outside the upper bound, i.e., $\neg\overline{I}_\psi$, *must not* reach a ψ-state. These two behaviors lie outside the interval I_ψ, in that they lie "below" the lower bound and "above" the upper bound, and have already been characterized by some analyzer; so they may safely be ignored by *quantify_in_bounds*.

3.3.2 Probabilistic Symbolic Execution

Probabilistic symbolic execution (PSE) extends symbolic execution by computing the probability of each execution path being triggered by a program input [18]. As in standard symbolic execution, a program execution path (or program path) is uniquely identified by its path condition.

Program paths are classified in one of three ways, as: (a) reaching a target state (denoted with a ψ superscript), (b) missing a target state (denoted with a $\neg\psi$ superscript), or (c) truncated (denoted with a ? superscript) because the execution along a path failed to reach a target state within the prescribed depth or time limit. This classification of the execution paths induces a partition on the path conditions into three sets: (a) $PC^{\psi} = \{PC_1^{\psi}, \ldots, PC_i^{\psi}\}$, (b) $PC^{\neg\psi} = \{PC_1^{\neg\psi}, \ldots, PC_j^{\neg\psi}\}$, and (c) $PC^{?} = \{PC_1^{?}, \ldots, PC_k^{?}\}$. Because each path gives rise to a disjoint path condition, a lower bound on the probability of reaching a target state is given by

$$Pr^{\psi}(P) = \sum_i Pr(PC_i^{\psi}). \tag{2}$$

The probability of missing a target state, $Pr^{\neg\psi}(P)$, and the truncated probability, $Pr^{?}(P)$, have analogous definitions. As the union of path conditions induces a partition of all execution paths, the sum of these probabilities is 1, entailing that with any two of them the third can be computed arithmetically.

When the analysis of PSE is focused within an interval I, the path conditions within PC^{ψ} will raise the lower bound, or safely overapproximate $\underline{I}$'s probability mass; and the path conditions within $PC^{\neg\psi}$ will dually drop the upper bound by the probability mass contained in the set, so PSE safely underapproximates $\overline{I}$'s probability mass. Thus PSE satisfies the postcondition of *quantify_in_bounds*.

3.3.3 Statistical Symbolic Execution

PSE inherits the path explosion issue of symbolic execution, in addition to the cost of quantification procedures, which may prevent it from exploring the entirety of a program's executions.

Statistical symbolic execution (SSE) [20] addresses the problem of incomplete exploration by prioritizing the exploration of paths based on their probability mass. At each branch point, SSE computes the probability of moving towards each of the possible successor states by quantifying the solution space of the branch condition. The exact probability of a path is computable after its complete traversal.

As a sampled path is completely characterized by its path condition, it does not need to be sampled again, and can be pruned out of the sample space. This pruning allows for faster convergence of the statistical estimator to a prescribed accuracy, deterministically guaranteed termination, and more efficient coverage of rare events (i.e., execution paths with low probability).

The classification of program paths into three distinct sets is the same as with PSE, as is the way in which the probability mass within the sets PC^{ψ} and $PC^{\neg\psi}$ is used to satisfy the postcondition of *quantify_in_bounds*.

4 Evaluation

In this section, we explore the cost and effectiveness of conditional quantitative analysis (CQA) compared to the state-of-the-art—namely probabilistic symbolic execution (PSE) and statistical symbolic execution (SSE)—, as well as how bounds within CQA can focus further analyses. Our goal is to provide information about the runtime, accuracy, and cost of quantification across techniques, when applied in a context that captures challenges for evidence generation for safety certification of software components. To this end, we look at three research questions.

RQ1 *How cost-effective is CQA compared to the state-of-the-art in terms of runtime and accuracy of probabilistic bounds?*
RQ2 *How much quantification can be avoided using CQA?*
RQ3 *How does conditioning within CQA progressively focus quantitative analysis?*

4.1 Algorithm Implementations

To maximize consistency in our evaluation of different algorithmic approaches we implemented them on top of a common set of existing analyses. We use the open-source ALPACA from [41] since it is the only tool we are aware of that implements a nontrivial instantiation of *generate_intervals* for computing sound conditional information to drive CQA. We made minor modifications to invoke a model counter [36] to count computed intervals. ALPACA uses the CIVL symbolic executor for C programs [67]; it also enabled us to use a portfolio of 9 different analyzers that participated in the SV-COMP'19 competition for the synthesis of conditioning intervals, namely: CBMC [68], CPA-BAM-BnB [69], CPA-Seq [70], ESBMC-incr [71], PeSCo [72], Symbiotic [73], UltimateAutomizer [74], UltimateTaipan [75], and VeriAbs [76].

For consistency with ALPACA, our implementations of PSE and SSE both build on CIVL. PSE extends the default depth-first search in CIVL to report the current PC at the end of each path, which will then be categorized as either PC^{ψ}, $PC^{\neg\psi}$, or $PC^{?}$: PC^{ψ} for paths that end due to a call to the SV-COMP function `__VERIFIER_error()`, $PC^{\neg\psi}$ for executions terminating within the time bound without invoking the error function, and $PC^{?}$ for paths that hit the search depth limit. Only PC^{ψ} and $PC^{\neg\psi}$ paths are counted, since $Pr(PC^{?}) = 1 - (Pr(PC^{\neg\psi}) + Pr(PC^{\psi}))$.

SSE is implemented following the design in [20], by annotating each explored node of the symbolic execution tree with the fraction of the input domain reaching it (quantifying the path condition up to the node). Transitions during SSE exploration are randomly chosen according to the relative probability mass of the direct successor nodes, that is, for each direct successor, the ratio between the fraction of domain that can reach it and the cumulative fraction of the domain that can reach any direct successor. On backtrack, either due to termination of a path or reaching the depth limit, SSE subtracts the probability of the final path condition from each node along the way up to the root; nodes with a probability of 0 are pruned from the tree and thus will not be visited again (intuitively, the probability mass annotating a symbolic node represents how much of the executions through the node have not been explored yet; when 0, all such executions have been explored and

the node will not be sampled in subsequent runs). We use Barvinok [36] for model counting (default parameters, no timeout). Using Barvinok off-the-shelf limits our prototype to linear integer constraints.

Before counting, path conditions are first simplified using the Z3 solver [77] and then sliced into independent subproblems that do not share symbolic variables (following the slicing rules in [19]). The simplification step helps to mitigate the impact of Barvinok's internal transformation into Disjunctive Normal Form for inputs containing nested disjunctions.

4.2 Artifacts

All artifacts, including tools, subjects, and generated data are publicly available at bitbucket.org/mgerrard/cqa.

International standards establish *safety integrity levels* (SIL) for safety-related functions. For the highest integrity level the standard imposes a probability of violation per invocation of less than 10^{-4}—*low demand* mode for SIL4 [9].[6] The selection and filtering of subjects suitable for evaluating CQA was driven by this requirement that each subject has a probability of failure below 10^{-4}.

Our choice of building on ALPACA, which relies on analyzers that competed in SV-COMP, naturally led us to consider the SV-COMP benchmark suite [42] since the analyzers generally are able to process subjects in the benchmark and interpret SV-COMP annotation primitives, for example, `__VERIFIER_error()`, whose reachability defines ψ-states in our evaluation.

More specifically, we considered the sequential subjects from the benchmark that have property violations. From these we selected 1400 for which at least one of the 9 tools used in ALPACA could detect a violation within 15 minutes (SV-COMP competition timeout); note that this does not mean that ALPACA can characterize all violating inputs for subjects making it past this filter. From these, we filtered out subjects that could not be processed by the ALPACA, PSE, and SSE implementations: 412 subjects suffered from the inability to confirm a witness to a violation (a known problem in multiple SV-COMP tools), 34 subjects contained double or float data that is not supported by the model counters we used, 172 could not be handled due to incomplete support for C expressions in the ALPACA implementation, and 420 subjects could not be processed by CIVL because it either enforced strict C standards that were not met by the subjects, or the subject contained an unsupported feature. The remaining 369 subjects were run through ALPACA, PSE, and SSE implementations.

In specific domains or applications, inputs are usually assumed as generated by an input probability distribution. Unfortunately, this information is not available for any SV-COMP benchmark, even when the benchmarks are components of real software systems. Consequently, for our evaluation we assume a uniform input probability.[7]

For the remaining 369 subjects we ran the five considered quantitative analysis techniques to determine whether any could compute a lower bound on the probability of violation that exceeded 10^{-4}. Such subjects do not reflect the type of *rare* violations that make certification evidence challenging to produce, e.g., [13], [14], [15], so we removed them from our study. This resulted in a final set of 136 C subject programs which average 2833 SLOC—only 6 of these subjects have less than 100 SLOC and the largest is 9464 SLOC.

4.3 Results

We report the results of running CQA and (unconditioned) PSE and SEE on the 136 subjects in aggregated data; the full details of our experiments are included in the electronic appendix.[8] These details include a variety of measures that capture characteristics of the subjects and their analysis, for example, the number of paths explored, the number of gray paths whose exploration was truncated at depth bounds, and the share of analysis time spent in model counting. We reference these measures in the discussion of our results below.

Setup. We ran our analyses on a 2x 16-core Intel Xeon Gold 6130 server with 64GBs of RAM running Ubuntu Linux 18.04. We established a timeout of 90 minutes for running any of the implementations on a subject. It is not possible to determine the optimal depth limit for a given subject ahead of time, so for this evaluation we used a depth limit of 1000 symbolic states for both PSE and SSE.

Results overview. Table 1 provides an overview of our study results. There is a row for each algorithmic variant, where CQA has three variants depending on the technique used to instantiate *quantify_in_bounds*: $\text{CQA}_{\#}$, CQA_{pse}, and CQA_{sse}. The columns classify the performance of each technique by the number of subjects, out of 136, that share a particular outcome. The **Most acc.** $\underline{I}$ and **Most acc.** $\overline{I}$ columns report on the number of subjects on which a technique computes the most accurate lower bound, and upper bound, respectively. For the analyses that complete, we report the **Average** runtime in seconds. The final three columns report on different characteristics across runs. The **Complete** column lists the number of subjects for which the technique finishes the analysis without timing out or being depth limited. The **Timeout** column lists the number of subjects that the technique could not analyze within the 90-minute bound. The **Depth Limited** column lists the number of subjects for which the technique hit the depth limit—this only applies to analyses using PSE or SSE.

We note that a run of a technique may be both depth limited and timeout; for a given technique, the counts for complete, timeout, and depth limited need not sum to 136. If two techniques produce the same most-accurate bound (for either $\overline{I}$ or $\underline{I}$), then both are counted as having produced the most-accurate bound; thus the columns reporting on accuracy in the table do not sum to 136.

RQ1 (time/accuracy): CQA improves the state-of-the-art in quantitative analysis by computing more accurate probability bounds than previous techniques, at a comparable cost.

6. More stringent probabilities are required for *high demand* contexts.

7. Arbitrary discrete distributions can be reduced to mixtures of uniform ones over a partition of the finite domain, as in [19] therefore supporting arbitrary discrete profiles would add a linear complexity factor in the size of the partition; however, no input distribution is specified in SV-COMP.

8. Included with the submission and also available at bitbucket.org/mgerrard/cqa.

	Most acc. $\underline{I}$	Most acc. $\overline{I}$	Avg. Time (s)	Run characteristics		
				Complete	T/O	Bounded
CQA$_{\#}$	80	8	1269	133	3	0
CQA$_{pse}$	113	77	2488	40	32	76
CQA$_{sse}$	93	59	1418	21	57	63
PSE	51	54	1672	34	31	97
SSE	60	57	2886	11	54	76

Table 1: Summary of evaluation by technique

Across the study, both CQA$_{pse}$ and CQA$_{sse}$ produced more accurate lower and upper bounds than their unconditioned counterparts. All CQA variants produce a greater number of most-accurate lower bounds than the state-of-the-art. This is because the variety of analysis techniques used with *generate_intervals* can discover ψ-state reachability within state spaces in which unconditioned PSE/SSE find little to no ψ-state reachability.

With respect to the upper bound, the instantiation of *generate_intervals* used in this study computes upper bounds that, when quantified directly with a model counter, are too approximate to compete with the exhaustive techniques of PSE/SSE. Accordingly, CQA$_{\#}$ computes the most-accurate upper bound for the seven subjects in which the intervals given by *generate_intervals* coincide; the eighth subject is a degenerate case on which all techniques compute the same trivial $\overline{I}$. However, when *quantify_in_bounds* is instantiated with PSE and SSE, these approximate bounds allow CQA$_{pse}$ and CQA$_{sse}$ to produce 23 and 2 more most-accurate upper bounds than unconditioned PSE and SSE, respectively.

Both CQA$_{\#}$ and CQA$_{sse}$ complete in less time, on average, than the state-of-the-art. The average runtime of CQA$_{pse}$ is less time than SSE, but is nearly 14 minutes longer, on average, than PSE; the tradeoff for the longer runtime is an increase in the accuracy of bounds produced by CQA$_{pse}$.

Some of the increased accuracy in CQA$_{pse}$ and CQA$_{sse}$ is a result of the provided conditioning allowing these techniques to complete on more subjects than their unconditioned counterparts. The time spent in *generate_intervals* causes CQA$_{pse}$ and CQA$_{sse}$ to time out on 1 and 3 more subjects than PSE and SSE, respectively, but this time spent refining the conditioning allowed CQA$_{pse}$ and CQA$_{sse}$ to avoid being depth-bounded on 21 and 13 more subjects than PSE and SSE, respectively.

Though the CQA variants produce a greater number of most-accurate bounds than PSE and SSE, their strengths were found to be complementary across this study.

Figures 1 and 2 use linear diagrams [78] to depict the overlap between techniques for achieving the most-accurate lower and upper bounds on given subjects, respectively. A linear diagram is an alternative to Venn diagrams in showing the intersection between sets, in which sets are depicted as horizontal lines across the diagram, and their intersection is given by overlapping vertical segments. Each subject is demarcated by a vertical stripe. For example, the lefthand side of Fig. 1's linear diagram tells us that CQA$_{pse}$ alone computes the most-accurate lower bound for four subjects, then both CQA$_{pse}$ and PSE compute the most-accurate lower bound for the next six subjects, and so on. Note that the ith stripe in Fig. 1 does not necessarily

separate interval quantification. While it is possible to merge each separate interval into a single one and focus quantitative techniques within this larger interval—by quantifying within the space described by the disjunction of each upper bound and the negation of each of the lower bounds (see Sec. 3.3)—we observed a 2.9× speedup within CQA$_{pse}$ by quantifying each interval in parallel compared to doing so all at once.[9]

In terms of both runtime cost and effectiveness in computing accurate probability bounds, CQA is a clear improvement on the state-of-the-art. But the two need not be competitors. The complementary strengths of CQA variants and unconditioned PSE/SSE observed across this study recommends that both the conditioned *and* the unconditioned techniques should be applied, if possible.

RQ2 (counter calls): As pointed out in previous work on quantitative techniques [18], [20], in addition to the traditional cost of program analysis—e.g., constructing a program model, exploring its feasible branches, etc.—there is a significant component of the cost that goes into quantification, i.e., calls to a model counter. Being able to focus quantitative analyses on small subspaces of a program can significantly cut down on the number of calls made to a model counter. When an interval is exact, i.e., $\overline{I} \equiv \underline{I}$, this reduction can be drastic. (Note that, even if the probability mass within a noncoinciding interval is relatively small, this interval may still contain a large number of paths, necessitating a proportional amount of model count calls within *quantify_in_bounds*.)

We restrict this discussion to comparing techniques among common subjects that complete. When two techniques complete on a subject, they have produced the same probability bounds, so the overall work to compute the bounds is fixed, and we can compare quantification head-to-head, e.g., we can compare the exhaustive quantification done by PSE against that of CQA$_{pse}$, which divides its work amongst ALPACA and PSE within intervals. Comparing on completing subjects allows us to evaluate the effect conditioning has on the cost of quantification. The possible improvement in quantification is related to how the probability mass outside of the conditioned intervals is distributed across paths. If the probability mass is concentrated in a single path outside of the intervals, then the improvement within CQA will be negligible (a single call saved); but if there are many such paths then the improvement can be substantial.

On the single subject with noncoinciding intervals for which both SSE and CQA$_{sse}$ complete—`cdaudio_simpl1`—, SSE spends 1246 seconds issuing 48758 counter queries, while CQA$_{sse}$ spends 252 seconds issuing 18360 queries. For 24 of the subjects with noncoinciding intervals on which both PSE and CQA$_{pse}$ complete, CQA$_{pse}$ reduces the number of counter queries and counter time by 15%—with CQA$_{pse}$ issuing an average of 15060 queries in 1233 seconds, and PSE issuing an average of 17412 queries in 1424 seconds. The 25th subject with a noncoinciding interval on which both PSE and CQA$_{pse}$ complete—`floppy_simpl3`—is an outlier in that CQA$_{pse}$ issues 1750 queries in 3973 seconds, while PSE issues 1756 queries in only 368 seconds. The reason for the large difference in count times is a combination of the fact that this subject has a large number of symbolic variables and the conditioning includes a disjunctive formula, causing an exponential blowup in each query's clauses and slowing down CQA$_{pse}$'s overall quantification considerably. This is a scalability issue specific to the model counter we used for the experiments when handling disjunctive constraints in high-dimensional spaces. Different model counters may offer different scalability tradeoffs for different classes of constraints, e.g., [49].

When CQA provides intervals that *do* coincide, then a significant number of model counts can be avoided using CQA. This effect is observed on 7 of the 136 subjects in our study. On one of those subjects—`kbfiltr`—it is possible to characterize this reduction since CQA$_{\#}$, PSE, and SSE all complete; for the remaining 6, PSE and SSE do not complete. On the single subject for which CQA's intervals coincide and both PSE and SSE complete—`kbfiltr`—, the CQA techniques spend under 1 second issuing 4 queries, while PSE spends 123 seconds issuing 600 queries, and SSE spends 211 seconds issuing 16408 queries.

The relation between reduced calls to a model counter and reduced overall analysis time depends on the complexity of a program's constraints. The subjects in this study have only linear integer constraints, meaning programs containing more complex constraints will only further highlight the benefit of fewer calls to a model counter.

RQ3 (focusing): This research question will not compare against the state-of-the-art but will instead look at how the framework of CQA itself is used to progressively focus the accuracy of its computed logical intervals. We assume for this question that the computed intervals do not coincide, i.e., $\overline{I}_\psi \neq \underline{I}_\psi$, and that there is some uncertainty that can be progressively resolved. (If this were not the case, and the lower and upper bounds of I_ψ coincide, then the "focusing" is a few calls to a model counter, as discussed in RQ2.)

When two bounds of an interval do not coincide, this means there is some amount of probability mass implied by the upper bound of which we are uncertain, i.e., we do not know if this mass leads to a ψ-state or not. CQA can focus further quantitative analyses within $\neg\underline{I}_\psi \wedge \overline{I}_\psi$, as discussed in 3.3. Within this focused subspace of the subject, any probability mass proven to lead to a ψ-state effectively raises $\underline{I}_\psi$ (the lower bound), and mass proven to miss a ψ-state reduces $\overline{I}_\psi$ (the upper). We will refer to the reduction of uncertainty as *tightening* an interval. How do further analyses tighten an interval over time, e.g., is the upper bound first reduced, followed by the lower?; is the rate of tightening linear? do intervals containing less probability mass get tightened in less time than those containing more mass?

Figure 3 depicts how I_ψ is tightened from below and from above across a sample of four subjects[10] whose bounds

9. Merging disjoint intervals into a single one causes a blowup of clauses-to-quantify due to the presence of disjunctions in the merged upper bound as well as disjunctions resulting from negating the conjuncted lower bounds, increasing the cost of constraint solving and model counting in our experiments.

10. We chose these four signatures as representatives of others with similar visual patterns. All signatures can be viewed at bitbucket.org/mgerrard/cqa_signatures.

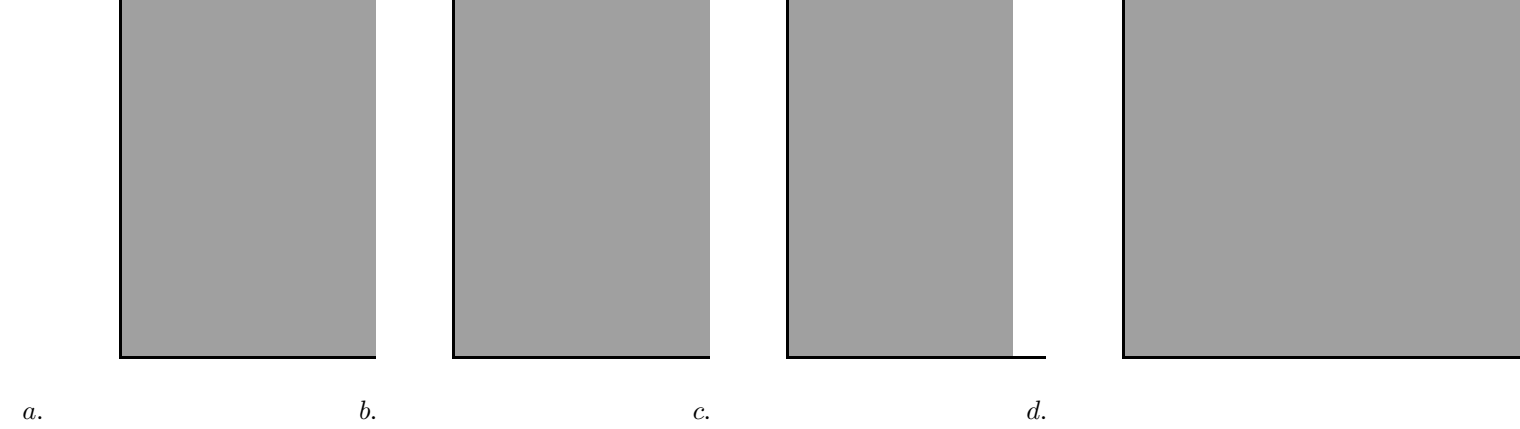

Figure 3: Signatures of conditioned PSE raising/reducing the lower/upper bound across different subjects.

do not coincide; we will call each depiction a subject's signature. The x-axis of a signature gives the running time of a quantitative analysis, from 0 to 270 seconds. The y-axis of a signature ranges from 0 to 1, and the gray areas represent the probability mass whose uncertainty has been left unresolved at a given time. The y-axis is depicted on a log scale in order to visualize miniscule changes in probability mass, ranging from 2.3×10^{-187} up to 1. This means that an upper bound of 1×10^{-11} will look the same as an upper bound of 0.9, so there can be a dramatic tightening of bounds that appears as slight reductions on the logscale plot, e.g., the upper bound of signature d is eventually lowered to 4.7×10^{-10}, though no shift in the upper bound is apparent in logscale. In Figure 3, the instantiated quantitative analysis is PSE. The visual steps are an artifact of PSE reporting its probability mass findings in 15-second intervals.

In the best case, a quantitative analysis will resolve all uncertainty, signified in a signature by the absence of gray after some time step. In the worst case, no uncertainty is resolved, and the gray area is not reduced at all.

The best case is visualized in both signatures b and c, though their respective uncertainties are resolved in different ways. In signature b, all probability mass collected accounts for inputs that do not lead to a ψ-state, and the upper bound is successively lowered. In signature c, probability mass accounting for both inputs that miss and inputs that lead to a ψ-state are collected in the first 15 second timestep, both raising the lower bound and lowering the upper bound; eventually just a sliver of uncertainty remains in a disjoint interval until PSE resolves this bit of probability mass.

Signatures a and d are examples of subjects whose uncertainty has not been resolved within a time bound. The uncertainty shown in signature a is tightened from above and below in distinct time steps, finally leaving a relatively small amount of probability mass unresolved. Signature d shows PSE raising the lower bound slightly, and, though not apparent with the log scale, the upper bound is lowered to 4.7×10^{-10}, but the remaining probability mass is left unresolved.

The diversity of signatures indicates how, in relation to some property, the resolution of a state space's uncertainty can occur in quite unpredictable ways. Some of this unpredictability occurs because certain portions of the state space contain more paths to explore than other portions; but part of the unpredictability comes from the fact that probability mass is not distributed evenly across paths.

The amount of probability mass contained in an interval is not related to the number of paths explored in this interval. For instance, one interval contains 6 paths whose probabilities sum to 2.3×10^{-9}, while another interval contains 7062 paths whose probabilities sum to the same amount. At the other end of the spectrum, one interval contains 16 paths whose probability mass covers most of the input space. This suggests that it is difficult to predict based on the amount of probability mass in unexplored intervals (given by: $(1 - \#\bar{I}) - \#\underline{I}$), how long such an interval will take to explore.

4.4 Discussion

This subsection is a more anecdotal discussion of observations culled from the study.

We observed that PSE and SSE are cost-effective when there are paths of modest number and depth whose constraints are amenable to efficient quantification. This was the case for 33 of the subjects in the study. These range from 1082 to 2726 SLOC with between 878 and 9032 paths—all of length less than 1000. A representative example is `email_spec8_product15`, which took 1996 seconds to analyze, of which 1568 seconds was spent in quantification procedures, and computed an *exact* probability of reaching a violation of 4.7×10^{-38}.

When a subject contains significantly more paths, as in `email_spec1_productSimulator`, a 3236 SLOC subject with at least 19705 paths, PSE times out. PSE faces challenges with subjects like `Problem01_label15` where paths are deep and quantification is expensive. On this 580 line subject, PSE explored 29562 complete paths, but was forced to abort the exploration of 42098 after hitting the depth limit and spent 84% of its time quantifying path conditions.

Like PSE, SSE also performs well on the subjects with small state spaces—a few thousand paths of depth less than 1000—but PSE always outperforms SSE on these cases (both explore the entire state space, but SSE has higher model counting overhead). The data reveal cases where SSE's ability to prioritize state-space exploration by probability mass has notable benefits. Both SSE and PSE timeout on `token_ring_08`. After analyzing 40999 paths, PSE is able to reduce the upper bound to 2.3×10^{-9}, but it does not find any paths to a ψ-state, and its lower bound is not raised at all. In contrast SSE, only analyzes 1275 paths before timing out, but is able to reduce the upper bound to 4.7×10^{-10} and raise the upper bound to 2.2×10^{-19}.

Many benchmarks in the study mimic the structure of embedded control system components. They include a top-level REPL that reads an input at each iteration, then applies a cascade of filters to the inputs to determine how to update its internal state, and finally executes an action based on the input and state. The subjects vary in the size of their internal state, the number of filters they apply, the nature of their state updates, and the specific location of the property violation. They range in size from 580 to 9464 SLOC and they have substantial state spaces, as evidenced by PSE's exploration of more than 100k paths prior to timing out on `Problem03_label43`.

Our manual analysis of representatives from this group of subjects reveals a common structure to their violations. Properties are violated only when sequences of values satisfying specific constraints are read during iterations of the top-level REPL. For all of these subjects, there is an iteration bound on the REPL beyond which no violation will be exhibited. This give rise to an unbounded nonviolating state space.

It is no surprise then, that on all of these subjects PSE and SSE either timeout or reach a depth limit. We note that for subjects like these, a depth-limited symbolic execution to handle infinite loops changes the semantics of such long running subjects and may lead to unusable results (i.e., there may be violations beyond the depth bound that would not be detected and quantified). While the maximum probability of such deep violations must be smaller than the probability of the gray paths, because their execution has been truncated, in REPL control loops the total mass of gray paths may be significantly large, preventing PSE and SSE from obtaining tight violation probability bounds within a feasible search depth.

While the CQA variants also cannot produce exact bounds on this group of subjects, the conditioning provided allows symbolic execution to avoid many unbounded non-violating state spaces—the number of depth-limited paths is always less for CQA_{pse} and CQA_{sse} than their unconditioned counterparts. In many cases, the reduction of depth-limited paths is drastic: on `Problem03_label26`, PSE is depth-limited on 64679 paths and yields an upper bound of 2.3×10^{-9}, while the conditioning given by *generate_intervals* allows CQA_{pse} to only hit 7176 depth-limited paths and yields an upper bound of 6.5×10^{-18}.

4.5 Limitations and Threats to Validity

Implementation. The main goal of this preliminary evaluation was to explore the capabilities of a proof-of-concept prototype of the mathematical framework behind CQA. Our implementation of PSE/SSE inherits all the limitations of the current version of CIVL's symbolic execution engine (e.g., strict conformance to C standards, limited support for non-integer domains, specific assumptions about the memory model) [79]. Our model counting interface delegates counting of linear integer constraints to Barvinok [36], after basic simplifications of the constraints via Z3 [77]; more advanced or specialized counting routines developed for established PSE/SSE analyzers may be faster.

Benchmark. Because there is no universally accepted specification format for properties and violation witnesses, to maximize compatibility with the tools in ALPACA we used the SV-COMP benchmark. Filtering out subjects not analyzable by our prototype tool implementations and with high violation probability, left a corpus of 136 programs that were sufficient to highlight limitations of all the approaches we considered. We remark that despite the limitations of the artifacts studied in this work, they have been able to confirm both the limitations and the potential of CQA techniques that were expected from their mathematical formalization.

We caution the reader in making conclusions about the external validity of our findings. While this corpus of programs may be a starting point, clearly a broader set of programs, ideally accompanied with realistic input distribution specifications, will be needed to construct a comprehensive benchmark for assessing quantitative analysis tools.

Internal validity. The ability to integrate available tools off-the-shelf [11] allowed us to develop prototype CQA implementations and assess their potential. However, the tools underlying ALPACA, PSE, and SSE are complex and highly-configurable. We use these tools in their default configuration and do not have control over their internals. We have not controlled for all of the factors that may influence their performance and this may impact the performance of CQA techniques. This is quite challenging and benchmarking the performance of static analysis tools and constraint solvers remains an open problem, e.g., [80], [81], [82], [83], [84]. Nevertheless, we took measures to cross check the probability intervals produced by all tools to confirm their consistency and we monitored for anomalies in underlying metrics reporting on the operation of the tools. After this check, we found no inconsistencies in reported bounds across the study.

4.6 A Benchmark for Analysis Techniques for High-Confidence Systems

The results of this study establish an absolute ground truth for 40 of the 136 subjects considered. These are the subjects on which some exhaustive technique (i.e., symbolic-execution-based) could complete. For those subjects on which none of the techniques could compute this ground truth, the probability mass of reaching a ψ-state is in general tightly bounded. There is a corpus of 135 examples on which the probability of reaching a ψ-state ranges from less than 4.7×10^{-10} down to 4.8×10^{-96}. (The 136th subject is a degenerate case in which all techniques compute the same trivial $\overline{I}$.)

Figure 4 depicts the least upper bound computed by any technique across subjects via an impulse plot. The y-axis gives the upper bound on the probability mass reaching a ψ-state in logscale, and each "impulse" on the x-axis represents one of the 135 nondegenerate subjects. The impulses are sorted by descending heights of least upper bounds, where the probability mass above each least upper bound represents the probability mass that is guaranteed to be safe. Fig. 4 demonstrates that the techniques in this study are able

11. Tools that participate in SV-COMP are able to interpret annotation primitives such as `__VERIFIER_error()`—whose reachability defines ψ-states in our evaluation, and `__VERIFIER_assume()`—which allows us to inject assumptions about program variables into the code. In this way, any analyzer competing in SV-COMP can be plugged into ALPACA off-the-shelf.

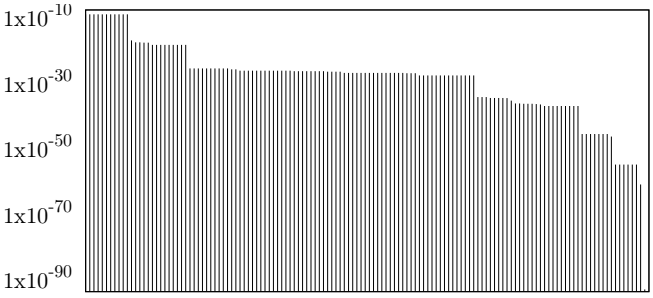

Figure 4: Least upper bounds per subject across techniques.

to compute highly accurate upper bounds on the probability of reaching a ψ-state for the given subjects, and we hope these bounds will soon be further lowered.

When analyzing the probability-of-failure in high-confidence systems, the upper bounds offer the most appealing guarantees, e.g., the assurance that your system will not fail outside of these bounds. For the state-of-the-art in quantitative analysis, the challenge and excitement lies in effectively reducing these upper bounds. While the techniques used within our study could not compute ground truths for all subjects, we hope this corpus will be used as a benchmark for other researchers to use in exploring analysis and testing for high-confidence systems.

5 Related Research

In this work we proposed a framework to use a combination of static analysis tools to synthesize conditions characterizing execution subspaces that *must* or *may not* reach a target state, and use this information to either bound the probability of reaching a target, or to condition subsequent path-sensitive quantitative analysis procedures for PSE or SSE.

Static program analysis of non-quantitative properties is a broad field, but we are applying the state-of-the-art rather than extending it. Our instantiation of *generate_intervals* using ACA (via ALPACA) provides us with ready access to a large and diverse set of analyzers [21], [22]. These include abstract interpretation based overapproximation tools, such as CPACHECKER, and SMT-based underapproximation tools, such as CBMC. The ACA framework builds upon research that combines may and must analyses [12], [85], [86], [87], [88], [89], [90]; applying these ideas is recommended in order to implement a nontrivial instantiation of *generate_intervals*. For CQA, the presence of overapproximating analyses is critical since they can summarize and classify entire sets of execution paths [91], which in turn can be quantified as a whole instead of requiring a more costly path-sensitive traversal of each path.

Conditioning is a method proposed in verification to combine the portions of state space confirmed as safe or failing using different techniques [47]. Researchers have proposed various methods to focus verification efforts on reduced portions of a program: passing state predicates between model checkers to restrict the considered state space [47], [92]; combining verification and systematic testing [93], [94]; transforming one program to another containing fewer execution paths while retaining the possible property violations of the original program [95], [96]. With CQA, we aimed at providing a mathematical foundation linking logical conditioning to conditional probability theorems, thus enabling the instantiation of conditional verification in the area of quantitative analysis.

Quantitative analysis in software engineering has historically been focused on the analysis of probabilistic abstractions of architectural models via probabilistic model checking [97] or on the definition of probabilistic abstract interpretation methods [28], [98], [99]. The former can take advantage of efficient probabilistic model checkers [97], but requires a manual construction of the models, which are difficult to keep consistent with code implementations. The latter proves difficult to effectively generalize to the constructs of modern programming languages (no tools exist for industrial-strength languages, to the best of our knowledge). Probabilistic symbolic execution [18] is a recent technique exploiting symbolic execution to extract conditions on the input leading to target states. We presented techniques in this family in Section 3. Variations of PSE and SSE have also been used for exact/approximate reliability analysis [19], [100], performance analysis [101], and detection and automated exploitation of side-channel vulnerabilities in both regular and probabilistic programs [102], [103]. In this paper, we focused on non-probabilistic programs; investigating possible extensions of CQA to general probabilistic programs [104] and approximate computing [105], [106] is left as future work.

Testing [52], [53] and classic underapproximating analyses aim at producing actionable evidence to drive the debugging process and their use is complementary to verification and CQA for certification as they cannot prove the absence of errors or sound bounds on the probability of error [51]. As already discussed, statistical techniques can be coupled with uniformly random testing to obtain statistical bounds on error probability (e.g., [24], [25], [26]), however, the number of runs required to achieve high accuracy and confidence bounds may be prohibitive and rare paths (e.g., guarded by an equality condition like `x==42`) are unlikely to be explored. As a white-box analysis, CQA pays the scalability cost of static analyses which can limit its applicability to larger problems, for which weaker statistical guarantees are the only viable alternative [107]. Despite their limitations, CQA techniques have proven applicable to components of up 9400 SLOC which is larger than component sizes suggested for safety critical systems [108].

6 Conclusion

We introduced *Conditional Quantitative Analysis*, which instantiates principles from conditional verification to allow logical evidence produced by non-quantitative static analyses to support more expensive quantitative analyses. CQA links logical conditioning used in verification to a conditional probability framework, enabling quantitative analyses to compute improved sound bounds on the probability of property violations. Our preliminary evaluation found evidence that CQA-enabled techniques extend the state-of-the-art in quantitative program analysis, highlighting the limitations and potential of several of them.

Our future work will seek to investigate further optimizations of the techniques enabling the CQA framework and their interactions, including the direct use of quantitative information within the iterative process of ACA to greedily drive logical interval synthesis and refinement towards efficiently accumulating violation evidence with the largest probability mass.

More broadly, we believe it essential for the research community to focus on developing mathematically well-founded analysis techniques to support the certification of software in critical systems. We plan to work with researchers and practitioners developing such systems to build a broader and more representative benchmark that can drive future research advances.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under grant numbers 1617916 and 1901769, by the U.S. Army Research Office under grant number W911NF-19-1-0054, and by the DARPA ARCOS program under contract FA8750-20-C-0507.

REFERENCES

[1] RTCA, "DO-178C : Software considerations in airborne systems and equipment certification," 2011.

[2] International Organization for Standards, "ISO 10218: Robots and robotic devices – safety requirements for industrial robots," 2011.

[3] ——, "ISO 13482: Robots and robotic devices – safety requirements for personal care robots," 2014.

[4] European Committee for Electrotechnical Standardization, "EN 50126: Railways applications – the specification and demonstration of reliability, availability, maintainability and safety," 2017.

[5] International Organization for Standards, "ISO 26262: Road vehicles – functional safety," 2011.

[6] International Electro-technical Commission, "IEC 62304: Medical device software software life cycle processes," 2006.

[7] M. P. E. Heimdahl, "Safety and software intensive systems: Challenges old and new," in *2007 Future of Software Engineering*, 2007, pp. 137–152.

[8] S. Nair, J. L. De La Vara, M. Sabetzadeh, and L. Briand, "An extended systematic literature review on provision of evidence for safety certification," *Information and Software Technology*, vol. 56, no. 7, pp. 689–717, 2014.

[9] International Electro-technical Commission, "IEC 61508: Functional safety of electrical/electronic/programmable safety-related systems," 2010.

[10] RTCA, "DO-333 : Formal methods supplement to DO-178C and DO-278A," 2011.

[11] D. D. Cofer and S. P. Miller, "DO-333 certification case studies," in *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, 2014, pp. 1–15.

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

[13] P. Ladkin, "Some practical issues in statistically evaluating critical software," in *10th IET System Safety and Cyber-Security Conference 2015*. IET, 2015, pp. 1–5.

[14] P. B. Ladkin and B. Littlewood, "Practical statistical evaluation of critical software," *electronic,(March 2015),[Online]. Available: http://www. rvs. unibielefeld. de/publications/Papers/LadLitt20150301. pdf*, 2016.

[15] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Trans. Softw. Eng.*, vol. 19, no. 1, pp. 3–12, Jan. 1993.

[16] J. Nutaro and O. Ozmen, "Using simulation to quantify the reliability of control software," in *2019 Winter Simulation Conference (WSC)*. IEEE, 2019, pp. 3267–3276.

[17] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[18] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 166–176. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336773

[19] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 622–631.

[20] 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*, ser. FSE 14. New York, NY, USA: ACM, 2014, pp. 437–448.

[21] M. J. Gerrard and M. B. Dwyer, "Comprehensive failure characterization," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 365–376. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155611

[22] ——, "ALPACA: A large portfolio-based alternating conditional analysis," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, 2019, pp. 35–38.

[23] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate probabilistic model checking," in *Verification, Model Checking, and Abstract Interpretation*, B. Steffen and G. Levi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 73–84.

[24] P. Zuliani, A. Platzer, and E. M. Clarke, "Bayesian statistical model checking with application to stateflow/simulink verification," *Formal Methods in System Design*, vol. 43, no. 2, pp. 338–367, Oct 2013.

[25] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification*, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 122–135.

[26] G. Agha and K. Palmskog, "A survey of statistical model checking," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 1, pp. 6:1–6:39, Jan 2018.

[27] G. Ramalingam, "Data flow frequency analysis," in *ACM SIGPLAN Notices*, vol. 31. ACM, 1996, pp. 267–277.

[28] D. Monniaux, "Abstract interpretation of probabilistic semantics," in *Static Analysis*. Springer, 2000, pp. 322–339.

[29] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 553–568.

[30] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[31] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy*, May 2012, pp. 380–394.

[32] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel, "CIVL: Formal verification of parallel programs," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2015, pp. 830–835.

[33] A. Barvinok and J. E. Pommersheim, "An algorithmic theory of lattice points," *New perspectives in algebraic combinatorics*, vol. 38, p. 91, 1999.

[34] V. Baldoni, N. Berline, J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu, "A user's guide for latte integrale v1. 7.2," 2014.

[35] L. Granvilliers and F. Benhamou, "Algorithm 852: Realpaver: An interval solver using constraint satisfaction techniques," *ACM Trans. Math. Softw.*, vol. 32, no. 1, pp. 138–156, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132973.1132980

[36] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Counting integer points in parametric polytopes using barvinok's rational functions," *Algorithmica*, vol. 48, no. 1, pp. 37–66, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1007/s00453-006-1231-0

[37] L. Luu, S. Shinde, P. Saxena, and B. Demsky, "A model counter for constraints over unbounded strings," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 565–576. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594331

[38] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 255–272.

[39] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser, "Compositional solution space quantification for probabilistic software analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594329

[40] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '85. New York, NY, USA: ACM, 1985, pp. 291–299. [Online]. Available: http://doi.acm.org/10.1145/318593.318659

[41] M. J. Gerrard, "Alpaca: An instatiation of the alternating conditional analysis framework," https://bitbucket.org/mgerrard/alpaca, Accessed June 15, 2019.

[42] "SV-COMP benchmarks," https://github.com/sosy-lab/sv-benchmarks, Accessed Aug 1, 2018.

[43] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[44] R. Malik, "Lecture notes in model checking," https://www.cs.waikato.ac.nz/~robi/comp552-07b/comp552-lecture10.pdf, September 2018.

[45] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[46] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge: MIT Press, 1999.

[47] 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*. ACM, 2012, p. 57.

[48] W. R. Pestman, *Mathematical statistics: an introduction*. Walter de Gruyter, 1998, vol. 1.

[49] M. Borges, Q.-S. Phan, A. Filieri, and C. S. Păsăreanu, "Model-counting approaches for nonlinear numerical constraints," in *NASA Formal Methods Symposium*. Springer, 2017, pp. 131–138.

[50] A. Biere and H. van Maaren, *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[51] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 621–631. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.68

[52] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[53] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.

[54] 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*, ser. ESEC/FSE 2015. ACM, 2015, pp. 866–877.

[55] L. Valiant, "The complexity of computing the permanent," *Theoretical Computer Science*, vol. 8, no. 2, pp. 189–201, 1979.

[56] B. Büeler, A. Enge, and K. Fukuda, *Exact Volume Computation for Polytopes: A Practical Study*. Basel: Birkhäuser Basel, 2000, pp. 131–154.

[57] J. Ninin, "Global optimization based on contractor programming: An overview of the ibex library," in *Mathematical Aspects of Computer and Information Sciences*, I. S. Kotsireas, S. M. Rump, and C. K. Yap, Eds. Cham: Springer International Publishing, 2016, pp. 555–559.

[58] M. Dyer, A. Frieze, and R. Kannan, "A random polynomial-time algorithm for approximating the volume of convex bodies," *J. ACM*, vol. 38, no. 1, pp. 1–17, Jan. 1991. [Online]. Available: http://doi.acm.org/10.1145/102782.102783

[59] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, "Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 447–458. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462179

[60] C. Robert and G. Casella, *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[61] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "Model counting for recursively-defined strings," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer, 2017, pp. 399–418.

[62] S. Chakraborty, K. S. Meel, and M. Y. Vardi, *A Scalable Approximate Model Counter*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 200–216.

[63] C. P. Gomes, A. Sabharwal, and B. Selman, *Model Counting*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 633–654.

[64] D. Chistikov, R. Dimitrova, and R. Majumdar, "Approximate counting in smt and value estimation for probabilistic programs," *Acta Informatica*, vol. 54, no. 8, pp. 729–764, Dec 2017.

[65] P. Hennig, M. A. Osborne, and M. Girolami, "Probabilistic numerics and uncertainty in computations," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 471, no. 2179, p. 20150142, 2015.

[66] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 3569–3576. [Online]. Available: http://dl.acm.org/citation.cfm?id=3061053.3061119

[67] 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*, ser. SC '15. Piscataway, NJ, USA: IEEE Press, Nov 2015, to appear.

[68] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *Proc. TACAS*. Springer, 2014, pp. 389–391.

[69] P. Andrianov, K. Friedberger, M. Mandrykin, V. Mutilin, and A. Volkov, "Cpa-bam-bnb: Block-abstraction memoization and region-based memory models for predicate abstractions," in *Proc. TACAS*. Springer, 2017, pp. 355–359.

[70] M. Dangl, S. Löwe, and P. Wendler, "Cpachecker with support for recursive programs and floating-point arithmetic," in *Proc. TACAS*. Springer, 2015, pp. 423–425.

[71] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer, "Esbmc 1.22," in *Proc. TACAS*. Springer, 2014, pp. 405–407.

[72] C. Richter and H. Wehrheim, "Pesco: Predicting sequential combinations of verifiers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 229–233.

[73] J. Slaby, J. Strejček, and M. Trtík, "Symbiotic: synergy of instrumentation, slicing, and symbolic execution," in *Proc. TACAS*. Springer, 2013, pp. 630–632.

[74] M. Heizmann, Y.-F. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler *et al.*, "Ultimate automizer and the search for perfect interpolants," in *Proc. TACAS*. Springer, 2018, pp. 447–451.

[75] M. Greitschus, D. Dietsch, M. Heizmann, A. Nutz, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski, "Ultimate taipan: Trace abstraction and abstract interpretation," in *Proc. TACAS*. Springer, 2017, pp. 399–403.

[76] P. Darke, S. Prabhu, B. Chimdyalwar, A. Chauhan, S. Kumar, A. Basakchowdhury, R. Venkatesh, A. Datar, and R. K. Medicherla, "Veriabs: Verification by abstraction and test generation," in *Proc. TACAS*. Springer, 2018, pp. 457–462.

[77] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[78] P. Rodgers, G. Stapleton, and P. Chapman, "Visualizing sets with linear diagrams," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 22, no. 6, pp. 1–39, 2015.

[79] M. B. Dwyer, J. G. Edenhofner, G. Gopalakrishnan, A. Marianiello, Z. Luo, Z. Rakamaric, M. S. Rogers, S. F. Siegel, M. Zheng, and T. K. Zirkel, "CIVL: The concurrency intermediate verification language reference manual, v1.19," https://vsl.cis.udel.edu/civl, Feb. 2019.

[80] D. Beyer, "Automatic verification of c and java programs: Svcomp 2019," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 133–155.

[81] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 291–302.

[82] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 53–68.

[83] M. B. Dwyer, S. Person, and S. G. Elbaum, "Controlling factors in evaluating path-sensitive error detection techniques," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, M. Young and P. T. Devanbu, Eds. ACM, 2006, pp. 92–104. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181787

[84] E. F. Rizzi, S. Elbaum, and M. B. Dwyer, "On the techniques we create, the tools we build, and their misalignments: A study of klee," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 132–143.

[85] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *International SPIN Workshop on Model Checking of Software*. Springer, 2003, pp. 235–239.

[86] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: a new algorithm for property checking," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 117–127.

[87] 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*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390634

[88] 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*, ser. POPL '10. New York, NY, USA: ACM, 2010, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/1706299.1706307

[89] 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*, ser. TACAS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 157–172. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28756-5_12

[90] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "Ufo: A framework for abstraction-and interpolation-based software verification," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 672–678.

[91] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 2005.

[92] D. Beyer and P. Wendler, "Reuse of verification results," in *Model Checking Software*. Springer, 2013, pp. 1–17.

[93] M. Czech, M.-C. Jakobs, and H. Wehrheim, "Just test what you cannot verify!" in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2015, pp. 100–114.

[94] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 144–155.

[95] K. Ferles, V. Wüstholz, M. Christakis, and I. Dillig, "Failure-directed program trimming," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 174–185.

[96] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim, "Reducer-based construction of conditional verifiers," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1182–1193.

[97] J.-P. Katoen, "The probabilistic model checking landscape," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '16. New York, NY, USA: ACM, 2016, pp. 31–45. [Online]. Available: http://doi.acm.org/10.1145/2933575.2934574

[98] D. Kozen, "Semantics of probabilistic programs," *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 328–350, 1981.

[99] P. Cousot and M. Monerau, "Probabilistic abstract interpretation," in *Programming Languages and Systems*. Springer, 2012, pp. 169–193.

[100] 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 Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 575–586.

[101] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884794

[102] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu, "Symbolic path cost analysis for side-channel detection," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 27–37. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213867

[103] P. Malacaria, M. Khouzani, C. S. Pasareanu, Q. Phan, and K. Luckow, "Symbolic side-channel analysis for probabilistic programs," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 2018, pp. 313–327.

[104] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 167–181. [Online]. Available: http://doi.acm.org/10.1145/2593882.2593900

[105] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 112–122, 2014.

[106] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain: A first-order type for uncertain data," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 51–66. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541958

[107] M. Böhme and S. Paul, "On the efficiency of automated testing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 632–642.

[108] A. Mayr, R. Plösch, and M. Saft, "Towards an operational safety standard for software: modelling iec 61508 part 3," in *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. IEEE, 2011, pp. 97–104.

Mitchell Gerrard is a PhD student at the University of Virginia, United States. His research interests include software analysis and verification, particularly in facilitating conditional verification between a diversity of analysis tools. His non-research interests include literate programming, functional languages and Shandeism.

Mateus Borges is a PhD student at the Department of Computing, Imperial College London, UK. His research interests include software engineering and program analysis methods to improve the accuracy, scalability and effectiveness of testing, debugging and quantitative verification techniques.

Matthew B. Dwyer is a professor and the John C. Knight faculty fellow in the Department of Computer Science at the University of Virginia, United States. His research interests include software analysis, verification and testing and his work in these areas has been recognized over the years with several test-of-time and distinguished paper awards. He is a Fellow of the IEEE and of the ACM.

Antonio Filieri is an assistant professor at Imperial College London, UK. His main research interests are in the application of mathematical methods for Software Engineering, in particular, Probability, Statistics, Logic, and Control theory. His recent work includes exact and approximate methods for probabilistic symbolic execution, incremental verification, quantitative software modeling and verification at runtime, and control-theoretical software adaptation. https://antonio.filieri.name.