Alternating Conditional Analysis

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

Mitchell J. Gerrard

August 2021

To Evan, Lola and Grandma Mac

Acknowledgments

Thank you to my advisor, Matt Dwyer, for many things: guidance, patience, inspiration, believing in me, fully funding my classes and research, teaching the art of hand-ground pour over coffee. Thanks for everything these past seven years, Matt.

Thank you to Evan Hemsley, my college roommate and best friend, for introducing me to the beauty in computer science. I remember standing in line waiting to enter Collins dining hall while Evan—his hair striking some fractal pose—explained merge sort for the first time, and his excitement resonated in me then and now.

Thank you to Professors Chuck Riedesel and Jitender Deogun, for encouraging me to pursue graduate studies, and mentoring me in graph and number theory. My first programming course was with Chuck; we used MATLAB, and every class was some permutation of brain teasers, non sequiturs, horrible puns (German sausage jokes are just the wurst), and lucid explanations of programming fundamentals. What a happy, Carroll-esque introduction to the magical incantations of symbolic logic.

Thanks to all my labmates at UNL and UVa, for working through research ideas, eating together during a long day in the lab, or just shooting the breeze; I couldn't have finished without you all. Special thanks to John-Paul, Eric, Jonathan, Mikaela, Kennis, James, Mouna, Mehrdad, Guolong, Dong, David, Tony, Ajay, Jared, Didier, Wayne, Matias, Natasha, Ellie, Brady, Justin, Nishant, Mateus, Jeremiah, Will, Carl, Meriel, Nora, Soneya, Trey, and Ashley. Thanks to the many researchers across the world who helped via email to fix some bug, to understand a paper, or to provide valuable feedback. Also thanks to all the wonderful nerds I met at the Marktoberdorf and Menlo College summer programs.

Thanks to those who hosted me outside of the U.S., namely Antonio Filieri at Imperial

College London, Jérémie Guiochet at the University of Toulouse, and Sylvie and Thierry in Bouxiéres-aux-Chênes. Thanks to all my roommates in the States: brudduh Eric, Mom and Dad, John and Lauren (and Herbert), and Robin.

A big thanks to the administrative and computing staff at UNL and UVa for taking care of many tedious and necessary details, especially Brandon, Shea, Deb, Sally, Shelley, Larita, Tyler, Rick, and Barbara. Thanks to my committee members for meeting multiple times, giving helpful suggestions, reading through this dissertation, and helping me to finish.

Thanks to friends and family for all the support through graduate school, and for only asking the vague "So, how's research going?" on the occasions when it seemed to be going well. Thanks to my brother-in-law, Brandon, for helping me early on with programming in Java. Thank you to Henry, a miniature chocolate schnauzer who was my constant companion for many years.

Thank you to Sebastian Elbaum for showing me how excellent research is done, for encouraging me, and for always asking the crucial questions. Thanks to Ari for helping put things into perspective. Thanks to my wonderful teachers preceding grad school, including Anne Cognard, Oona Eisenstadt, and Darryl A. Smith. Thanks to all the enchanters I communed with in the evenings—Nabokov, Melville, Sterne, Zitkála-Šá, the one whose words made the western welkin blush. Thanks to Don Knuth—the cynosure of this field—for sharing the idea of literate programming, and for kindly replying to each of my inquiries.

Finally, thank you Lola for all your love.

Abstract

We rely on safety-critical software, so judging its correctness is important. If a pacemaker exhibits buggy behavior, just how buggy is it? Will a patch to some bug also patch other program paths that lead to the same bug? How could we guarantee this? Currently, the tools we use to judge software correctness paint an incomplete picture of how a program's inputs relate to its behavior, by giving a rough binary judgment of "correct" or "not correct." But it is possible to combine the efforts of these tools to say *which* portions of a program are correct, buggy, or uncertain.

In this dissertation we develop a novel meta-analysis framework that generates more informative program correctness proofs by combining results from an algorithmically diverse set of program analyzers. To safely combine information from overapproximate and underapproximate program analyzers, we define the concept of a program interval, which encodes these two kinds of information in a way that can be shared with other analyzers. To compute this program interval, we employ multiple program analyzers as black boxes that can exchange analysis results, such that the results from one analyzer can condition another to avoid reanalyzing some part of the program. We alternate between the guarantees of different analyzers to construct a program interval, and we define a generalization mechanism to ensure convergence. The constructive characterization is given in logical formulae collected by a directed symbolic execution.

We evaluate this framework on a set of C benchmarks and a case study and find that program intervals can be computed in an efficient, effective, and safe manner. We use program intervals to improve on the state-of-the-art in quantitative program analysis in providing probabilistic guarantees for safety-critical software standards. We explore how a diversity of analyzers is used to construct program intervals, and employ the framework to perform modular analyses.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The concept of a program encapsulates a wide range of complexity—from a one line print statement to a sprawling codebase made from thousands of contributors. The range of possible program behavior is so broad, that determining an interesting program property, e.g., does the program terminate on all inputs, is undecidable for the general class of programs. It is remarkable, then, that we *can* determine interesting properties about many programs, and do so all the time with the help of program analyzers. A program analyzer is itself a program that takes another program as input and tries to determine interesting facts about it, such as guaranteeing that some program will not attempt to dereference a null pointer. These analyzers are used to verify the correctness of safety-critical software, and are also found as components of many everyday applications, such as compilers, and IDEs.

While program analyzers can help us discover facts about a program, analyzers currently paint an incomplete picture of how program behavior relates to the desired facts. That is, an analyzer will either: make a statement about all program executions; make a statement about a single program execution; or declare that it does not know. This picture leaves out all the descriptive variations that lie between characterizing all program executions as a whole and characterizing a single one. Some indeterminacy is unavoidable, i.e., no analyzer can be sound and complete, but a more complete picture could describe program behavior by relating portions of a program's input to one of these three characterizations of "fact holds," "fact doesn't hold,"

or "don't know." We can talk about these characterizations in terms of the conditions that lead to them. Why would this be helpful? By making explicit which portions of the program have been accounted for by some analyzer, we can increase our confidence in the correctness of some program; we can use conditions produced by one analyzer to confine the space of uncertainty to a smaller set of program inputs that can be targeted with more specialized analyses; we can use this comprehensive description to see how sets of inputs are related, e.g., there may be two distinct paths leading to an error state—calling for a more general bug fix.

A standard program analyzer such as a dataflow analysis cannot compute this comprehensive description of program behavior, in general. As dataflow analysis reasons about more program executions than are feasible, its determinations are an all-or-nothing affair: all executions are reasoned about, or no definite answer can be given. A different analysis technique such as symbolic execution reasons only about feasible program executions, and in general cannot reason about all possible executions, e.g., when exploring an unbounded loop. To obtain our desired comprehensive description, we need to alternate between analysis techniques that reason about overapproximations and underapproximations of a program's executions. Why do we need alternation? The underapproximate analyzer is able to demarcate exactly which portions of the program have been accounted for, and the overapproximate analyzer can possibly reason about the portions of the program that have not yet been accounted for. To avoid reanalyzing already-characterized inputs, we *condition* an analysis by restricting it from analyzing demarcated regions of its input domain.

Instead of creating a bespoke program analyzer to compute this comprehensive description of program behavior, we develop a meta-analysis framework into which widely varying analyzers can be plugged. This dissertation makes contributions to generating more informative program correctness proofs by combining a diverse set of program analyzers. The four main contributions of this dissertation are in:

- enriching the results of a program analysis to encode overapproximate and underapproximate information
- developing a novel program meta-analysis framework that combines artifacts produced by over- and underapproximate analyzers to compute such a result

- empirically evaluating that framework
- applying the framework to improve the scalability and accuracy of downstream analyses

Enriching analysis results

A program analyzer tries to determine if a program is correct with respect to some specification of correctness. There are three results given by a typical analyzer: yes, the program is correct; no, it is not correct; or, "I do not know." This ternary view cannot capture more nuanced descriptions of a program's correctness, e.g., that all program paths except one are correct. We introduce a description of program behavior that provides a direct characterization of which parts of the program conform to these three answers. That is—which portions of the program's input are guaranteed to be correct, which are incorrect, and which are undetermined. This description is given in terms of logical formulae that capture which parts of the program have been assessed by which kind of analyzer. We call such a characterization a *program interval*.

Developing a meta-analysis framework

To compute this characterization of program behavior, we develop the meta-analysis framework of ACA (alternating conditional analysis). ACA employs multiple program analyzers as black boxes that can exchange analysis results that are typed as overapproximate or underapproximate proofs of correctness. The results from one analyzer can condition another to avoid reanalyzing some part of the program. ACA dynamically decomposes the program being analyzed such that each decomposition can be explored by all analyzers in parallel. A stopping point is reached when all program behavior has been characterized. We specify interfaces to: share information between analyzers, encode analysis results into a program interval, and widen the scope of the interval if necessary. Previous analysis techniques can be recast as instances of ACA.

Empirical evaluation of framework

To evaluate ACA we implemented the framework in a Haskell tool named ALPACA. We instantiate ALPACA with program analyzers that reason about the correctness of C programs. We employ 9 state-of-the-art analyzers as black boxes, and run each in a parallel portfolio in search of some full or partial correctness guarantee. The implementation includes instantiations for central

ACA components, but is architected to allow different instantiations of subcomponents—e.g., generalization—to be easily substituted. We also implement an extension that allows for ACA to be run in a modular fashion.

ALPACA is run in an observational study of 380 C programs comprising 798,544 source lines of code from the SV-COMP benchmark suite. We also conduct a case study over the `chrony` codebase. We find that ACA is able to compute program intervals in an efficient, accurate, and safe manner.

Application to other analyses

The results of ACA—a program interval—can be applied in various downstream analyses. We look at one such application: using the program interval to focus the efforts of expensive quantitative analyses. A program interval allows us to factor out parts of the state space that do not need to be explicitly quantified, leaving a potentially much smaller portion to be handled by a fine-grained analysis. We find that using ACA as a preprocessing step can improve the runtime and accuracy of state-of-the-art quantitative techniques.

The remainder of the dissertation is organized as follows. Chapter 2 discusses related work; we focus on program analysis techniques and the kinds of proofs they provide. In Chapter 3, we describe how to combine overapproximate and underapproximate program analyzers into the framework of ACA. Chapter 4 reformulates the techniques of Chapter 2 as instances of ACA. We detail the implementation of the core components of ACA within the tool ALPACA in Chapter 5. Chapters 6, 7 and 9 evaluate ACA on a set of benchmarks, in a case study, and through the lens of algorithmic diversity, respectively. The application of ACA to techniques used to quantify program correctness is given in Chapter 8. Chapter 10 concludes and looks to future research directions. Appendices A and B provide details of the raw empirical data interpreted in Chapters 6 and 8, respectively.

Chapter 2

Related Work

In this section, we look at the theoretical foundations that our current work builds upon. The pieces to this foundation are imported from different decades and appear in dissimilar dress, but are all related under the category of *formal software analysis*. As defined in [87], this kind of analysis is "a mathematically well-founded automated technique for reasoning about the semantics of software with respect to a precise specification of intended behavior for which the sources of unsoundness are defined." Formal software analysis differs from more informal—though not unhelpful—techniques such as manual code reviews or scanning program text to reveal patterns correlated with errors.

We separate the formal analyses into three broad groups. The first is that of the "classical" analyses, which reason about either a superset (an overapproximation) or a subset (an underapproximation) of a program's possible executions. These analyses were first formalized in the 1960's through the 1980's. The second group consists of analyses that combine the overapproximations and underapproximations of the classical analyses in a principled, bespoke manner. These analyses appeared beginning in the late 1990's, largely due to efficiency advances in satisfiability solvers. The third group seeks to employ multiple distinct analyses either by defining external interfaces or by trying to predict which analysis is best suited to a particular program. This last group is the most recent, less than a decade old.

2.1 Definitions

The definitions in this section are adapted from [190]. The program analyses we discuss in this dissertation all rely on some notion of a *program model*.

Definition 1. *A program model M is a 5-tuple $(Q, \mathcal{A}, \rightarrow, s, F)$, where*

1. *Q is a set of nodes or program states*
2. *$\mathcal{A}$ is a set of actions modeling the effect of program statements*
3. *$\rightarrow \subseteq Q \times \mathcal{A} \times Q$ is a set of guarded transitions modeling control flow*
4. *s is the start state*
5. *F is the set of final states*

We consider a *program model* to be a synonym for both a *transition system* and a *guarded transition system*. The *state space* of a model the set of all reachable states from the start state s.

A program analysis will try to compute facts about this program model. For example, after traversing an edge guarded by the condition $(x < 0)$, some program analysis can reason that the value of x is negative in the updated state. For a program model, the set of all facts is usually structured as a lattice, where the partial order $\sqsubseteq$ relates how precise these facts are, e.g., if facts are propositions on an integer variable x, then $(x \equiv 1) \sqsubseteq (x < 5)$, because $(x \equiv 1)$ is less precise than $(x < 5)$.

The lattice operations $\sqcup$ (or *join*) and $\sqcap$ (or *meet*) constructs a least upper bound (or *lub*) and a greatest lower bound (or *glb*) from subsets of facts. The *join* operation can move elements "up" in the lattice of facts, i.e., make the analysis less precise; and the *meet* operation can move elements "down" in the lattice, i.e., make it more precise. For instance, if we want to join two facts collected on different paths through the program model—$x \equiv 2$ and $x \equiv 4$—we may move up in the lattice of facts to $(x \equiv 2) \vee (x \equiv 4)$, telling us that x has the value of either 2 or 4. The *join* and *meet* operators are analogous to the *union* and *intersection* of sets. When we refer to a function or operator being "monotone," we mean that it preserves ordering, i.e., if function f is monotonic, then $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$.

We consider the term "logical implication" to be equivalent to "entailment" and "logical consequence," which may be given by either of the symbols $\models$ or $\Rightarrow$. We use both symbols to ease readability when there are more than two statements being related. Logical implication is a relation between two statements P and Q saying that all models that make P true also make Q true, written as $P \models Q$, or $P \Rightarrow Q$. Because no models make the statement "*false*" become *true*, Q is the empty set when P is *false*. This differs from the meaning of "material implication."

We will be working with a *logical lattice*, which is a lattice whose elements are made up of logical formulae. The formulae are related in a lattice in which the ordering $\sqsubseteq$ is defined by logical implication. Supposing that formulae f_1 and f_2 describe constraints on the values of program inputs, the idea is that $f_1 \sqsubseteq f_2$ corresponds to:

$$\{i \mid i \text{ is input implied by } f_1\} \subseteq \{i \mid i \text{ is input implied by } f_2\}.$$

Intuitively, f_2 described a larger (or equal-sized) set of inputs than f_1. The maximal value in the lattice—denoted by the symbol $\top$—is "*true*", and implies all inputs, while the minimal values—denoted by $\bot$—is "*false*" and implies the empty set of inputs. The *join* and *meet* operators are defined as logical disjunction ($\vee$) and logical conjunction ($\wedge$), respectively.

Note that the conventions we use here are the duals of conventions used in dataflow analysis [167] or denotational semantics [168]. That is, in dataflow analysis, the $\bot$ value conveys no information and the analysis becomes more precise as you move up the lattice. We adopt the conventions of abstract interpretation (described in Sec. 2.2.1), in which $\top$ conveys no information, and precision is added as you move down the lattice.

2.2 Classical Analyses

A program analysis tries to determine if a program satisfies some property. In Schmidt and Steffen's paper "Program Analysis *as* Model Checking of Abstract Interpretations," [190] the machinery of a program analysis is placed together in a comprehensive three-step process: first a program model, M, is constructed; second, M is abstracted to M' by reducing the amount of detail in its nodes and edges; finally, M' is checked to see if some property holds—if so, the

property holds in M.

A large number of program analysis problems can be transformed into reachability problems within a given program model [178]. The reachability problem is that of determining whether there exists a path in a program's transition system from the initial state to some specified state.

2.2.1 Overapproximations; or *may* analyses

The distinguishing characteristic of the analyses discussed in this subsection are that, given a program model M, they reason about an overapproximation of M, call it $\overline{M}$. This allows these analyses to collect facts that are true about all program executions in $\overline{M}$, and thus in M as well. Information computed over overapproximate program models is called *may information*, and *may analyses* are algorithms that compute just this. There are many variations of *may* analyses, but we will focus on three: abstract interpretation, model checking, and deductive methods.

Abstract interpretation

Abstract interpretation [69] is the general framework for reasoning about an overapproximation of a program's state space. Analyzing the overapproximated state space is handy because it is often smaller and simpler, and therefore easier to reason about; and if you can show a property holds in the overapproximated state space, then it also holds in the state space of the original program.

In its classical form, the algorithm for computing an abstract interpretation boils down to three steps:

1. define the abstract domain
2. define the abstract semantics
3. iterate over this abstracted transition system until reaching a fixed point

Steps one and two involve defining Q' and $\rightarrow'$ as abstractions of Q and $\rightarrow$ of our transition system in Def. 1. How should we define these abstractions? This depends on the property you would like to verify.

Suppose you want to know *whether some variable was always nonnegative* in some program. Let's first consider abstracting the domain for this problem, which should reduce the number of states to consider. Instead of keeping track of all possible integer values a variable can take, you only need to keep track of each state variable being negative or not, along with some value representing the fact that you do not know. This reduces the possibility of variable values down from the size of the integer domain to just three values; we lose information about the precise original values, and keep only the information necessary to our analysis, making this a "good" abstract domain.

In designing an abstract domain, there are important properties to preserve, such as: representing the domain as a lattice with either a *join* or *meet* operation defined, defining an abstraction function to map an element in the abstract domain to a set of elements in the concrete domain, and defining a concretization function to map a concrete element to an abstract element. The precise definitions and algebraic properties are out of the scope of this work (see [167, 70, 68] for details), but it suffices to know that this framework computes sound *may* information.

Abstract semantics define the mapping between abstract elements in our domain after applying some program statement. In the context of our sign domain, a decrement operation would map a negative value onto a negative value, but after decrementing a nonnegative value, it may still be nonnegative or it may be negative, so the nonnegative value is mapped to the maximal value in the lattice: $\top$. The $\top$ value signifies "I do not know what this concrete value could be." The main requirement in designing the abstract semantics is that if s' is an overapproximation of s, then the abstract interpretation of s' is an overapproximation of the concrete interpretation of s.

Once the abstract transition system has been defined, then, starting from the initial state, you make repeated transitions in the system until you have reached a "fixed point," meaning the states do not change after applying additional transitions. (We assume the domain and semantics have been defined in a way that ensures that the analysis completes.) After a fixed point is reached, you can make a judgment about how the property relates to the overapproximated—or abstractly interpreted—state space.

Model checking

Model checking was developed in the 1980's [90, 62, 63] as a way to verify the model of some system through an exhaustive state space exploration. While model checking can reason about properties in a variety of logics, here we consider the simple, but useful, case of safety properties [14]. Reasoning about safety properties reduces to computing reachability, i.e., to prove a safety property holds search for an unsafe state in an overapproximate program model and if you fail to find one you have succeeded in your proof. To guarantee termination, the reachable state space of the model must be finite. If no property violation is found, the system is proven correct with respect to the model and the property. If a violation *is* found, this could represent either a true property violation in the system, or a spurious violation existing in the abstracted system but not in the concrete one.

The structure can be explored by any graph traversal algorithm such as a depth-first search or a breadth-first search, backtracking when the same state is seen. Model checking is a path-sensitive analysis, as it does not summarize information computed on different paths (as an abstract interpretation would do when computing the abstract semantics of an `if` statement—summarizing the information computed over the `then` and `else` blocks with a *join* operation), but considers all paths independently. For this reason, model checking excels at discovering property violations in concurrent systems with many possible execution interleavings, as each combination is explicitly checked.

Path-sensitivity comes at a cost when a system with a large number of branches leads to an exponential number of states to be checked; this is known as the state-space explosion problem. One way to mitigate the blow-up is by creating a more abstract model with fewer transitions, though this can introduce an undue number of spurious violations. Automated methods to deal with spurious violations were introduced later and are discussed in Sec. 2.3.1.

Deductive methods

Deductive methods date back to the earliest work in formal software analysis. Precursors to deductive methods from the 1940's include Turing's embedded assertions in "Checking a Large Routine" [201] and von Neumann and Goldstine's assertion-annotated flow charts [105]. But

it was not until the late 1960's that Floyd and Hoare formalized the meaning of program correctness [96, 124].

Using the logic developed by Hoare, any program command C can be described by a triple

$$\{P\}C\{Q\}$$

where P and Q are formulae in predicate logic that describe properties of the state expected to hold before and after the execution of C, respectively. P is called the *precondition*, and Q is called the *postcondition*. A triple is considered *valid* if all states satisfying P are transformed by C into a state satisfying Q, upon C's termination. Given inference rules for each command in a programming language, these rules can be composed together such that pre- and postconditions can be given for each command in a sequence of commands—yielding input/output summaries across all execution paths.

As the postcondition of a branching command, e.g., an `if` statement, must summarize executions for each branch, deductive methods are not path-sensitive. This can lead to analysis imprecision that may be mitigated by adding stronger assertions at certain program points, such as at loop headers. These assertions are often manually added, though there are techniques that can automate much of this invariant discovery [145].

2.2.2 Underapproximations; or *must* analyses

The distinguishing characteristic of the analyses discussed in this subsection are that, given a program model M, they reason about an underapproximation of M, call it $\underline{M}$. This allows these analyses to collect facts that are true about specific program executions in $\underline{M}$. Because M contains all executions that are in its underapproximation $\underline{M}$, these represent feasible executions. Information computed over underapproximate program models is called *must information*, and *must analyses* are their corresponding algorithms. Here we focus on two *must* analyses: symbolic execution and bounded model checking.

Symbolic execution

Symbolic execution [135, 64] is a way to step through the execution paths of a program using symbolic values instead of concrete inputs. It can be formulated, albeit loosely, as an abstract interpretation where the abstract domain is composed of logical formula over free variables capturing input values, and the abstract semantics are defined as the concrete semantics; the requirement of reaching a fixpoint is relaxed, meaning symbolic execution will not terminate for unbounded state spaces. If the entire state space *can* be exhaustively explored, then a fixpoint is reached and the analysis soundly approximates the program semantics. A more formal treatment of the connection between abstract interpretation and symbolic execution is given in Section 3.4.5 of [67]. The following paragraphs give a more traditional formulation of symbolic execution.

A program's transition system is interpreted symbolically by maintaining a state that holds a mapping from variables to symbolic expressions, and a path condition (PC), which is a quantifier-free first order formula over symbolic expressions. The PC describes constraints on inputs that cause execution to flow down a given path.

The PC is initialized to *true*, meaning all input values are initially unconstrained. When a branch with condition c is encountered, symbolic execution explores both possible paths, using a SAT solver to check if the formulae encoding the paths are satisfiable. If any PC is found to be unsatisfiable, symbolic execution halts along that path. Let's see how this works on a small example.

Consider the control flow representation in Figure 2.1 of some program, where nodes are program locations and edges are annotated with assumptions on the state that must hold for the transition to occur. (Note that a transition system can be derived from this, given an initial state.) In Figure 2.1, there are two distinct paths from l_0 to l_5.

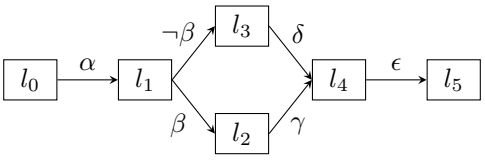


Figure 2.1: Example control flow graph

A symbolic execution over the control flow graph of Figure 2.1 begins by setting $PC = true$,

and exploring the first transition, so now $PC = true \wedge \alpha$; a SAT solver checks the satisfiability of PC—and α is found to be satisfiable. Now a branch condition at l_1 is encountered, so two separate PCs are enumerated (eliding the trivial *true* clause), namely $PC_1 = \alpha \wedge \beta$ and $PC_2 = \alpha \wedge \neg\beta$; and satisfiability is checked for each. In this way, symbolic execution will systematically enumerate the conditions describing each feasible path in a program, i.e., $PC_1 = \alpha \wedge \beta \wedge \gamma \wedge \epsilon$ and $PC_2 = \alpha \wedge \neg\beta \wedge \delta \wedge \epsilon$.

The advantage of using symbolic execution is that any path determined to violate a property is guaranteed to be a feasible one; you do not have to deal with spurious counterexamples. The disadvantage is that a program containing unbounded loops or recursive function calls will be explored by symbolic execution indefinitely, creating an unbounded number of PCs to check. In practice, symbolic execution engines are bounded in their search in some way, e.g., by number of paths or by a timeout.

Bounded model checking

Bounded model checking [59, 32], or BMC, involves considering the semantics of traces that are bounded in length by encoding them as one big logical formula related to a property, and asking a SAT solver if any of the considered traces violate that property.

BMC is similar to symbolic execution, but instead of enumerating all program paths and solving the constraints for each of these paths, BMC will encode all possible program paths (up to a depth bound) into a single formula. This often pushes the computational complexity of exponential branching behavior into the backtracking search used within constraint solvers. So in the control flow example from the previous section, instead of creating two separate formulae, BMC will stuff the branching behavior following l_1 into a disjunction; and the formula

$$\alpha \wedge ((\beta \wedge \gamma) \vee (\neg\beta \wedge \delta)) \wedge \epsilon \wedge \psi$$

is checked for validity to see if some property ψ is satisfied along all (bounded) paths.

Many programs have traces that are unbounded in length, so this approach cannot provide a safe overapproximation as abstract interpretation or CEGAR-based techniques can. Even with a bounded exploration, some transition systems generate formulae too large or complex for a SAT

solver to efficiently solve. But in many cases this technique is effective in finding feasible traces of the system that violate some property, also known as bugs.

One technique used to avoid dealing explicitly with the large formulae that can come with fully unwinding program loops is called k-induction [191, 81]. This technique is analogous to traditional mathematical induction, where you try to prove—using bounded model checking—that a property holds for the first k states (this is the base case), and then prove via induction that this property must hold in the next state. This technique allows BMC-based analyzers to safely prove unreachability of a property.

2.3 May-Must Combinations

The classical analyses overviewed in Sec. 2.2 were categorized as either *may* or *must* analyses. These categories correspond to the approaches' respective strengths: *may* analyses work best when proofs of unreachability can be effectively computed, while *must* analyses work best when proofs of reachability can be effectively computed. But what happens when neither types of proof are easily found?

A *may* analysis can produce an inordinate amount of false positives, i.e., reachability traces in an abstract model that do not correspond to concrete program traces. Each of the reachability traces must be checked by hand, and if a large portion consists of false positives, a frustrated user will stop using the analysis. On the other hand, a *must* analysis may exhaust its resources searching for a reachability proof in a large state space, and return with an unknown result. Perhaps setting a larger resource bound will lead to the discovery of a reachability proof, but this hope is uncertain. The imagined user can become discouraged with the limits of a pure *must* analysis.

Noting the fundamental limitations of these two "pure" approaches, researchers in the last few decades have attempted to combine *may* and *must* analyses such that their respective weaknesses can be mitigated by the others' strengths. In this section, we look at how these combinations are made and what guarantees they provide. We group the *may-must* combinations broadly into: counterexample-guided abstraction refinement, verification techniques followed by validation, and finally, approaches that employ the combinations in a more synergistic fashion.

2.3.1 Counterexample-guided abstraction refinement

In order to ensure a fixed point is reached in a program analysis, the abstract domain must be sufficiently coarse, but it must also be fine enough to prove the property-to-be-checked. Notice that in the classical formulation of an abstract interpreter, after the abstract domain is defined, it does not change; so if the domain is too coarse, the abstract interpreter may report many false positives.

One technique to use these false positives in order to find a sufficiently-precise abstraction is called Counter-Example Guided Abstraction Refinement, or CEGAR [60], which is an automatic method for focusing the precision of an abstract domain. The idea behind CEGAR is to start with a very coarse abstract domain, and refine this domain only if this abstracted transition system produces a violation trace proven to be infeasible in the concrete semantics.

CEGAR is often used with a domain modeled by predicate abstraction, where the state variables are related to a finite set of predicates. In the preceding section, each state variable is related to the predicate (< 0), (≥ 0), or $\top$ (unknown). When discussing abstract interpretation, we saw how applying the decrement operation to a nonnegative variable value results in moving its abstract element up to $\top$. This transition system could reveal a counterexample to the property being checked (all variable values stay nonnegative) that turns out to be spurious—the variable value in the concrete semantics is determined to be zero after decrementing. CEGAR will refine the abstract domain by introducing a new predicate based on this counterexample, and rerunning the analysis. The predicate added is ($= 0$), so now the domain is split into the predicates (< 0), ($= 0$), (> 0), and $\top$. After this refinement, the spurious trace is removed from the transition system, and the property of nonnegativity can be safely proven.

There are three main steps used within CEGAR: (1) step through the transition system to find a trace violating a property, (2) check if this trace is feasible, (3) if not, then generate a new predicate. Any one of the above steps can be optimized, but the success of a CEGAR-based approach strongly depends on the method for generating new predicates. The ideal is to find a predicate that can remove many spurious counterexamples from the transition system at once.

One common technique for predicate generation involves Craig interpolation [159, 74]. If some trace has been found to be infeasible, represented by the unsatisfiable conjunction of $A \wedge B$,

we would like some kind of explanation of *why* the trace is infeasible according to the concrete semantics. This is the role of a Craig interpolant, which—given $A \wedge B$ is unsatisfiable—is some formula I such that (1) $A \Rightarrow I$, (2) $I \wedge B$ is unsatisfiable, and (3) the variables found in I are also found in A and B. Finding a sufficiently general interpolant can be a difficult task, one that is out of the scope of this discussion; see [159, 7, 9] for more details. Later techniques such as [8] introduce interpolation-based methods that enumerate multiple program executions at one time. This work allows you to parameterize the degree to which this interpolation-generation drives the analysis, i.e., guessing at a safe inductive invariant for the program, versus only using interpolants to refine the abstract domain. Further improvements to the traditional CEGAR approach are given in [114].

The guarantees provided by CEGAR are: any proof of unreachability is sound, and if it reports reachability in the form of a counterexample, this is also a sound proof, i.e., it represents a feasible trace.

Variants of CEGAR

There are some techniques that, like CEGAR, are overapproximate but are still able to determine if a counterexample in the model is feasible or not. Unlike CEGAR, the abstraction refinement in these techniques is either done implicitly or not performed at all. An example of a technique that performs implicit refinement is given in [119], where a call to a SAT solver is used to check for feasibility (as with CEGAR), but in the case of an infeasible counterexample, the trace is directly removed from the model via a simple graph operation. In [172], no automated refinement is performed, but if some counterexample is returned it is guaranteed to be feasible. This is done by checking properties of the trace, e.g., is some path fully deterministic, and by running a concrete simulation.

Property Directed Reachability

A technique introduced in 2011—*Property Directed Reachability*, or IC3 [41, 42]—has been shown to be an effective procedure for proving properties of both finite and infinite transition systems. IC3 attempts to find an inductive invariant to prove some property. It does so by computing a sequence of sets, each of which is an overapproximation of the set of reachable states in at most i steps through the transition system, or "frames." An SMT solver checks whether the

given frame can violate the property-to-be-checked. If this is the case, additional strengthening clauses are added to the description of the overapproximated set. This process continues until either a feasible counterexample is found or two successive frames are found to be equal, in which case the property is proved.

IC3 differs from the original CEGAR approach in that it does not try to prove one large assertion over the state space; instead, many simple inductive assertions are added and checked incrementally. Similar to the interpolation used within many instantiations of CEGAR, IC3 is property-directed in the sense that each new assertion is added in order to remove the possibility of a property-violating trace that ends up being spurious.

2.3.2 Verification followed by Validation

In contradistinction to the *may-must* combinations in this subsection that blend *may* and *must* information throughout the analysis, the combination now considered operates in a staged manner: a *may* analysis—verification—is run first, and is followed by a *must* analysis—validation.

The impetus for this kind of staged analysis was the observation that most implementations of *may* analyses make unsound compromises in order to be effective over real-world codebases [86, 57]. The unsound compromises include ignoring arithmetic overflow by considering only the ideal domain of integers, disregarding exceptional control flow, and many others. When these unsound assumptions are left unstated, it becomes unclear whether the safety guarantees returned by a *may* analysis hold or not. However, if the unsound assumptions are made explicit, a *must* analysis can check the (hopefully small) set of remaining execution paths that have not yet been verified.

Later variations of this staged approach [76, 58, 77, 56, 6, 26] include stating unsoundness resulting from partial verification due to resource limitations, e.g., stopping due to a time bound, along with predefined sources of unsoundness, e.g., unrolling a loop a fixed number of times. The guarantees offered by this sequential *may-must* composition typically come in the form of a more robust test suite according to some coverage metric. That is, given a program P, a property ψ, and a coverage metric C, verification followed by validation computes the function $P \rightarrow \psi \rightarrow C$, where C should yield higher coverage than is possible with a *may* or *must* analysis on its own. The best case of C covering the whole program space occurs when either a *may* analysis can

soundly verify a program in the first stage, or when the subsequent *must* analysis can cover the uncovered residual space in the second stage.

The work considered in this subsection tends to use symbolic execution to perform the second phase of analysis, but other underapproximate techniques such as bounded model checking may also be used, as in [149].

2.3.3 Synergistic Combinations

Whereas CEGAR is largely a *may* analysis that uses a *must* component if refinement is needed, and the techniques in Sec. 2.3.2 cleanly separate the *may* and *must* components into distinct phases, the approaches discussed in this subsection rely on the close interaction of *may* and *must* components throughout the analysis. We name these last kinds of approaches synergistic combinations.

One distinguishing feature of synergistic combinations is the maintenance of separate overapproximate and underapproximate program states. This idea goes back to Wong-Toi and Dill's work in the 90's [84, 209], which is formulated in the context of timed automata (finite automata extended with a set of real-valued clocks) [10]. To compute whether a property is satisfied, they extend the classic dataflow analysis iterative fixpoint algorithm [134] to track both under- and overapproximate sets of reachable states. The coarseness of the overapproximating set is reduced by determining what they term *separating classes* of states that restrict the application of the *join* operator. The underapproximate set is obtained by replacing the *join* operator with an underapproximating operator, i.e., a union is replaced with a subset of the union. The basic idea is simple: if the overapproximated set of states does not contain a violation, the system is safe; whereas if the underapproximated set contains a violation, the system is not safe. Other synergistic combinations are variations on this theme that use different methods of generating these two sets.

A fruitful variant of the synergistic combinations led to the SYNERGY [112], DASH [20], and SMASH [104] algorithms. These all come from the same group of researchers and largely build on one another, so we will only summarize DASH here. The underapproximate data structure in DASH is a forest of trees, where each path in the forest corresponds to a concrete program path

that is discovered by symbolic execution. The overapproximate data structure is a finite relational abstraction of the state space that begins as the control flow graph, and is refined over time. Each iteration of DASH either expands the forest to include more known (*must*) reachable states, or refines the partition. This is done by finding a path to an error state in the overapproximation, and then discovering the *frontier* between regions known to be reachable, and those not known to be reachable; this frontier is the boundary between *must* and *may* information. Directed symbolic execution is then used to try to "expand" the frontier, and if no such expansion is possible, this information is used to refine the overapproximation—marking that there are no more states to explore beyond that frontier. The abstract error traces help direct symbolic execution, and the non-existence of concrete reachable states helps refine the overapproximation. The algorithm terminates either when an error trace exists in the forest of concrete paths, or when no path to an error exists in the overapproximation.

An extension of the DASH algorithm is given by the SMASH algorithm [104], which also uses two different structures to store *may* and *must* information, but (unlike DASH) is compositional. Separate *may* and *must* summaries are computed for each function on demand, and as the state space is explored, these summaries are queried to see if a given pre-state satisfies some *must* condition or some *not-may* condition. The result of these queries dynamically refines the two summaries of the function being analyzed. Distinct *may* and *must* function summaries are also used in [136].

2.4 Cooperative and Meta-Analyses

The analyses described in Sec. 2.2 and Sec. 2.3 are assumed to be implemented in a standalone tool, where each has a custom internal program representation, each tracks facts about the program state in different ways, etc. In contrast, the approaches described in this section assume the existence of multiple standalone analyses that can be combined in some way, either via composition, into a portfolio framework, or within a meta-analysis framework that defines interfaces for verification artifacts.

2.4.1 Conditional model checking

When an analysis fails to return an answer (e.g., due to a timeout), the computation performed up to that point is not typically communicated to the user, and a new verification attempt must be started from scratch. Researchers have long known this, but not until recently have there been calls to explicitly account for this partial verification evidence [86]. A first step in this direction appeared with the idea of *conditional model checking* [24]. In conditional model checking, given a program and a property, a model checker can provide one of three answers: a proof that the property hold, a proof it does not hold, or—in the case of failure—a condition describing the state-space explored and verified thus far. Any state space that satisfies this condition does not have to be re-verified. This condition can then be passed to a subsequent model checker, in the hope that the remaining state space outside of this condition can be effectively reasoned about by a different analysis.

The formulation of conditional model checking was abstract enough to be applied to any model checker, but in practice, the externalized condition was written in a domain-specific language closely tied to a particular model checking toolchain [28]. This prevented widespread adoption of conditional model checking outside of this toolchain. In an attempt to avoid this tight coupling, there have been various proposals to communicate the condition via an external interface that all analyzers can take as input: the source code of a program.

Embedding the condition into the program itself has been done in two ways—either explicitly through program annotations, or implicitly by excising program paths in the source code that have been verified. Using program annotations still comes with the difficulty that another analyzer must be able to understand the annotation language, making some of these approaches again tool-specific [57, 58]. Our work uses the source language as its annotation language (see 3.4), which allows for generic interfacing between analyzers at the cost of reduced expressivity that a full-blown annotation language like ACSL [19] gives.

Rather than explicitly embedding annotations, it is also possible to remove parts of the program that have been verified, leaving a *residual program* to be checked [93, 27]. The idea is that the residual program will have fewer execution paths, improving the effectiveness of downstream analyses. While similar in spirit to program slicing [208, 34] and program transformations to better

suit static analyses [141, 165, 203], which remove program statements or perform semantically-preserving code transformations, the techniques to produce residual programs remove full program paths. One drawback to residual programs is that, while the number of execution paths have been reduced, the size of the program in terms of lines of source code counterintuitively *increases*. This occurs when the condition encodes a large amount of branching, e.g., explicitly unrolling a loop a certain number of times, which is more compactly expressed in the original program (though the compactness describes more paths). The enlarged residual program can cause downstream techniques to perform worse compared to the original program.

2.4.2 Portfolio frameworks

A portfolio framework employs a large diversity of analysis tools trying independently to solve a verification problem, meaning the separate tools do not communicate with one another. If computational resources are available, one can run a portfolio in parallel, as the tools run independently, as in [16].

To avoid wasting computational resources, you can try to guess which tool would work best. This is the problem of algorithm selection [179]—finding the most effective algorithm (according to some criteria, e.g., lowest runtime, simplest output, etc.) given an input instance. This approach has been shown to be helpful in portfolio-based SAT-solvers [211], where certain structures of clauses in a boolean formula tend to be more easily solved by one SAT-solver over another.

Portfolio frameworks that employ algorithm selection are now being applied to choosing the best analyzer for a given program and property [82, 75, 180, 182]. A "selection model" is first abstracted from the program, which focuses on certain of its features, e.g., number of branch points, number of variables. Based on the model, a strategy selector assigns a ranking to the analyzers. The selection model and the strategy selector are usually the result of machine-learning (though not always, e.g., [22]). As a machine-learned function, the selections made are dependent on the underlying distribution of the training set. All such approaches have used the SV-COMP benchmarks [195] in their evaluations, and perform well. It has not been demonstrated how much these results generalize outside of this benchmark.

Unlike the other the other combination techniques described in this section, a portfolio

framework will pick one tool (or employ all in parallel) to search the *entire* state space of a given program. In contrast to the portfolio approach, the approaches in the preceding and following subsections decompose the program state space in some meaningful way.

2.4.3 Cooperative frameworks

The groundwork has been laid for analyzers to cooperate with each other as black boxes within a meta-analysis framework. The foundations involve: analysis engines that offer sound guarantees on either overapproximations or underapproximations of a program's state space; methods to combine these two approximations; and ways to externalize partial verification results. One such meta-analysis framework is given in [115], where separate analyzers discover and externalize different program invariants, and feed these invariants to a "master" (*sic*) verifier, which attempts to solve the overall verification problem. This approach differs from our current work in a number of ways, notably that in [115] much of the analysis burden still rests on a single analyzer, the framework does not compute comprehensive summaries of *must* and *may* information, and there is no attempt at generalizing partial evidence when separate analyzers do not discover new invariants. Some possibilities for integrating multiple tools into a framework for cooperative verification are given in the recent survey paper [30].

2.5 Novelty of Current Work

The ACA framework advances prior work by computing program intervals that yield sound over- and underapproximate guarantees for a given program and reachability property (described in the following chapter); by applying conditional verification in a blackbox manner; and by employing alternation between the respective sound and complete guarantees of generic $\overline{A}$ and $\underline{A}$ analyzers in a parameterizable framework.

Program intervals can characterize ψ-state reachability in a more comprehensive way than previous techniques. While prior work can compute a limited kind of program interval, ACA allows one to compute other program intervals that encode richer descriptions of input constraints (see Chap 4). Conditional analysis has previously been used either with model checkers coming from the same toolchain (CPA), or this toolchain with one additional analyzer. We apply

conditional verification in a blackbox manner, such that the conditions generated by a generic analysis tool can be passed to another; the back-and-forth can continue in a synergistic way. Previous work combined *may* and *must* information in a bespoke way, usually with two kinds of analysis engines coupled closely together, while we are able to alternate between these types of information using a framework into which you can plug multiple analyzers in a blackbox fashion.

Chapter 3

Alternating Conditional Analysis

This chapter details the core contribution of this thesis—the framework of an alternating conditional analysis. The chapter is outlined as follows: a brief motivation; formal definitions of key concepts; a discussion of the two different kinds of conditioning used; a presentation and explication of the pseudocode for an alternating conditional analysis framework; and an expanded discussion of the framework components.

3.1 Motivation

Software developers spend much of their time determining whether a program behaves as intended. This time is spent in writing specifications, perhaps documentation, unit tests, system tests, and—upon the exhibition of some program failure—debugging and reviewing the previously listed items for insights into the cause(s) of failure. Fixing the mismatch in intended and actual program behavior involves understanding the failure, repairing the fault that caused the failure, and finally making sure the repair is correct and deploying the patch. These steps can be done by hand, but previous research has shown that much of this software development can be automated, e.g., [175, 147, 143, 155, 160, 212, 150, 213].

Each of these automated techniques could be improved by broadening the failure summary from just one input vector to a more comprehensive characterization describing a set of inputs. With this characterization, duplicate failure reports could be identified, the root cause of failure

could be better understood upon seeing multiple input instances, the repairs could be more robust by covering a larger portion of the input space, and validation can be concentrated on exactly the space of inputs given by the comprehensive characterization.

In this chapter, we describe a framework that begins with individual evidence of statement reachability in some program and produces a generalized characterization of program behavior that may reach this statement. This characterization is to be both *comprehensive*, in that it safely characterizes *all* reachable behavior, and *constructive*, in that it can exactly characterize reachable behavior for some inputs. To do so, we define a *logical interval* as a pair of logical formulae that bound the input space reaching some statement in a program. (While the framework defines the general concept of reachable statements, in the remainder of this section we will continue to assume this refers to the instance of a failure statement.)

Suppose some failure report carries as evidence the triggering input vector of `<23, 9, "foo">`. We would like to compute a pair of formulae defining an upper bound, e.g., $I_1 > 0 \wedge I_2 < I_1$, and a lower bound, e.g., $I_1 > 0 \wedge I_2 < I_1 \wedge I_2 < 10$, on the failing input space; I_i represents the ith input. The upper bound defines the space of inputs on which the program *may* fail, whereas the lower bound defines the space of inputs on which it *must* fail. We conjecture that this richer failure information can be leveraged in downstream automated development processes. For example, the upper bound above indicates that the third input is not implicated in the failure, which can simplify the state space in a model checker significantly. Whereas the lower bound establishes a minimal set of inputs for regression testing, the upper bound establishes a maximal set since inputs outside of that bound are guaranteed to be failure-free. In this work, these characterizations are passed on to other automated tools—the human is kept out of the loop.[1]

To compute this characterization, we combine over and underapproximating static program analyses. Overapproximating analysis tools, such as IKOS [43], SeaHorn [113], Astrée [72], and Facebook Infer [49], can prove the absence of particular kinds of program failures. This advantage comes with the drawback that failures reports from these tools may be spurious—they may not correspond to valid program executions. Underapproximating analysis tools, such as CBMC [61],

[1] While we can imagine outputting analysis results in a form that would directly aid the programmer in answering reachability queries, this is left to future work.

KLEE [47], and Mayhem [50], have the advantage that they never report a spurious failure. The drawback with this class of tools is they may miss failures, and so cannot prove their absence.

As discussed in Chap. 2, researchers have long understood that there are advantages in combining these approaches. Conceptually, these techniques alternate the application of over and underapproximating analyses with the output of each driving the other toward convergence. Unlike these approaches, which develop bespoke alternating analyses, we develop a framework to extract and combine information from existing static analysis tools to achieve the *efficient*, *accurate*, and *safe* computation of comprehensive reachability characterizations.

The resulting framework leverages the strengths of existing state-of-the-art analysis techniques and tools, while mitigating their weaknesses.

3.2 Overview

In order to compute all the ways a program can reach some state—with some degree of accuracy—we introduce the framework of an *alternating conditional analysis*, or ACA.[2] This analysis takes as input a program and a state reachability property, and outputs a logical characterization of all ways that this property can be satisfied. A state reachability property is a property asserting that a particular state *can* be reached, while a safety property asserts that a "bad" state is *never* reached; so reachability properties can be seen as the negation of a safety property [154]. Because the focus of this work is on characterizing reachable states, safety is discussed in this negated sense.

ACA computes the characterization of state reachability by alternating between the soundness of overapproximation and the completeness of underapproximation to characterize portions of the input space as either satisfying a reachability property or not; conditioning analysis tools to ignore already-examined inputs; and repeating this process until all inputs have been accounted for. To simplify the presentation, we assume in this subsection that the alternation is strict, i.e., an overapproximation is always followed by an underapproximation and vice versa, but the algorithm in Sec. 3.5 does not require this strictness.

Before defining the formalisms used in ACA, let's look at a small example. The C program in

[2]This is an improved formulation of work first published in [101].

Fig. 3.1 includes an *input*( ) function, which returns an arbitrary integer; a **while** loop whose body contains some computation that does not modify the value of x; and a call to the function *property_satisfied*( ), whose reachability we want to characterize.

```
int main( ) {
  int x = input( );
  while (input( ) ≠ 0) {
    /* computation not involving x */
  }
  if (x > 0) { property_satisfied( ); }
}
```

Figure 3.1: Uninstrumented program

A run of ACA computes a characterization of all inputs that lead to *property_satisfied*( ) being called. The analysis begins by running an overapproximating analyzer on the original program, searching for the reachability of *property_satisfied*( ). Suppose this analyzer claims that it has found a reachable path, and passes the evidence of this claim to be validated (or invalidated) by an underapproximating analyzer. The underapproximator validates the evidence, and collects a logical characterization of its reachability in terms of x, namely $x > 0$.

We now know the program's behavior when $x > 0$, so ACA conditions future rounds of analysis to ignore program executions satisfying this input constraint, and checks if there are any *other* inputs that can reach *property_satisfied*( ). This conditioning is embedded in an *assume* statement. The semantics of *assume*, given some predicate p are: *if(p): skip; else: exit;*—this prunes out execution paths that do not satisfy p. We can place the *assume* statement at any point between the initial read of x and the **if** statement; we choose to place the *assume* statements early in the program encoding so that they have have the possibility to block more execution paths from having to be analyzed. Under these new assumptions, ACA again runs an overapproximating analyzer on the instrumented program of Fig. 3.2.

This time an overapproximate analyzer declares *property_satisfied*( ) to be unreachable. Because this result is overapproximate, we are safely guaranteed that all program paths have been

```
int main( ) {
  int x = input( );
  assume(¬(x > 0));  /* ignore executions where x > 0  */
  while (input( ) ≠ 0) {
    /* computation not involving x */
  }
  if (x > 0) { property_satisfied( ); }
}
```

Figure 3.2: Instrumented program

accounted for, and the reachability of *property_satisfied*( ) can be exactly characterized by the logical formula $x > 0$.

ACA relies on the alternation of over- and underapproximation to compute the reachability characterization. An overapproximate analysis is able to safely compute a fixpoint when reasoning about the body of the **while** loop, whereas an underapproximate analysis may get "stuck" exploring this loop, and run out of resources before ever reaching *property_satisfied*( ). In addition, overapproximation can safely declare portions of the input space as unable to satisfy a reachability property, which underapproximation cannot do in general. On the other hand, underapproximation can accurately characterize an execution path as feasible or not, mitigating the nuisance of false positives inherent to overapproximation. While at least one alternation is needed in order to establish an upper and a lower bound, we note again that a strict alternation is not necessary in ACA.

3.3 Definitions

To combine *may* and *must* information—computed by a variety of program analysis tools—we define a logical characterization of all program inputs, with respect to some state reachability property. We call this logical characterization a *program interval*; it consists of an *upper bound* that is guaranteed to subsume all reachable paths (the *may* information), and a *lower bound* that is guaranteed to be subsumed by all reachable paths (the *must* information). We now define the formalisms used in computing this characterization.

Let $\mathcal{P}$ be the domain of programs and Ψ the domain of reachability properties. A program

analysis, A, targets a pair of elements $\langle p, \psi \rangle$, $p \in \mathcal{P}$ and $\psi \in \Psi$, in order to provide information about whether the executions of p conform to $\neg\psi$, i.e., whether $p \models \neg\psi$. We refer to states that satisfy ψ as ψ-states.

Most modern program analysis frameworks produce some form of evidence about their claims of ψ-state reachability or unreachability. Evidence of reachability can be as complete as initial concrete inputs or a full trace of program execution leading to a potential ψ-state, or as partial as designating a single statement at which the ψ-state may occur. A general model for such evidence, termed an *error automaton*, has been developed in [31]. These automata can be used as evidence of ψ-state reachability for both over- and underapproximating analyses. Within error automata, a program is represented as a *control flow automaton*, which is a control flow graph where the nodes and edges are flipped, i.e., the program locations are the nodes and the program statements are the edges.

Error automata have a start state that coincides with the initial program state and transitions that are labeled by control flow branches. The language of an error automaton is a set of program executions, e.g., the set of all executions for an automaton that accepts immediately, or a single execution which has a single enabled transition at each state.

Definition 2 (Error automaton)**.** *An error automaton $A = (Q, \Sigma, q_0, q_\psi, \rightarrow)$ contains a set of program states Q, the alphabet of program statements Σ, the initial state q_0, the target state of q_ψ, and the relation $\rightarrow$ determining which states can transition to another after an application of some program statement.*

Let $\mathcal{E}$ be the domain of evidence produced by a program analysis. This would include evidence of ψ-state reachability, R_ψ, or unreachability, U_ψ. An evidence-producing program analysis is $A : \mathcal{P} \times \Psi \rightarrow \mathcal{E}$. The SV-COMP competition contains over 30 analyzers that are evidence-producing [198].

An overapproximating analysis $\overline{A} \in A$ is one where $\overline{A}(p, \psi) = U_\psi \implies p \models \neg\psi$. An underapproximating analysis $\underline{A} \in A$ is one where $\underline{A}(p, \psi) = R_\psi \implies p \not\models \neg\psi$. Conceptually, $\overline{A}$ is capable of proving the unreachability of a ψ-state and $\underline{A}$ is capable of proving ψ-state reachability.

We assume that a program, $p \in \mathcal{P}$, has a set of input statements, $i \in I$, that return well-typed values, where $dom(i)$ is the domain of i's type. To simplify the presentation of ACA, we

formalize the analysis for programs that read from each input statement a single time, thus the input domain of p is $D = \prod_{i \in I} dom(i)$, which means D is finite. The framework, and the implementation described in Chapter 5, work more generally by modeling the kth execution of an input statement by the pair (i, k), and the input domain as the product of execution-specific input domains, $\prod_{1 \leq j \leq n} dom(i_j)$, where the inputs on an execution are $\langle (i_1, 1), \ldots, (i_n, n) \rangle$. The implementation can enforce a bound on the length of sequence to restrict analysis to a finite D; in doing this, it may lose accuracy by characterizing all inputs beyond the given length as potentially reaching a ψ-state.

State Characterization

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 *logical intervals*.

We characterize program behavior using logical formulae that encode regions of D on which ψ-states are detected. This characterization consists of *a collection of lower bounds* ($\underline{\mathcal{I}}$) guaranteed to be subsumed by a program's true ψ-state reachability, R_ψ, and *a collection of upper bounds* ($\overline{\mathcal{I}}$) guaranteed to subsume R_ψ. A sufficient condition on p's inputs reaching a ψ-state is given by $\underline{\mathcal{I}}$, while $\overline{\mathcal{I}}$ comprises a necessary condition for the same.

Definition 3 (Logical Interval)**.** *A logical interval, $I = [\underline{I}, \overline{I}]$, is an ordered pair of logical predicates on the inputs such that $\underline{I} \Rightarrow \overline{I}$. We call the first element, $\underline{I}$, the interval's lower bound; and the second, $\overline{I}$, the upper bound. The $\underline{I}$ and $\overline{I}$ terminology will be overloaded to also stand for the left and right projections, respectively, of I. A logical interval describes constraints on a subset of program inputs, where $\underline{I}$ describes a portion of the input space that must reach a ψ-state, and $\overline{I}$ describes a portion of the input space that may reach a ψ-state.*

Assuming a domain of integer variables, an example of a logical interval is $[(x < 3), (x < 5)]$, as $(x < 3) \Rightarrow (x < 5)$. Equivalently, the set of inputs described by the lower bound—integers less than three—is subsumed by the set described by the upper bound.

Definition 4 (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.

The logical intervals $[(x = 3), (0 < x < 7)]$ and $[(x = 14), (11 < x < 15)]$ are disjoint, as $(0 < x < 7) \wedge (11 < x < 15) \equiv \emptyset$. But the intervals $[(x < 3), (x < 5)]$ and $[(x < 4), (x < 7)]$ are *not* disjoint, because their upper bounds have a nonempty intersection.

Definition 5 (Program Interval)**.** *A program interval, $\mathcal{I}$, is a set of logical intervals $\{I_1, \ldots, I_n\}$ that semantically bound a program's ψ-state reachability, R_ψ, such that R_ψ lies between the disjunction of lower bounds and the disjunction of upper bounds. That is, let the disjunction of lower bounds, $\bigvee_{i=1}^{n} \underline{I}_i$, be named $\underline{\mathcal{I}}$, and the disjunction of upper bounds, $\bigvee_{i=1}^{n} \overline{I}_i$, be named $\overline{\mathcal{I}}$, then $\underline{\mathcal{I}} \Rightarrow R_\psi \Rightarrow \overline{\mathcal{I}}$.*

Suppose there is a program whose ψ-state reachability, R_ψ, is exactly defined by the formulae $(2 < x < 6) \vee (11 < x < 15)$, i.e., the ψ-state will be reached when x takes on a value from the set $\{3, 4, 5, 12, 13, 14\}$. One possible program interval that bounds R_ψ can be given as a set of disjoint intervals: $\mathcal{I} = \{[(x = 3), (0 < x < 7)], [(x = 14), (11 < x < 15)]\}$. The description of lower bounds is given by $\underline{\mathcal{I}} = ((x = 3) \vee (x = 14))$, and the upper bounds by $\overline{\mathcal{I}} = ((0 < x < 7) \vee (11 < x < 15))$. Notice that $\overline{\mathcal{I}}$ is a strict overapproximation of R_ψ, as it contains all values described by R_ψ, and includes some that are not—namely the set $\{1, 2, 6\}$. A strict underapproximation is given by $\underline{\mathcal{I}}$, which includes the values 3 and 14 found in R_ψ, but does not include the other values $\{4, 5, 12, 13\}$.

The range of input space a program interval can characterize lies between two extremes: $\underline{\mathcal{I}} = \overline{\mathcal{I}} = R_\psi$ (most informative, exact characterization) and $\{[\mathit{false}, \mathit{true}]\}$ (non-informative). A partition on $\mathcal{I}$ induces a set of *disjoint intervals*.

Ideally a program interval encodes a small reachable subspace of D. In the worst case, $\overline{\mathcal{I}} = D$, and all of the input space is implicated. Even here ACA is able to isolate the inaccuracy to one interval that will have as its upper bound the complement of the disjunction of all of the other upper bounds. For example, continuing the running example above, let the first disjoint interval be named I_1, and the second I_2. Now if $\overline{\mathcal{I}} = D$, then the inaccuracy can be isolated to the last element of the program interval $\mathcal{I} = \{I_1, I_2, [\mathit{false}, (\neg(\overline{I}_1 \vee \overline{I}_2))]\}$. The remaining intervals provide useful information about ψ-state reachability, especially in their lower bounds.

⋮
directive@L15: *true* branch
directive@L23: *true* branch
⋮

```
...
L14 assume(c_j);
L15 if (c_j) {
...
L22 assume(c_k);
L23 if (c_k) {
...
...
```

Figure 3.3: Line directives derived from E (above); Program instrumented with full direction using these directives (below).

```
main() {
  x = read();
  assume(!(x<5));
  ...
  if (x<2) \{
    /* ignore */
  }
  ...
  if (x==4) \{
    /* ignore */
  }
  ...
}
```

Figure 3.4: Program conditioned to ignore previously-analyzed subspaces

3.4 Conditioning Program Analyses

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

There are many possible ways to define conditioning, and in this work we employ two different approaches. The first approach focuses the analyzer's search space when characterizing reachability by telling it precisely where to go, i.e., search *only* along this path. To do this, given an error automaton E, the automaton structure is used to direct the state-space search performed by $\underline{A}$. Any branches that are inconsistent with E go unexplored.

For example, in Fig. 3.3, the error automaton (shown above the horizontal rule) gives directives for which branches to take at each line of a conditional, e.g., at both lines 15 and 23, $\underline{A}$ should take the *true* branch. The program (shown below the horizontal rule) is then instrumented with an `assume` statement before each conditional statement according to the given directive.

The second approach to conditioning asks the analyzer to ignore a given portion of the input space by telling it areas that do not need to be explored, i.e., do *not* look at this set of paths. To do this, given E_u as a logical formula defined over D, we instrument the program with the statement `assume`$(\neg E_u)$ to inform A that it need not consider that behavior.

For example, in Fig. 3.4, the `assume` statement at the top of the program conditions analyzers to avoid exploring the body of both `if` statements. Note that the conditioning in Fig. 3.3 differs from that in Fig. 3.4 in that in the latter, the direction is not explicitly given for each branch. The clause `(x<5)` is more general than either of the `if` conditionals in Fig. 3.4, and consequently can prune away more of the execution space to-be-explored at one time.

While the second form of conditioning aims to restrict the analysis to avoid previously analyzed subspaces, it does not guarantee that this will be effective. For example, if some expression e in `assume`(e) cannot be accurately modeled by an abstract domain in some $\overline{A}$, then the semantics of the `assume` will be overapproximated. Generalization is used to address this issue.

We consider this second form of conditioning to be "effective" if some analyzer is able to produce a new piece of evidence on the conditioned program. We will place this into a definition for cross referencing in Chapter 5.

Definition 6 (Effective conditioning)**.** *When a program is conditioned to avoid the input space described by some program interval $\mathcal{I}$, and some analyzer produces evidence describing a portion of the state space not already encoded in $\mathcal{I}$, the conditioning is called effective.*

Dually, the first form of conditioning $\underline{A}$ aims to restrict the analysis to a ψ-state reachable space of the program behavior in order to confirm its reachability and characterize it. Effective conditioning for $\underline{A}$ means that the ψ-states that are characterized are guaranteed to be executable. When such guarantees are not computed, e.g., due to timeouts or overapproximation in underlying constraint solvers, either the bottom *false* value can be safely returned, or specialization can be applied to further restrict the conditioning.

3.5 ACA Algorithm

The structure of the ACA algorithm is depicted in Figure 3.5, and its recursive definition in pseudocode is given in Algorithm 1. The ACA algorithm depends on certain properties of the algorithms that it combines. Specifically, it assumes that over and underapproximating program

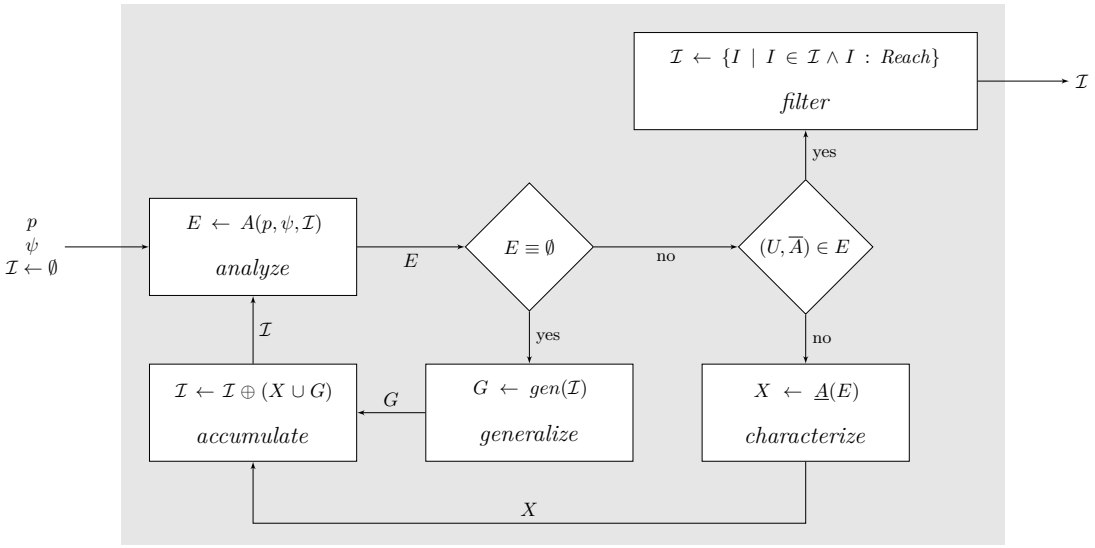


Figure 3.5: Alternating Conditional Analysis Framework

analyses are typed:

$$A : \mathcal{P} \times \Psi \rightarrow \{U_\psi, R_\psi, \perp\}$$

where $\perp$ encodes the inability of the analysis to compute a result, e.g., due to a timeout, encountering an unsupported language feature, or unsoundness relative to the nature of the analysis' approximation. For example, if some $\underline{A}$ returns U_ψ, this result is converted to $\perp$.

ACA extends logical intervals with two types: descriptions of reachability and descriptions of unreachability, i.e., $I : \{Reach, Unreach\}$. In addition, it assumes the strict monotonicity of generalization relative to a finite order on formulae. The formulae are related in a lattice in which the ordering is defined by logical implication.

The algorithm begins by calling ACA with an initial state of a program p, a state reachability property ψ, and an empty program interval $\mathcal{I}$. The call to *analyze* first instruments p with the negation of already-characterized state space in $\mathcal{I}$ (the first form of conditioning discussed above), after which, any number of evidence-producing analysis tools search for ψ-state reachability in this instrumented program. Because each analysis tool's search for reachability evidence is distinct from another's, a portfolio of tools with differing strengths can be launched in parallel. Their evidence is collected in the set E.

There are three possible cases to consider depending on what is contained in E. In the first case, when E has some unreachability evidence given by an overapproximate analyzer, then the current program interval has safely characterized all reachable ψ-states. The *filter* function removes any "false positives" potentially collected during previous calls to *characterize*, retaining the intervals typed as reachable characterizations, and the final program interval is returned.

In the second case, some analyzer has found evidence of ψ-state reachability. (We keep a cache of evidence to ensure that some $\overline{A}$ does not keep declaring previously-seen evidence as new.) The function *characterize* takes each piece of evidence in E and runs a directed symbolic execution according to the assumptions given in the corresponding error automaton. The output of *characterize* is a logical interval extended with a type annotation that denotes whether the path condition validates ψ-state reachability or not. If the evidence is invalidated, this represents a false positive. Intervals typed as false positives will be filtered out before termination, but can be helpful in conditioning future rounds of analysis *away* from these paths. The path condition is safely sliced and is added to $\mathcal{I}$. Slicing eliminates branches from the path condition that are independent of reaching the ψ-state, and is a way of safely increasing the accuracy of the lower bounds—by dropping irrelevant clauses in the path condition, more inputs can be characterized as definitely reaching the ψ-state. If symbolic execution does not complete within a time bound, the evidence can be successively specialized (down to the limiting *false* value) until a characterization is returned. The typed logical intervals are then placed in X.

In the final case, there is no new evidence, and the program interval must be generalized, or "widened." The generalization process will converge to *true*, due to monotonicity. The *accumulate* function ensures that the space described by logical intervals that are added in *characterize* or widened in *generalize* are added to $\mathcal{I}$ such that the input space described by $\overline{\mathcal{I}}$ grows monotonically.

Theorem 1 (Termination)**.** *Algorithm 1 terminates if each A terminates and the generalize function is strictly monotone relative to a finite ordering on formulae.*

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

An execution of function ACA is guaranteed to terminate if every call to A terminates and if

Algorithm 1 Alternating Conditional Analysis

```
1: function ACA(P, ψ, 𝓘)
2:     E ← analyze(P, ψ, 𝓘)
3:     switch
4:         case (U_ψ, ¬A) ∈ E
5:             return filter(𝓘)
6:         case (R_ψ, A) ∈ E
7:             X ← characterize(P, ψ, 𝓘, E)
8:             𝓘′ ← accumulate(𝓘, X)
9:             ACA(P, ψ, 𝓘′)
10:        case E ≡ ∅
11:            G ← generalize(𝓘)
12:            𝓘′ ← accumulate(𝓘, G)
13:            ACA(P, ψ, 𝓘′)
```

Specifications for ACA Functions

analyze$(P, \psi, \mathcal{I})$
Input: Program P, reachability property ψ, program interval $\mathcal{I}$
Output: Set of analysis evidence E

filter$(\mathcal{I})$
Input: Program interval $\mathcal{I}$
Output: Program interval characterizing positive reachability $\mathcal{I}'$

characterize$(P, \psi, \mathcal{I}, E)$
Input: Program P, reachability property ψ, program interval $\mathcal{I}$, set of analysis evidence E
Output: Program interval X

accumulate$(\mathcal{I}, X)$
Input: Program interval $\mathcal{I}$, program interval X
Output: Program interval $\mathcal{I}'$ that merges $\mathcal{I}$ and X such that $\mathcal{I}' \sqsupset \mathcal{I}$

generalize$(\mathcal{I})$
Input: Program interval $\mathcal{I}$
Output: Program interval G such that such that $G \sqsupset \mathcal{I}$

generalize produces a result that is greater in the ordering due to the finite bound on chains in the ordering. (Recall that the input domain D is made finite.)

The recursive loop executes at most once for each element of X or G that is computed. New elements are computed by either *characterize* (in case 2) or *generalize* (in case 3); otherwise case 3 leads to termination. Each element in turn is added to $\mathcal{I}$ in *accumulate*. There can be only finitely many elements of $\mathcal{I}$, since the upper bounds of the reachable characterizations—intervals typed as *Reach*—and the upper bounds of the unreachable characterizations—intervals typed as *Unreach*—are disjoint, and their union must be a subset of D. □

Theorem 2 (Safety)**.** *Algorithm 1 terminates with a program interval $\mathcal{I}$ such that the reachable space of p relative to ψ, R_ψ, is bounded by $\mathcal{I}$, $\underline{\mathcal{I}} \implies R_\psi \implies \overline{\mathcal{I}}$.*

Proof. For all elements of $\mathcal{I}$, the lower bounds are computed by $\underline{A}$ and then sliced. By definition $\underline{A}$ computes a safe underapproximation and thus any ψ-state reachability detected is guaranteed to be feasible. If no reachability is detected, then *false* is used as the lower bound which is guaranteed to be safe. Slicing only eliminates subformulae that are provably independent from the ψ-state reachability characterized by the interval. Thus, while the sliced lower bound may subsume the original path to the ψ-state detected by some $\overline{A}$, it is guaranteed to underapproximate the space of all paths to the ψ-state that are equivalent up to execution of independent branches.

For any iteration of the ACA function, the upper bounds from all prior rounds of analysis are conditioned into the program in *analyze*. The algorithm terminates only if: (1) the conditioning of all upper bounds permit some $\overline{A}$ to prove the program free ψ-state reachability, or (2) a final run of *generalize* results in $\perp$ in which case $\overline{\mathcal{I}} \equiv true$. The first case guarantees that the upper bounds of $\mathcal{I}$ are safe, since an $\overline{A}$ computes a safe overapproximation, and the second case is trivially safe. □

3.5.1 Specification of *analyze*

The purpose of *analyze* is to run one or more program analyzers in search of ψ-state reachability in a program conditioned by some program interval $\mathcal{I}$. The analyzers are typed to specify whether they produce only overapproximate proofs, only underapproximate proofs, or are able to compute

both. Any proofs of ψ-state reachability are returned in the form of evidence (e.g., an error automaton) that may be interpreted by some underapproximate analyzer.

Much of *analyze* is left underspecified because most of the work is done within the black boxes that produce program proofs. We also want to allow for freedom in the way conditioning is implemented, e.g., with residual programs, or with `assume` statements. The three expected invariants of *analyze* are: 1) that overapproximate analyzers are sound with respect to their typing, i.e., a proof of unreachability can be trusted; 2) in the case of a reachability proof, an analyzer can return an interpretable proof such as an error automaton; and 3) that conditioning removes the concrete traces described by $\overline{\mathcal{I}}$ from the semantics of the unconditioned program. Soundness in underapproximate proofs is desired, but as it will be reverified by a separate $\underline{A}$ in *characterize*, this is not a requirement.

Each application of program conditioning yields a new program, and at each new application of *analyze*, this new program can be given as a verification problem to a large number of analysis tools. As each of these tools operate independently and do not share results within *analyze*, this is an embarrassingly parallel problem, so you can run as many at one time as computing power allows. This function can be implemented with different strategies for when to stop—after the first piece of evidence is found, or allow other analyzers to continue trying to find some different piece of evidence.

3.5.2 Specification of *characterize*

The purpose of *characterize* is to validate (or invalidate) any given reachability proofs and translate this validation into an interval to be added to $\mathcal{I}$.

The only requirement of *characterize* are that it safely underapproximate the program behavior of program P conditioned by $\mathcal{I}$, i.e., it explores only concrete program traces not implied by $\overline{\mathcal{I}}$.

The pseudocode for *characterize* is given in Algorithm 2. The *condition* function on line 2 refers to the first form of conditioning described in Sec. 3.4, and guarantees some underapproximator $\underline{A}$ must only explore traces not implied by $\overline{\mathcal{I}}$. We guarantee termination when ACA calls *characterize* by requiring specialization within *directed_symex*. A failed computation can occur if some branch constraint is too complex for a solver inside a symbolic execution engine. We guarantee that some

concrete path is produced by requiring the call to directed symbolic execution in *directed_symex* on line 5 to concretize any problematic branch constraints, down to the limiting value of *false*. For example, if a boolean expression with a modulo operator `(x % 2 == 0)` cannot be handled, concretizing this expression would assign a value to `x`, allowing traces beyond this problematic branch point to be explored. Concretizing a branch can be done with techniques used in concolic execution [102].

Algorithm 2 *characterize*

```
1: function CHARACTERIZE(P, ψ, 𝓘, E)
2:     p' ← condition(P, 𝓘)
3:     intervals ← {}
4:     for e ∈ E do
5:         r ← directed_symex(p', e)
6:         r' ← slice(r)
7:         intervals ← {r'} ∪ intervals
8:     return intervals
```

The requirement on *slice* is that, given some path condition computed by *directed_symex*, the sliced path condition remain a safe underapproximation. For example, suppose in some program a ψ-state is reached exactly when integer variable x is less than 2. But suppose the path condition, $x < 2 \wedge y > 0$ retains information about an irrelevant branch involving another integer variable y. Now *slice* can remove the constraint $y > 0$, as this is still a sound underapproximation of all ψ-state reachability; but *slice* cannot remove both $y > 0$ and $x < 2$, because that is an overapproximation of ψ-state reachability. Static slicing techniques fulfill this soundness requirement [208], while most dynamic slicing techniques do not [5, 210].

Lines 3 and 7 specify that the sliced characterizations be collected in some set, which is returned in line 8.

3.5.3 Specification of *generalize*

The purpose of *generalize* is to increase the number of paths accounted for by $\mathcal{I}$ by relaxing some upper bound. When an upper bound is relaxed, yielding a logical formula that constrains less of the input space than before, the area between upper bounds and lower bounds becomes *may* information: the formula in the logical lattice between $\overline{I}$ and $\underline{I}$ describe inputs that may reach a ψ-state. Figure 3.6 shows two logical intervals that describe disjoint regions of the input

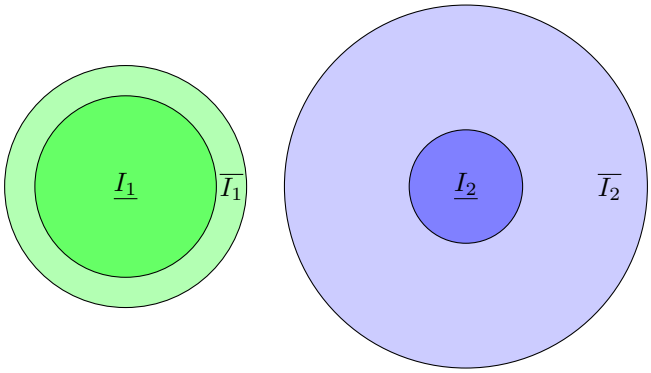


Figure 3.6: Two logical intervals I_1 and I_2 whose upper and lower bounds describe disjoint regions of the input space.

space, where the lighter shadings represent the *may* space between some $\overline{I}$ and $\underline{I}$. Note that I_1 and I_2 have different degrees of approximation: the space between $\overline{I_1}$ and $\underline{I_1}$ is smaller (i.e., less approximate) than that between $\overline{I_2}$ and $\underline{I_2}$. . Generalization is needed for ACA to reach a fixed point, so it is required to be monotone in the lattice of logical formula (the top value being *true*).

The pseudocode for *generalize* is given in Algorithm 3, which takes a program interval $\mathcal{I}$ as its only argument. The first step is to choose some interval I from the collection of intervals in $\mathcal{I}$, given as *choose_interval* on line 2. This selection can be according to some heuristic, e.g., choose the last-discovered interval, choose the interval whose upper bound contains the most conjuncts, etc. We are looking for a relaxed upper bound that will provide effective conditoning in the subsequent round of analysis.

The upper bound of the selected interval, $\overline{I}$, is relaxed, or generalized in *relax_upper* on line 3. The only requirement on *relax_upper* is that it strictly move the upper bound $\overline{I}$ higher in the logical lattice, i.e., $I \sqsubset \mathit{relax\_upper}(I)$. Line 4 updates the program interval with the interval returned by $\mathit{relax\_upper}(\overline{I})$, which is returned in line 5.

Algorithm 3 *generalize*

```
1: function GENERALIZE(𝓘)
2:     I ← choose_interval (𝓘)
3:     I′ ← relax_upper (I)
4:     𝓘′ ← 𝓘 \ I
5:     𝓘″ ← {I′} ∪ 𝓘′
6:     return 𝓘″
```

3.5.4 Specification of *accumulate*

The purpose of *accumulate* is to merge the *may* and *must* information encoded in different intervals I_i into the program interval $\mathcal{I}$. The merge operation is depicted by $\oplus$ in Fig. 3.5. The only requirement on $\oplus$ is that it yields a value in the logical lattice that is higher than either of the operands, that is, for program intervals $\mathcal{I}$ and $\mathcal{X}$, $\mathcal{I} \sqsubset (\mathcal{I} \oplus \mathcal{X})$ and $\mathcal{X} \sqsubset (\mathcal{I} \oplus \mathcal{X})$.

The implementation for *accumulate* may be as simple as a set union, or it may preserve properties amongst intervals such as disjointness, or it may even be forgetful of certain intervals. This last option could be desired to drop any intervals tagged as describing paths that do not reach a ψ-state—this is one way to reduce the complexity encoded in some program conditioned by $\mathcal{I}$.

3.5.5 Specification of *filter*

The purpose of *filter* is to remove intervals that describe program executions that do *not* reach a ψ-state. If the intervals are extended with the two-valued type of {`Reach`, `Unreach`}, this filter can be implemented by a match on types. The reason we throw `Unreach` types away is to ensure that, at that end of ACA, the invariant holds that program intervals describe ψ-state reachability. If a complete run of ACA is seen as an atomic action to compute some $\mathcal{I}$, this invariant can be temporarily broken across rounds of analysis.

3.6 Modular ACA

We have discussed the ACA framework in terms of analyzing whole programs, but it is also possible to use the ACA framework to analyze smaller fragments of programs. Why would we want to do this? Programs may include portions that are difficult for analyzers to handle, such as external library calls [123], functions that contain nonlinear arithmetic [98], or massive state spaces to explore [14]; and these can cause analyzers to either fail, or to return uninformative results (e.g., $\top$). Rather than simply giving up when whole program analysis fails, we can break the program into smaller fragments to provide sound characterizations of the portions of the program over which the analyzers can successfully operate.

3.6.1 Formulation of Modular ACA

To adapt the above formulation of ACA to a modular analysis, we define the concepts of a *module* and a *prestate*. (Prestates are closely related to preconditions [124] and guards [83], but are not equivalent.)

Definition 7 (Module)**.** *A module is any function definition or compound statement. We do not require the function to be "closed" in the sense that all called functions have full definitions. Called functions can be modeled by specifications of pre- and post-states, the loosest being a pre-state of true and a post-state of a nondeterministic value of the called function's type.*

Modeling called functions by returning any value introduces overapproximation, meaning only the information in the upper bound $\overline{\mathcal{I}}$ is meaningful in the returned $\mathcal{I}$. We explore propagating $\overline{\mathcal{I}}$ in this modular setting in Chap. 7. To retain a valid lower bound $\overline{\mathcal{I}}$, we can impose the requirement of being closed with respect to called functions.

Definition 8 (Prestate)**.** *A prestate consists of an environment of typed variables along with logical formulae constraining their values, modeling the possible states prior to executing some module.*

A module, m, is analyzed in an execution context constrained by a *prestate*, consisting of declarations and constraints on state variables that are referenced in a given module. For variables of basic types such as `int` or `char`, declaring and constraining state variables can be done by following a nondeterministic initialization with an `assume` statement constraining this variable. To create nondeterministic complex data types, e.g., a `struct`, you must recursively initialize each of its fields with a nondeterministic value and constrain it accordingly. Arrays that are nondeterministically bounded must be given some concrete bound. For creating nondeterministic pointers to variables, we assume the analysis tool will have an annotation primitive to signal this, such as the `__VERIFIER_nondet_pointer()` construct used by SV-COMP tools.

Definition 9 (Modular Alternating Conditional Analysis)**.** *A modular alternating conditional analysis is a modification of ACA given in Algorithm 1, where program p is replaced with module m, and where the first statement in m initializes and constrains the execution environment as specified by a prestate α.*

Running modular ACA on m results in a program interval whose upper bound describes the constraints on state variables in relation to a state reachability property; by Definition 1, these constraints are guaranteed to subsume all reachable paths after executing m with a prestate of α. This program interval may then be reused to drive further analysis in some module in which m appears, as we explore in Chap. 7.

3.6.2 Limitations

When we do not have information on a module's prestates, in order to maintain soundness, modular ACA must assume maximally permissive prestates. That is, each state variable is allowed to take on any value in its domain. Our definition of module also allows for functions called within the module to be modeled overapproximately. Overapproximate assumptions such as these can lead to paths that are infeasible at execution time being declared feasible. This is a fundamental limitation of assuming maximally permissive prestates, and this overapproximation can be improved by obtaining more accurate runtime models that provide tighter constraints on the prestate.

Chapter 4

Existing Analyses as ACA

This brief chapter recasts prior work as different instantiations of ACA, showing both the generality of the framework and highlighting connections between seemingly distinct analysis techniques. Each prior technique is recast into ACA terminology by detailing how the framework should be parameterized in each case, and describing the expected output in terms of a program interval. Recall that if an ACA interval has a lower bound $\underline{\mathcal{I}}$ other than *false*, that means that the ψ-state is reachable. For instance, a proof of reachability is given in a program interval as $\{[c, \mathit{true}]\}$, i.e., a ψ-state is reachable under the input constraints of c, but we say nothing about the rest of the input space—so the upper bound is *true*. A proof of unreachability is given as $\{[\mathit{false}, \mathit{false}]\}$, i.e., ψ is provably unreachable.

We divide existing analyses into the three broad categories given in Chapter 2: classical analyses—using exclusively over- or underapproximate techniques; may-must analyses—employing a blend of over- and underapproximation; and cooperative analyses—using distinct analyses out-of-the-box and possibly combining their partial analysis results.

4.1 Classical Analyses

4.1.1 Overapproximators

The analysis techniques we classify as purely overapproximate include abstract interpretation, data flow analysis, model checking, and deductive analyses. Their distinctive characteristic is that, given a reachability property, each of these techniques can potentially prove its unreachability. The inability to find a proof could be the result of hitting a resource limit; or could be due to finding a reachable state in the analysis' model, which may be infeasible in concrete executions.

We will first cover the commonalities of these techniques within ACA, and discuss where they diverge in the subsequent paragraphs. As an instance of ACA, each overapproximate technique would instantiate the analysis portfolio with a pure overapproximator, i.e., one that is typed as $A : \mathcal{P} \times \Psi \to \{U_\psi, \bot\}$. From this typing, it follows that an overapproximate ACA can produce only two kinds of intervals: either $\{[\mathit{false}, \mathit{false}]\}$, i.e., ψ is provably unreachable; or $\{[\mathit{false}, \mathit{true}]\}$, i.e., ψ *may* be reachable. How to report potential reachability is left unspecified in the classical formulations of overapproximate techniques; a given tool could be as crude as simply reporting "may be reachable," or could be more specific and output a trace in the state space model. In all cases, any potential reachability is not provable, so *characterize* must always return the interval that implicates the whole program in potential reachability: $\{[\mathit{false}, \mathit{true}]\}$.

The difference among techniques lies in their respective instantiations of *analyze* and *generalize*. In model checking's original formulation, *analyze* constructs a finite-state model, and given an initial state, a graph search is performed to enumerate each reachable state. Abstract interpretation's instantiation of *analyze* requires specifying the semantics of transfer functions, and tracking the possible values variables can take within a lattice as the transfer function is applied while symbolically executing a program.

The *generalize* instantiation is either nonexistent in model checking, implicit if using symbolic model checking, given ahead of time by creating a more-abstract state space model by hand, or given by some *acceleration* technique such as partial order reductions. The *generalize* instantiation for abstract interpretation is either given implicitly in the semantics of the transfer function, or given explicitly with a widening operation [69, 71, 66].

4.1.2 Underapproximators

The techniques we classify as underapproximate include symbolic execution and bounded model checking. These techniques can be seen as duals of the overapproximate ones. As such, their distinctive characteristic is that, given a reachability property, an underapproximate technique can potentially prove its reachability. Because these techniques operate over the concrete semantics of a program, they cannot in general compute sound and complete proofs of a property's unreachability (see Halting Problem, Rice's Theorem).

As an instance of ACA, each underapproximate technique would instantiate the analysis portfolio with a pure underapproximator, i.e., one that is typed as $A : \mathcal{P} \times \Psi \rightarrow \{R_\psi, \bot\}$. From this typing, it follows that an underapproximate ACA can produce two variations of intervals: either $\{[\mathit{false}, \mathit{true}]\}$, i.e., ψ *may* be reachable—this may occur when hitting some resource bound, for example; or $\{[c_1, c_1], \ldots, [c_i, c_i], [\mathit{false}, \mathit{true}]\}$, i.e., an accumulation of exact, disjoint characterizations of ψ-state reachability. Note the final logical interval, $[\mathit{false}, \mathit{true}]$, which implies that all inputs outside of $\bigvee_{i=1}^{i} c_i$ can possibly reach a ψ-state.

The differences between symbolic execution and bounded model checking lie in their instantiations of *analyze* and *characterize*. In symbolic execution, the *analyze* phase examines one collection of concrete execution paths (i.e., path conditions) at a time, where input variables assume symbolic values constrained by branch conditions. At each branch, a SAT solver checks for feasibility, and records this result in a conjunctive formula; in the case that a ψ state is reached, the *characterize* implementation is simply yielding the conjunctive formula that is already at hand.

In bounded model checking, the *analyze* phase consists in constructing the entire finite state system (up to a bound) in the form of a disjunctive logical formula and checking for its satisfiability via a SAT solver. If the SAT solver finds a solution to its query, this means a ψ state is reachable, and the *characterize* phase consists in converting the SAT assignment into some counterexample.

As pure underapproximate techniques, there is no specification of *generalize*; so in the case of the *analyze* phase finding no evidence, these techniques must return the noninformative program interval of $\{[\mathit{false}, \mathit{true}]\}$.

4.2 May-must Combinations

4.2.1 Counterexample-Guided Abstraction Refinement

Instantiations of CEGAR, discussed in Sec. 2.3.1, can be modeled in ACA by instantiating the portfolio with the desired model checker as a pure overapproximator, and the underapproximator used in *characterize* as a SAT solver, and by modifying the definition of *accumulate*. The *accumulate* function will be the abstraction-refinement when the characterization is typed as a false positive. If reachability to a ψ-state is confirmed, ACA exits upon finding this reachability. The possible intervals produced by CEGAR within ACA are $\{[c, \mathit{true}]\}$ or $\{[\mathit{false}, \mathit{false}]\}$.

4.2.2 Verification followed by Validation

Instantiations of verification followed by validation, discussed in Sec. 2.3.2, can be modeled as a one-iteration pass of ACA. The portfolio employed in *analyze* is instantiated with one or more overapproximate analyzers that will produce as artifacts either generalized witnesses, i.e., witness automata that do not specify branch directives for all branch points, or residual programs, as in [27]. The final *filter* phase would be modified to return all reachability information, both paths typed as *Reach* and paths typed as *Unreach*.

4.2.3 Synergistic combinations

The synergistic combination of *may* and *must* information given in the DART algorithm can be instantiated in an ACA that does not use conditioning or generalization. The overapproximator $\overline{A}$ monitors an internal finite abstraction of the program, and is allowed to track the answers of $\underline{A}$ in *characterize* across rounds of analysis. The underapproximator $\underline{A}$ runs a guided symbolic execution based on evidence from $\overline{A}$. Depending on $\underline{A}$'s answer—i.e., does the evidence represent a feasible trace or not—, $\overline{A}$ will refine its overapproximation by internally marking its frontier (see 2.3.3). In this way, conditioning is happening internal to $\overline{A}$, but is not given explicitly.

No generalization is given, and because proving program correctness is undecidable in general, it is possible to loop between *analyze* and *characterize*, indefinitely refining an abstraction. Termination is not guaranteed by DASH. The possible intervals produced by DASH within ACA

are $\{[c, \mathit{true}]\}$ or $\{[\mathit{false}, \mathit{false}]\}$.

4.3 Cooperative analyses

The techniques in Sec. 2.4 of Chapter 2 will only be touched upon, as they are either an integral component of ACA (conditional model checking), orthogonal to ACA (algorithm selection), or descriptive of ACA itself (meta-analysis framework).

The original description of conditional model checking can be instantiated within ACA by significantly disabling most of the framework's features: removing the *generalize* case; removing all characterization; replacing interval parameters I by evidence E; and requiring that all portfolio tools be able to emit and consume partial evidence. The only informative interval possible in this instantiation is when a combination of analyzers proves unreachability—$\{[\mathit{false}, \mathit{false}]\}$. But the basic insight of conditional model checking—using previous results of differing analyzers to help focus the exploration of later analyzers—is present in each iteration of ACA.

Modeling instantiations of algorithm selection from the literature also involves disabling most of ACA's features: instead of running all analyzers within *analyzer*, an algorithm selector would choose a single one, and ACA would terminate after a single iteration. There could be two informative intervals, the same as with CEGAR: a proof of unreachability—$\{[\mathit{false}, \mathit{false}]\}$, or an interval whose lower bound contains a single reachability trace: $\{[c, \mathit{true}]\}$.

4.4 Program intervals unique to ACA

ACA offers a flexible framework into which prior work can be instantiated, but it also can compute program intervals that prior work cannot. To the best of our knowledge, no other work can compute a program interval such as $\{[c, c]\}$, where ψ-state reachability is exactly characterized by input constraints in c, outside of which, i.e., $\neg c$, there is a proof of ψ-state unreachability. Nor can prior work compute a program interval such as $\{[c, c], [d, e]\}$, where one portion of the state space is exactly characterized—$[c, c]$—while another is approximate (assuming $d \sqsubset e$). This means there is *must* information in c and d, and *may* information is given in two different ways: between d and e, i.e., the set of inputs in $(\neg d \wedge e)$, denotes a region where you *may* reach a

ψ-state; while outside of the upper bound, i.e., $\neg\overline{\mathcal{I}} \equiv \neg(c \vee e)$, you are guaranteed to not reach a ψ-state (referred to as *not-may* information in the literature). Combinations of the preceding examples are unique to ACA and offer richer characterizations of ψ-state reachability than given in prior work.

Chapter 5

Implementation of ACA

This fierce abridgement
Hath to it circumstantial branches, which
Distinction should be rich in.

Cymbeline 5.5.383–385

This chapter discusses an instantiation of the ACA framework in a tool named ALPACA, which stands for A Large Portfolio-based Alternating Conditional Analysis. ALPACA is an open-source tool available for download at `github.com/mgerrard/alpaca`. The majority of ALPACA is written in Haskell (5284 SLOC), with some portions in Java (1851 SLOC).

In discussing specific components of ALPACA's implementation, there are a few different approaches one can take: using the Haskell code given in ALPACA; detailing the operations in prose; or developing my own custom notation that models the code. I will employ a combination of the first two. The central ACA algorithm and the main data structures will be given in Haskell. Haskell is concise and readable, and as a machine language, it avoids some of the ambiguity that a pure prose presentation may introduce. Where needed, I will explain Haskell's syntax. Prose is often necessary—e.g., in motivating the selection of some algorithm, or in explaining a workaround imposed by a resource limitation—to account for what is not explicit in the code. The components written in Java will be sketched in prose, as the majority of Java code deals with manipulating data structures specific to a symbolic executor, and details of this kind are

outside of our current scope.

We begin with an overview of the general ACA algorithm within Haskell, and the main data structures used. Next we discuss ALPACA's tool portfolio—the analyzers included, running them in parallel, and adding new tools. In Sec. 5.3 we give one possible instantiation of generalizing some $\overline{\mathcal{I}}$. A method of slicing path conditions is given in Sec. 5.4. The different approaches we take to conditioning analysis tools to explore certain subspaces of a program are given in Sec. 5.5. We discuss an implementation of a modular ACA in Sec. 5.7. A brief tour of commonly-used modules in the ALPACA codebase concludes the chapter.

5.1 ACA in Haskell

We chose Haskell because it allows for a compact representation of the core ACA components, it facilitates implementations of parallel programming, and features of the language—e.g., immutability, referential transparency, pattern matching, type classes—make reasoning about subprograms easier.

The main ACA algorithm in Haskell is given in Listing 5.1, which is a nearly one-to-one implementation of the pseudocode given in Algorithm 1. The leading line is the type signature for the function `aca`; these types and others will be discussed in this chapter.

The `::` notation denotes that the function named on its lefthand side has the type signature given on the righthand side. The return type of a function in Haskell is the rightmost type in the type signature, e.g., in the case of function `aca`, the return type is a `ProgramInterval` computed in the context (monad) of `AcaComputation`—this context is discussed below.

While Haskell is a "pure" functional language (generally meaning stateful mutation is not allowed), the steps in Algorithm 1 involve a number of updates, as well as calling a portfolio of tools that must produce side effects. This seeming mismatch of functional and imperative styles is resolved by using a state transformer, which is the type given by `AcaComputation`. A state transformer is a way to combine Haskell's modeling of IO computations with its modeling of stateful computations. Both of these models are defined within a programming interface called a monad that can then be composed together; details on monads and transformers can be found in [161, 174, 142, 146]. The state transformer `AcaComputation` encapsulates important state

```haskell
aca :: Program -> ProgramInterval -> AcaComputation ProgramInterval
aca program interval = do
  result <- exploreSubspace program interval
  case result of
    UnreachableEvidence -> do
      interval' <- enforceDisjointness interval
      lastWrites interval'
      return interval'
    (ReachableEvidence ev) -> do
      interval' <- characterize ev
      program' <- condition program interval'
      aca program' interval'
    NoEvidence -> do
      interval' <- generalize interval
      program' <- condition program interval'
      aca program' interval'
```

Listing 5.1: High-level ACA function

information that is updated as ACA runs, such as the program conditioned under a new $\mathcal{I}$, as well as information that is not updated, such as parameters passed on the command line.

The state transformer `AcaComputation` is updated via recursion within function `aca`. The expression `aca program' interval'` yields a new `AcaComputation` representing the updated context, and this in turn is evaluated. In the cases of tools discovering evidence of the types `ReachableEvidence` and `NoEvidence`, there is a recursive call to `aca`. The termination condition is the case of some $\overline{A}$ finding `UnreachableEvidence`, in which—after some cleanup steps (e.g., writing to log files)—the `ProgramInterval` is returned.

The data types that appear here are `Program`, `ProgramInterval`, and three kinds of evidence. The `Program` type includes a reference to an abstract syntax tree (AST) to make conditioning simpler (see 5.5), and the path to the textual representation of this AST.

The `ProgramInterval` type is defined in Listing 5.2. The definition of data types in Haskell is similar to how a `struct` is defined in C. The `[..]` notation stands for a list of some type. So `[DisjointInterval]` is a list of `DisjointIntervals`. The `deriving (Show, Eq)` directive gives the type `ProgramInterval` a way to pretty print—via `Show`—, and a way to compare two intervals for equality—via `Eq`. This assumes the types within `ProgramInterval` also have `Show`

```
data ProgramInterval = ProgramInterval {
  disjointIntervals :: [DisjointInterval],
  inputCountMap :: CountMap,
  inputTypeMap :: TypeMap,
  spuriousSpace :: [Conjunction]
  } deriving (Show, Eq)
```

Listing 5.2: Definition of `ProgramInterval` data type

```
data DisjointInterval = DisjointInterval {
  upperBound :: UpperBound,
  lowerBound :: LowerBound,
  assumptions :: [Conjunction]
  } deriving (Show, Eq)
```

Listing 5.3: Definition of `DisjointInterval` data type

and `Eq` instances defined. Derived instances are automatically generated by Haskell (see Chapter 11 in [157]).

The `ProgramInterval` type follows the definition of $\mathcal{I}$ in Chapter 4, Def. 5, with the extra type information of input counts and types necessary for conditioning (see 5.5), the typing of false positives is given as a list of conjunctions returned by `spuriousSpace`.

5.2 Portfolio of Analysis Tools

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

The first two requirement in implementing the *analyze* specification given in 3.5.1 are guaranteed by tools competing in SV-COMP, namely: 1) that overapproximate analyzers are sound with respect to their typing, i.e., a proof of unreachability can be trusted (modulo bugs in underlying tools); and 2) in the case of a reachability proof, an analyzer can return an interpretable

- CBMC [138]
- CPA-Seq [78]
- ESBMC [163]
- PeSCo [181]
- SeaHorn [113]
- Symbiotic [194]
- UAutomizer [118]
- UTaipan [109]
- VeriAbs [79]

Figure 5.1: Portfolio of analysis tools in ALPACA

proof—SV-COMP tools must report reachability in the form of an error automaton. The final requirement is covered below in Sec. 5.5.

5.2.1 Tools Used

ALPACA instantiates the ACA framework with a tool portfolio of nine different static analysis tools shown in Fig. 5.1.

A discussion of each tool's theoretical underpinnings is given in Chapter 9. We will just note here that the tools in this portfolio cover a diversity of techniques, including abstract interpretation, automata-based model checking, bounded model checking, property-directed reachability (IC3), and symbolic execution.

5.2.2 Parallelism

All tools in the portfolio are treated as black boxes, and on each iteration of ACA, each tool's search for reachability evidence is distinct from another's, i.e., there is no communication between them. This makes running the portfolio an "embarrassingly parallel problem."

We launch all tools in parallel. By using the BenchExec library in container mode, we can ensure that the processes do not interfere with each other, e.g., by one tool calling `killall z3` as part of some cleanup process and clobbering any extant `z3` instances [29]. The portfolio runs can be terminated eagerly—stopping the other tools after one reports new evidence, or patiently—allowing all tools to run to completion (up to a time bound), each reporting potentially different evidence.

In Haskell, true parallelism must take place inside of a "pure" function with no side effects. But analysis tools produce many side effects in general. So within ALPACA, parallelism is actually achieved via the `Async` concurrency library. This library is used to take advantage of its robust exception handling needed to deal with each tool's intensive I/O operations. More details

on modeling parallelism with concurrency in Haskell can be found in Chapter 13 of [156].

5.2.3 Enlarging the Portfolio

Adding a new tool to the portfolio involves three tasks: writing a simple Python wrapper to integrate into the BenchExec framework, defining the tool according to ALPACA's definition of an `Analyzer`, and adding the tool to the command line options.

The Python wrapper tells `BenchExec` how to execute a tool, and how to interpret its output as one of three values: true, false, or unknown. Instructions for writing these wrappers are given at `github.com/sosy-lab/benchexec/blob/master/doc/tool-integration.md`. Examples of wrappers can be found at `github.com/sosy-lab/benchexec/tree/master/benchexec/tools`. At the time of writing, there are 110 modules for tools already in BenchExec's repository, so if some user would like to quickly try a tool not in the portfolio, it is possible that the first task has already been done.

The Haskell definition of an `Analyzer`, taken from `Portfolio.lhs`, is given in Listing 5.4. In supplying the definition of an `Analyzer`, the user must: create a unique name for the analysis tool as a Haskell data constructor (simply an uppercase name, e.g. `Seahorn`), supply a unique string name, provide the directory from which to execute the tool, specify command line options, declare whether it can provide a safe overapproximation of program semantics, give a default timeout in seconds, specify whether its witness type is given as branch directives or as concrete inputs (see Sec. 5.5), give a default timeout for the generalization phase, and give a default timeout for the initial iteration of ACA. The data constructor for your new tool, e.g., `Seahorn`, should be appended to the list following the definition of `AnalysisTool` in `Portfolio.lhs`.

Adding the tool to the command line options involves editing the `Main.lhs` file to include the tool's unique lowercase string identifier, and creating a case for the function `correspondingTool` in `Portfolio.lhs` to return the appropriate data constructor for the given string, e.g., "seahorn" would return the constructor `Seahorn`.

```
data Analyzer = Analyzer {
  analysisTool :: AnalysisTool,
  analysisName :: String,
  analysisDir :: FilePath,
  analysisOptions :: [Option],
  safeOverapproximation :: Bool,
  analysisTimeout :: Int,
  witnessType :: WitnessType,
  generalizeTimeout :: Int,
  initTimeout :: Int
  }
```

Listing 5.4: Definition of `Analyzer` data type

5.3 Generalization

Initially ACA attempts to condition the upper bound to avoid previously-characterized reachability traces. If that is unsuccessful, generalization is applied to compute effective conditioning and to guarantee ACA's termination. Conditioning is "effective" if some analyzer is able to produce a new piece of evidence on the conditioned program (see Def. 6).

The specification for *generalize* given in 3.5.3 require two functions be defined. The first, *choose_interval*, is implemented in ALPACA by the simple heuristic of choosing the last interval that was added to $\mathcal{I}$. The intuition behind this heuristic is that if no effective evidence was found after the last interval, I_n, enlarged $\mathcal{I}$, something about the formulae in I_n may be difficult for other analyzers to handle and by relaxing $\overline{I}_n$, the other analyzers may deal better with the simpler formula.

The second function given in 3.5.3, *relax_upper*, requires that, given some interval I, *relax_upper* returns an interval I' whose upper bound $\overline{I'}$ is strictly higher in the logical lattice, i.e., $I \sqsubset$ *relax_upper*(I). We detail how this requirement is fulfilled below.

We implement a structural generalization of the upper bound, which is always a conjunctive formula. To generalize a conjunction of n clauses, we first construct the powerset lattice over the set of conjuncts; this lattice has height n, the upper bound as the top element (at height n), and *true* as the bottom element (at height 0). Having failed to demonstrate that the element at height n constitutes effective conditioning, we determine whether the elements at height 1, i.e.,

the singleton clauses, are effective.

For those that are effective we compute the least-upper-bound and determine if it is effective; if k singletons are effective then the *lub* will be at height k. We refer to this as a "round" of a binary search on the conditioning sublattice. Each round successively narrows that sublattice, by raising the height of the bottom and lowering the height of the top. This process defines a bounded finite order and thus, generalization is guaranteed to terminate. Generalization returns the effective condition that is lowest in the lattice. If none are effective, then *true* is returned.

As an example, suppose an analysis tool found a reachability condition that is described by a conjunctive formula made up of three clauses: $x \wedge y \wedge z$. When ALPACA attempts to block the input space described by this formula by injecting `assume`$(\neg(x \wedge y \wedge z))$, suppose no analyzer can either find new reachability evidence nor prove unreachability. This triggers the generalization mechanism.

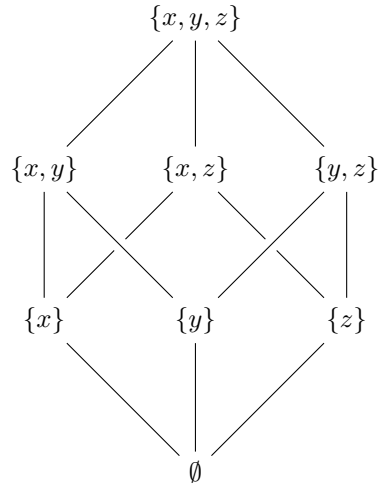


Figure 5.2: The powerset lattice over a set of three conjuncts.

The powerset lattice for x, y, and z is constructed, as shown in Fig. 5.2. The top element—$\{x, y, z\}$—did not constitute effective conditioning, so the binary search moves down to the singleton elements, and tries to see if blocking the subspace described by each of these effectively conditions the program. That is, we create three separate copies of the program, and see if, by injecting `assume`$(\neg(x))$ in the first copy, `assume`$(\neg(y))$ in the second, and `assume`$(\neg(z))$ in the third, some analyzer can now find new evidence or prove unreachability. If no conditioning is

effective at the singleton level, we must move to the bottom element, and return *true* as the generalization. However, if conditioning is successful at the singleton level, the binary search moves up the lattice, and tries blocking each of the pairwise combinations. In the case of multiple elements on the same lattice level succeeding, we break ties by selecting an arbitrary element from this level.

In our example, suppose $x \wedge y \wedge z$ was the only reachability condition found before generalization is triggered, and that generalization terminates with an overapproximator declaring unreachability of a ψ-state after blocking the clause $x \wedge z$. The interval returned will have the "generalized" upper bound of $x \wedge z$ and a lower bound containing the single path condition $x \wedge y \wedge z$.

5.4 Slicing

When some analysis tool provides evidence of ψ-state reachability in the form of an error automaton (see Def. 2), ALPACA uses this evidence to perform a directed symbolic execution in order to either confirm or refute the claimed reachability. The result of a directed symbolic execution is a program trace. This program trace may contain branches that are independent of reaching the ψ-state. If possible, we would like to eliminate these independent branches from the trace in order to obtain an interval whose lower bound safely describes a larger portion of input space. Eliminating these independent branches is referred to as slicing [208, 5].

As an example, consider the program fragment in Fig. 5.3.

```
...
if (y < 3) { ... } // no updates to x
if (x > 5) { psi(); }
...
```

Figure 5.3: Fragment of code to be sliced.

One reachable path to ψ that a fully-directed symbolic execution may yield is given by the constraint $(y < 3) \wedge (x > 5)$. However, the first branch does not influence whether or not ψ is reached—the path $\neg(y < 3) \wedge (x > 5)$ is also valid. The $(y < 3)$ branch can be "sliced" away, giving the more compact and descriptive constraint of $(x > 5)$. The slicing module attempts to determine which branches, if any, can be safely sliced away.

To determine which branches were relevant in reaching a ψ-state, we extend the symbolic execution engine CIVL [192]. We implemented a version of Xin and Zhang's dynamic slicing algorithm [210] on the CIVL representation of the replayed program trace. Xin and Zhang define a new notion of dynamic control dependence—i.e., whether a statement execution depends on some predicate at runtime—by combining statically computed information about postdominators with a stack-like structure that reflects the calling contexts of a concrete trace. We extend CIVL by constructing a control flow graph, computing postdominators, and implementing Algorithm 3 from [210]. Because this slicing algorithm detects only *dynamic* control dependence, to ensure a safe underapproximation—one whose logical formula encodes definite traces reaching a ψ-state—we also run an inexpensive static analysis. This fulfills the requirement on *slice* is that, given some path condition computed by *directed_symex*, the sliced path condition remain a safe underapproximation.

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

5.5 Conditioning

Conditioning attempts to restrict the focus of program analyzers. As discussed in Chap. 3, ACA employs two different kinds of conditioning: one to direct an underapproximate analyzer down a single execution path; the other to direct all analyzers away from some portion of a program's state space.

The first form of conditioning is a standard application of directed symbolic execution. The symbolic execution engine is guided along specific branches instead of attempting to explore

all branches. The directions come from a witness `graphml` file produced by any tool claiming reachability. The SV-COMP requirements ensure that each witness is in a standard format that can be easily parsed.

The witness file can either specify each branch direction explicitly, e.g., take the *true* branch on line 28; or the witness can provide initial concrete values, e.g., `x=4; y=-2;`, and drive execution in this way. In the latter case, we first normalize the concrete witnesses to be in the form of branch directives. If a branch is not specified in the witness file, we do not inject instrumentation at this branch, and the symbolic executor will do a standard exploration of both branches at this point. We implement the function *directed_symex* within the symbolic execution CIVL, and can be turned on with the configuration option `-direct` when using CIVL's `verify` feature. The implementation consists of 338 SLOC in Java.

The second form of conditioning is seen by all analysis tools during the *analyze* phase of Algorithm 1. This conditioning is the addition of `assume` statements that attempt to direct tools away from exploring already-analyzed subspaces. Within the `assume` statements we place the negated formulae of the upper bounds of each interval. We place the `assume`s at the top of the program—after first reading in relevant inputs—to immediately prune the state space that has yet to be explored.

The final requirement of *analyze*, given in 3.5.1, is fulfilled by this `assume`-based conditioning; namely, that conditioning removes the concrete traces described by $\overline{\mathcal{I}}$ from the semantics of the unconditioned program. This follows from the semantics of `assume` statements, which simply exit when constraints are not satisfied, i.e., the concrete traces described by the formulae in $\overline{\mathcal{I}}$ are removed upon execution of the `assume` statements.

The metadata needed correctly set up instrumentation is stored in the `inputCountMap` and `inputTypeMap` fields of the `Interval` data type given in Sec. 5.1. The instrumentation and injection of `assume`s is performed as a transformation of the AST.

5.6 AST Transformations

We embed the conditioning of $\mathcal{I}$ into a program via transformations on the program's abstract syntax tree (AST). To parse the full C99 language with GNU-extensions is a non-trivial task, and

we use the Haskell library `Language.C` to help us do so. We briefly describe the infrastructure of `Language.C` [116].

The API of the parser is found in `Language.C.Parser`; we only use the function `parseCFile` from this module. Once we have the program parsed into an abstract syntax tree, we use the API defined in `Language.C.AST` to traverse and transform the AST. The tree structure defined in `Language.C.AST` follows the grammar in Appendix A of *The C Programming Language* [132]. Tree traversal is done by decomposing the AST via pattern matching. The tree transformations are done within a state transformer, which returns new tree with each transformation. The implementations of the AST transformations are found in `Transformations.lhs` and `Transformer.lhs`.

The `Language.C` also provides a pretty-printer in `Language.C.Pretty`, which allows us to write our transformed ASTs to C files.

5.7 Implementation of Modular ACA

This subsection discusses an instantiation of modular ACA in a tool named LLAMA, which stands for Lessen Large ASTs (for) Modular Analysis.[1] LLAMA is an open-source tool written in 753 SLOC of Haskell, available for download at `github.com/mgerrard/llama`.

The module LLAMA expects to extract and embed within `main` is any valid function defined within a C program. Given a function name and a C abstract syntax tree, LLAMA creates a new abstract syntax tree that allows a program analyzer to reason about reachability properties from the callsite of the function-of-interest. Because this kind of modular analysis rips the function out of its runtime context, we have to assume that the global variables as well as the function's parameters can take on any value.

To create this new AST, LLAMA declares the parameters of the function-of-interest as globals (giving them unique names, to use in the issued call) just before `main`, collects all globals, makes each of the globals symbolic at the entry point of `main`, and finally issues a call to the function-of-interest. The resulting AST is hopefully more amenable to a modular analysis when pretty-printed to a file.

We will briefly cover LLAMA's three main components: pruning out superfluous declarations

[1] This Andean camelid tool name is clearly a backronym.

from an AST; setting up a symbolic environment appropriate for a function-of-interest (the module); and embedding this module within the call to `main`.

5.7.1 Pruning the AST

We often need to remove external declarations that are guaranteed to be unused, starting from some entry function. The reason we do this is because, after preprocessing files in large codebases, their header files include many library functions that makes the tools in ALPACA's portfolio fail—usually because these functions are not modeled in the tools.

To prune the unneeded external declarations, we first traverse the AST starting from the given entry function and collect the names of functions, global variables, structs, and enums encountered. We then retain any external declarations whose name is found in our collection of names.

5.7.2 Symbolic setup

To safely model the calling context of some function, we have to extract each of its parameters and make them symbolic, i.e., give them nondeterministic values. We must do the same with any unpruned global variables.

Making a direct type, e.g., an integer variable, is straightforward: we construct an initialization statement with the variable's identifier on the lefthand size and the SV-COMP primitive "`__VERIFIER_nondet_type()`" on the righthand side.

The more involved cases are the pointer variables and the struct variables. In the case of a pointer, we first have to declare an auxillary variable that has a unique name and make this auxillary variable symbolic; now the pointer can actually point to something in memory; we can then assign the address of this auxillary variable to the pointer. In the case of a struct, we do something similar, where we first make each of the fields of the struct hold symbolic values in memory by declaring respective auxillary variables, and then we can define the struct using these auxillary variables as the initializers.

5.7.3 Embedding

After the AST has been pruned and the symbolic environment has been prepared, the final step of embedding the function within `main` is simple. LLAMA creates a function call statement with the appropriate name and uniquely-named parameters that have been made symbolic; and then glues together the symbolic setup and this function call into a new function definition that we call "`main`". This new definition of `main` is now appended on to the pruned AST's list of external declarations.

5.8 Stepping through a run

This subsection is a whirlwind tour through ALPACA's codebase for someone who intends to actually modify or study ALPACA's source code. The tour is a stepping-through of a "typical" run of ALPACA, with the hope that this will give a rough feel of how the program is executed, helping a potential user to find their way around the codebase. To connect these internal steps with what the user sees on the terminal, we will interleave the terminal output (shown with a light grey background) with the numbered steps.

Assume we want to use the analyzers CBMC and CPA-Seq to characterize the simple C program in Fig. 5.4, named `foo.c`.

```
int main( ) {
  int x = input( );
  if (x < 0) { error( ); }
}
```

Figure 5.4: Example program `foo.c`

When running `alpaca --portfolio cpaSeq,cbmc foo.c` on this program, the following steps occur.

1. the command-line arguments are parsed in `Main.lhs`
2. the options are placed in a data structure defined in `Configuration.lhs`
3. this configuration is used to instantiate the state of ACA's computation (defined as a state transformer in `AcaComputation.lhs`)

4. an artifacts folder is created at `./logs_alpaca/foo/`; this is where conditioned programs, tool results, and logs about each iteration will be written to

5. the setup steps are done in the `runAca` function within `Analysis.lhs`, and the core logic can now be launched from the `aca` function (see Sec. 5.1) within the same file

6. the portfolio of two tools is launched from the function `exploreSubspace` within `Analysis.lhs`, which uses functions defined in `RunPortfolio.lhs` and `LaunchBenchexec.lhs`

```
* Running ALPACA on foo.c

** Iteration 1 ************

Launching tool portfolio.
```

7. suppose CBMC finds reachability evidence: $x < 0$; this evidence is verified and characterized as a logical formula via the symbolic execution engine CIVL, using code from `RunPortfolio.lhs`, `CivlParsing.lhs`, and `Characterize.lhs`

```
CBMC found reachability evidence.
(Adding reachability condition to program interval.)

Partial Program Interval:

  Partition:
    Upper Bound:
      (X_int_0[0] + 1 <= 0)
    Lower Bound:
      (X_int_0[0] + 1 <= 0)
    Assumptions:
      (X_int_0[0] - 1073741823 <= 0 && 0 <= X_int_0[0] + 1073741823)
```

8. this evidence is added to the comprehensive state characterization (`ProgramInterval`) defined in `IntervalTypes.lhs`

9. `foo.c` is instrumented to block the input space accounted for in $\mathcal{I}$ by adding `assume(!(x<0));` using code from `Transformations.lhs` and `Transformer.lhs`

```
Instrumenting foo.c
 to block space covered by partial program interval.
```

10. the tool portfolio is run on this conditioned program in a second call to the `aca` function; this time CPA-Seq declares that the remainder of the program cannot reach the `error()` statement, which is ACA's termination condition

```
** Iteration 2 ************

Launching tool portfolio.

CPA_Seq declares unreachability.

_ Terminating ALPACA ______________
```

11. the final `ProgramInterval` is displayed, the final artifacts are written, and ALPACA exits

```
Final Program Interval:

  Partition:
    Upper Bound:
      (X_int_0[0] + 1 <= 0)
    Lower Bound:
      (X_int_0[0] + 1 <= 0)
    Assumptions:
      (X_int_0[0] - 1073741823 <= 0 && 0 <= X_int_0[0] + 1073741823)
```

Note that this step-through is not at all exhaustive: it does not cover the case when ACA requires generalization (which invokes code within `Characterize.lhs` and `BinarySearch.lhs`), does not discuss all configuration options (shown with `alpaca -h`), does not list each created artifact, does not name each code module used, etc.

Chapter 6

Evaluation of ACA

We conducted a study to explore the cost and effectiveness of ACA in computing logical intervals of C programs. Our goal is to provide information about the efficiency, accuracy, and safety of the ACA framework.

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

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

RQ3: *How do the ACA components that ensure safety influence the efficiency and accuracy of ACA?*

RQ4: *How does the accuracy of ACA-computed intervals compare to prior work?*

6.1 Subject Selection

This observational study uses a selection of the SV-COMP benchmarks [195]. We consider the sequential subjects from the benchmark that have property violations. These property violations model both real failures and seeded failures. We do not consider subjects that contain no property violations, because the current ACA framework depends on the existence of at least one reachable property.[1]

[1] Any program can be artificially decomposed by injecting reachable states, e.g., at callsites, and thus amenable to analysis within ACA. Doing so across SV-COMP benchmarks is left to future work.

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 ALPACA: 352 subjects suffered from the inability to confirm a witness to a violation (a known problem in multiple SV-COMP tools), 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. During the course of our study we detected 76 SV-COMP benchmarks that either read no input or read no input on failing behaviors. We do not think that these are representative of real programs failures, since all program runs lead to failure, and keeping these in our study would inappropriately, albeit positively, bias our results. The remaining 380 subjects were run through ALPACA.

6.2 Experimental Setup

The experiments were executed on a dedicated computing cluster consisting of 10 Intel Xeon Gold 6130 CPUs (2.10GHz), each with 16 physical cores and 128G of RAM. The cluster runs on CentOS Linux release 7.8.2003. The experiments in this study were run on the `exp` branch of ALPACA from revision:

`c835094c52c2019232d379bd97e5c0928ba5d72b`.

On each `subject.c` we ran the command:

```
timeout --signal=SIGKILL 90m $ALPACA --docker -t 300
 --generalize-timeout 300 -p allDock --known-reach subject.c
```

The first three tokens of the command—`timeout --signal=SIGKILL 90m`—set a 90 minute time bound on a run of ALPACA using the GNU Coreutils program `timeout`; after 90 minutes, a `SIGKILL` signal is sent to ALPACA. We assume `$ALPACA` points to the executable path for ALPACA. The `--docker` flag runs all tools in a Docker container. Using Docker is necessary

when running ALPACA on non-Ubuntu based machines, as we make use of a restrictive mode in the library BenchExec that is only supported on Ubuntu machines (which we can model in a Docker image). The `-t 300` flag sets a 300 second time limit for each tool on each round of analysis before trying to generalize. The first round of analysis has a default value of 900 seconds, per SV-COMP time bound standards, so the `-t` flag applies to all analysis rounds after the first. The `--generalize-timeout 300` sets a 300 second timeout on exploring each lattice level during generalization. The `-p allDock` flag instantiates the portfolio with the 9 tools used in this study. Finally the `--known-reach` flag tells ALPACA that there is at least one reachability condition in the program, so any immediate unreachability proofs should be discarded as unreliable.

6.3 Results and Discussion

We report results of running ALPACA on the 380 SV-COMP C programs both in aggregated data and through a series of plots that depict the variability of informative metrics across the programs. All raw data used to build the tables and figures in this section can be found in Appendix A.

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

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

Table 6.1 reports the average, maximum, minimum, and median times, in seconds, to run ACA components *analyze*, *characterize*, and *generalize* in our study. The average time to compute a final interval is 17 minutes. This runtime is dominated by the cost of running the portfolio and generalization; each taking on average over 52% and 43% of the runtime, respectively. Not listed in the table is the cost of miscellaneous ACA internals which include constructing an abstract syntax tree, writing to log files, performing conditioning, and merging disjoint partitions. Compared to the time spent running heavy-weight analyses, the cost of these plumbing details is

Component	Avg.	Max.	Min.	Med.	% Total
analyze	543	3537	17	419	52.4%
characterize	43	848	2	7	4.1%
generalize	446	5073	0	306	43.0%
Total	1036	5400	21	840	

Table 6.1: ACA runtime (rounded to the nearest second)

negligible—less than 1% of the analysis time on average.

When all analysis tools fail to produce effective evidence after the *analyze* phase of ACA, the interval is generalized to block more of the program space—the hope being that the generalized interval is simpler, allowing some tool to find either reachability or unreachability evidence on the remaining program space. The time spent in the *generalize* phase depends on the generalization procedure used, and how aggressively generalization is applied. To limit excessive generalization time in this evaluation, we chose to be quite aggressive in how we generalized, by parameterizing ALPACA to immediately try blocking the singleton elements of the powerset lattice (discussed in Sec. 5.3). This generalization strategy is often effective in helping ACA to converge without moving the analysis to Top. For instance, on subject `transmitter.01`, after 5 iterations and 11 minutes have passed collecting evidence, generalization is triggered; and a clause that widens the interval and leads to convergence is discovered in just over a minute (67 seconds). In half of the subjects where generalization is required, the cumulative time spent in *generalize* is less than 10 minutes.

The cumulative time spent generalizing can still be costly if it needs to be called multiple times and if there are many lattice elements to explore. Both of these conditions are met on subject `minepump_spec4_product38`. Generalization is required in four separate iterations, and in the final iteration, the logical formula ALPACA needs to generalize consists of 10 conjuncts. Exploring all possible 3-way and even 2-way combinations of 10 conjuncts would be computationally expensive; it is examples such as these which led to us parameterize ALPACA to jump straight to the singleton layer in the logical lattice when generalizing. For all that, because each of the blocked conjuncts is launched with the portfolio, if we do not limit number of asynchronous threads, the system can become overloaded and begin to thrash. To prevent this overloading, we run the exploration in batches when there are many conjuncts to explore. The combination of calling

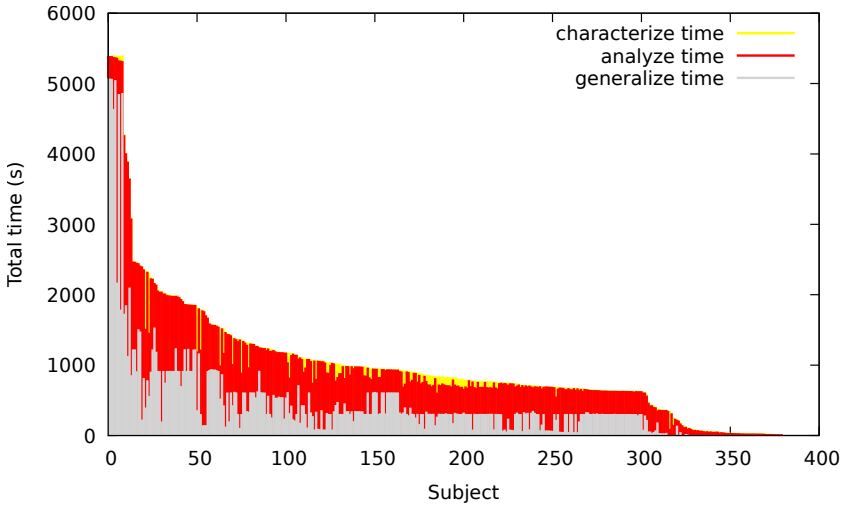


Figure 6.1: Total and Component ACA Runtime

generalize multiple times with the computational consequence of exploring many conjuncts results in subject `minepump_spec4_product38` spending over 18 minutes in generalization.

The most expensive instances of generalization can be tolerated only because of parallelization. The earlier prototype of ACA was run sequentially, and though its portfolio only consisted of two tools, generalization took over 73 minutes on average [101]. We took advantage of the fact that each problem instance is embarrassingly parallel to both increase the size of the portfolio and significantly decrease the accrued runtime.

Figure 6.1 shows the variability in runtime across the 380 programs. This is a stacked impulse plot with the time spent in *generalize* on the lowest part of the impulse, then time spent in *analyze*, then time in *characterize* at the top. The height of the entire impulse represents the total time in seconds of ACA runtime per subject. Characterize times are visible in brief bursts of yellow when zooming in on the plot.

Some subjects have massive state spaces that all tools in the portfolio struggle to explore, both in the *analyze* and *generalize* phases. On the subject `minepump_spec1_product43`, by the sixth iteration there are nine symbolic integers to account for. There are 2,147,483,648 possible values each of these symbolic integers can take,[2] so the cross product of these ranges yields a

[2]This is half the range of a `long int` in standard implementations of C. We halved the range because CIVL rewrites a path condition in the domain of ideal integers, which leads to overflow violations upon conditioning in certain edge cases. Restricting to a smaller range still yields immense state spaces.

state space with 9.7e83 possible values. Faced with this large state space, after the first piece of reachability evidence is found, tools in the *analyze* phase find no more effective evidence for the subsequent iterations; this adds up to 3385 seconds of unsuccessful attempts. Generalization on this example is necessary, and proves to be effective, but still takes up 1472 seconds overall.

The time spent in both *analyze* and *generalize* can also increase due to blocking processes within the tool portfolio. To simulate parallelism in Haskell, we use the `async` library [12] to create and destroy asynchronous threads. When effective evidence is found, ALPACA calls the `cancel` function from this library to stop outstanding threads. However, once the `cancel` signal is sent to all asynchronous analyzer threads, some process could block, e.g., if it is executing a foreign call, and cannot receive the asynchronous exception. This can lead to inflated times in some phases—primarily in *generalize*, where exploring separate lattice elements necessitate the creation of n times more asynchronous threads than *analyze*, where n is the size of the tool portfolio.

What is happening in ACA on subjects that require significant generalization? We look at the six subjects that spend over 75% of their total runtime within *generalize*: `token_ring.{08,10,13}`, `Problem03_label52`, and `Problem04_{19,55}`. All six subjects have the pattern of: finding a reachability condition on the first round of analysis that has a large number of conjuncts (greater than 16), failing to find effective evidence on the second round of analysis within *analyze*, and spending the remaining time in *generalize* trying to find an element higher in the lattice that can produce effective evidence. No effective evidence is found, and ACA times out, moving $\overline{\mathcal{I}}$ to the top value—*true*.

What can be done to reduce this time in generalization? After the first round of analysis, each of the subjects' $\mathcal{I}$ contained a single logical interval, so *choose_interval* is not relevant for these subjects. We can implement *relax_upper* to move aggressively up the lattice, that is, jump straight to the singleton values one relation below *true*; this is in fact what we do. In our implementation of ACA, *generalize* employs a call to the tool portfolio to check if the generalized bound was effective. As each element in the lattice can be explored in parallel, n elements corresponds to n parallel calls to the portfolio, which itself launches separate parallel calls for each tool. When there are many lattice elements being explored, we run the search in batches, to limit the machine becoming overloaded with too many processes. With an upper bound with a large number of

conjuncts, running these batches can be expensive.

Another possible way to reduce time in generalization when running in batches is to come up with some ordering heuristic. For instance, you could try first running lattice elements that would block the most space, e.g., choose $(x \not\equiv 3)$ instead of $(x \equiv 1)$, as the constraint given by negating the first: $(\neg(x \not\equiv 3)) = (x \equiv 3)$, constrains much more of the program space than the relaxed formula given by negating the second: $(\neg(x \equiv 1)) = (x \not\equiv 1)$. The intuition here is that the portfolio tools need to consider less of the state space in their search. We employ this ordering heuristic when generalizing. Even when aggressively relaxing the upper bound and using heuristics to choose the order in which lattice elements are explored, the six given subjects did not have any successful exploration within *generalize*, i.e., one producing effective evidence. In these cases, spending as much time trying to find some evidence other than *true* seems acceptable.

In general, characterizing and slicing a path condition will take up a trivial portion of the overall analysis time. This is to be expected, because characterization in our implementation is a fully directed symbolic execution, i.e., only *one* path is explored, followed by an inexpensive slice of the path condition. The cumulative time to characterize and slice a path condition is below 40 seconds for all but one class of subjects.

However, on 56 subjects from the ECA-RERS category, the cumulative time spent in characterization ranges from 104 seconds up to the maximum of 848. Why do these subjects spend significant time in characterization? We look at the eight subjects that spend more than 30% of their total runtime within *characterize*: `Problem10_{15,24,29,41,47}` and `Problem11_{15,29,36}`. After the call to *directed_symex* within *characterize*, each of these subjects contains a conjunct in its path condition that is a large disjunction, e.g., $((Y \equiv 6) \vee (Y \equiv 4) \vee \cdots)$. The slowdown comes from the call to *slice*, and is due to the fact that these disjunctions lead to an exponential number of SAT queries within an implication check in the slicing module. One way to avoid this slowdown is to come up with a heuristic that either disables slicing when a large disjunction is spotted, or to try to transform this disjunction into an equivalent, non-disjunctive term before running the implication check—e.g., transform $(X \equiv 1) \vee (X \equiv 2)$ to $(X \geq 1) \wedge (X \leq 2)$.

Figure 6.2 shows the variability in the number of iterations of ACA needed to reach convergence. This ranges from two to fourteen across the data set. Two iterations is the minimum in this

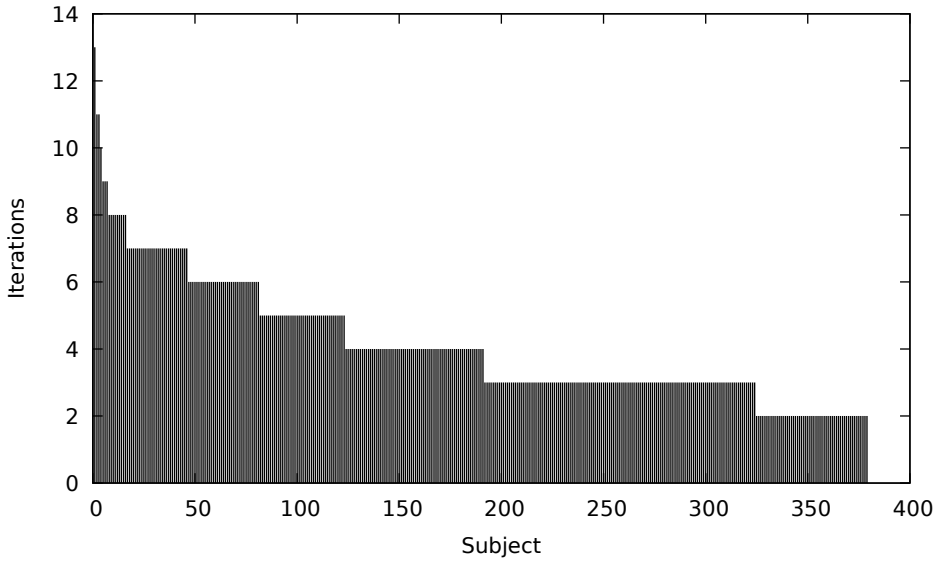


Figure 6.2: ACA Iterations to Convergence

study, the best case being: a reachable state is discovered in the first iteration, and is determined to be the only path to reachability in the second—final—iteration. This minimal case occurs on 49 subjects of the study, such as `s3_srvr_6`, in which CPAchecker quickly finds the reachability path encoded by the formula $(X \equiv 0) \wedge (Y \equiv 0)$; and in the second iteration, UltimateAutomizer determines the absence of reachability outside of this path.

Any subject containing multiple reachable paths to a given property—paths that cannot be simplified via slicing—call for multiple iterations of ACA.[3] We do not know a priori how many paths there may be to a ψ-state, so to ensure convergence, generalization is triggered after the 4th iteration. Depending on how much the logical interval is widened during generalization, there can still exist many paths to a ψ-state outside of this disjoint interval. As each iteration adds to the overall runtime, it is desirable to limit excessive iterations. Using this heuristic of triggering generalization allowed ACA to converge in seven or fewer iterations for more than 95% of subjects.

RQ1 Findings. The majority of runtime in ACA is split between the *analyze* and *generalize* components. As each round of analysis presents verification tools

[3]This statement may not hold if each of the various reachable paths are identified by distinct tools in the first iteration, but we did not see this case across the study.

with a new verification problem, the time spent analyzing is dependent on some tool in the portfolio finding effective evidence for that particular problem; and this varies across subject from a minimum of 17 seconds up to a maximum of 3537. By moving up to the singleton layer of the lattice within *generalize*, time spent trying to find effective evidence for each generalization is reasonable—less than 8 minutes on average. Running the portfolio tools in parallel on each round of analysis is necessary for an efficient implementation of ACA, or there could be an n times slowdown in finding effective evidence, where n is the number of tools in the portfolio. The *characterize* component should be inexpensive in general; the few instances we observed *characterize* taking longer could be amended by an added case within the implementation of *directed_symex*.

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

The accuracy of a final interval should be judged based on the input space it describes, i.e., how many inputs leading to a ψ-state are not characterized by the lower bound, and how many inputs that avoid a ψ-state are characterized by the upper bound. This presupposes we know the exact relation of ψ-state reachability to a program's state space, which is hard to determine. In this study, we use two proxy measures to provide information on accuracy. First, we know that any $\mathcal{I}$ with coinciding bounds is completely accurate. Second, we know that any intervals that have *true* as an upper bound are inaccurate—since we removed programs from the study that always reach a ψ-state. Intuitively, the greater the number of intervals with coinciding bounds and the fewer the number of *true* upper bounds, the more accurate the analysis.

The intervals for all 380 subjects in the study were comprised of 620 disjoint intervals. Of these, 60 (9.7%) have coinciding bounds and 109 (17.6%) have *true* values as upper bounds. The remaining 443 disjoint intervals have noncoinciding bounds whose upper bound is not *true*.

Though an interval with *true* as its upper bound indicates inaccuracy, this inaccuracy could either be the result of ACA not being able to compute a sufficiently tight interval, or it could be the result of a subject whose program paths almost always lead to a ψ-state—in which case an upper bound of *true* is a reasonable approximation. To see how often the latter is the case in our study, we use statistical symbolic execution to sample the paths with the most probability mass

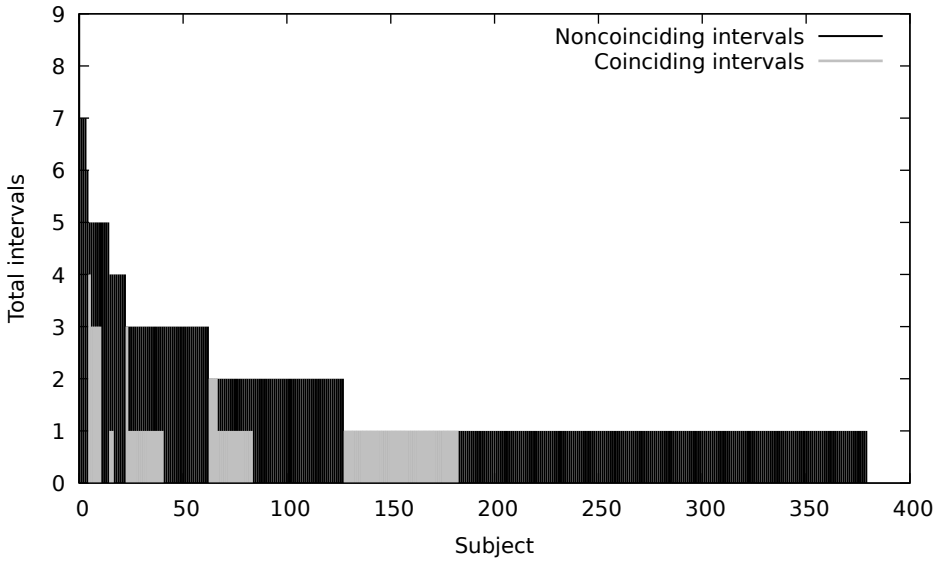


Figure 6.3: Impulse plot of interval accuracy

from each of the 109 subjects that yield *true* upper bounds; this should give us an idea of how often a ψ-state can be expected. We run SSE for 30 seconds, which allows time to account for a portion of probability mass. For 69 of these subjects—including all 53 ECA-rers variants—the collected probability mass of hitting a ψ-state is below 0.1%. For 27 subjects the probability mass is above 99.9%. The remaining 12 subjects have all probability mass that could not be accurately classified. In many cases, *true* is a close approximation of the ideal upper bound. Whether or not ACA yields a poor approximation is highly dependent on the interaction between a program's structure and the abstract domains of the analyzers in the portfolio. The cases we observed where $\overline{\mathcal{I}}$ is a poor approximation tended to have large state spaces made of an unbounded "evaluate" loop controlled by some complex condition on state variables.

Figure 6.3 plots the variability across the examples in terms of the number of disjoint intervals computed for the final interval, and whether those disjoint intervals have coinciding or noncoinciding bounds. The significant number of noncoinciding intervals corresponds to the need for generalization across the study subjects.

There are a number of subjects in Fig. 6.3 that display a mix of coinciding and and non-coinciding intervals. This indicates that the logical interval contains some disjoint intervals that exactly describe conditions for reaching a ψ-state, while other disjoint intervals contain

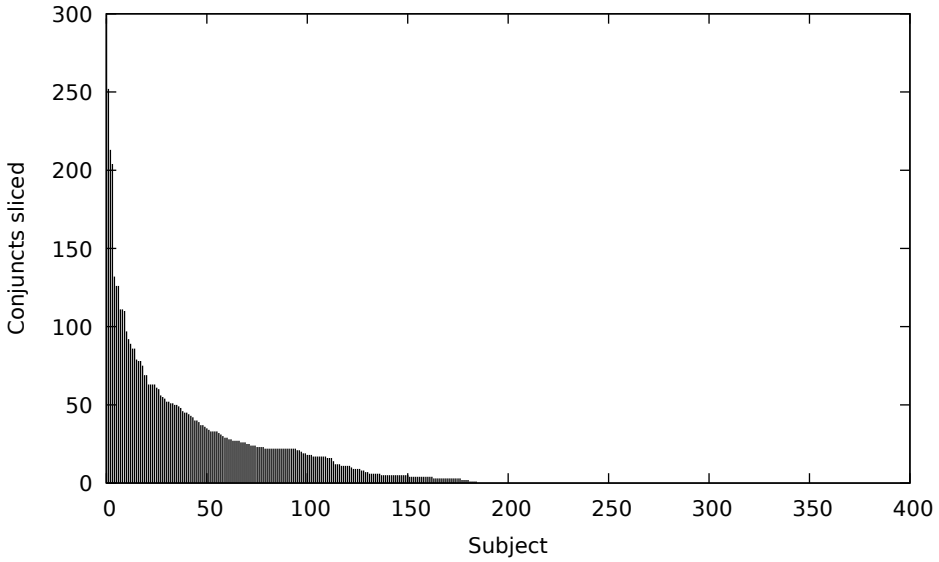


Figure 6.4: Impulse plot of conjuncts sliced

a portion of input space that cannot be determined as reaching a ψ-state or not. This is the case with `minepump_spec3_product30`, which completes with three coinciding intervals, and two noncoinciding intervals. In addition to calling *generalize* to "raise" the upper bound, `minepump_spec3_product30` is also able to "raise" the lower bound by slicing a total of of 22 conjuncts away; this slicing significantly improves an interval's accuracy.

Generalization influences the accuracy of upper bounds, but slicing influences the accuracy of lower bounds, i.e., these bounds are generalized yet they remain underapproximations. Figure 6.4 plots the number of conjuncts sliced across a run of ACA for each subject. For more than half of the programs, some slicing is performed.

In certain subjects, slicing helps to drastically simplify interval bounds, such as `test_locks_14`, where 46 conjuncts are sliced away across the run of ACA. In the first iteration, 12 conjuncts are sliced away from the path condition: this is a major help in producing a more comprehensive interval and in accelerating convergence for ACA. Without slicing, ACA would have to block the full path condition, and—as each of the 12 irrelevant branch points must be explored in all combinations—would then need to block the other 4095 ($2^{12} - 1$) path conditions to arrive at the point slicing sweeps us along to. The program `test_locks_14` models a system with 14 semaphores, and tests that these semaphores can be acquired and released correctly. Each

semaphore is represented by a symbolic value, and the main logic of the program consists in checking if a process has its respective lock before releasing that lock. This program structure means that the majority of branch conditions deal with distinct semaphores and do not influence each other—thus can be safely sliced away. Though the model in `test_locks_14` is small, it exemplifies simple boolean tests in a program that do not share data dependencies, and exemplifies how helpful slicing can be.

Slicing can also have a downstream effect on performance within *generalize*, since reducing the size of an initially coinciding disjoint interval reduces the number of conjuncts considered when the upper bound of this disjoint interval is being generalized.

The study in this chapter does not quantify accuracy in absolute terms, nor does it quantify the size of an interval in terms of their semantics, i.e., the input mass described by an interval's bounds. These quantifications are explored in Chapter 8. The study does provide evidence that even when ACA is configured with a limited form of generalization it can converge to final intervals that bound the space of ψ-state reachability to a strict subset of the input space.

> **RQ2 Findings.** The most accurate $\mathcal{I}$ is given when $\overline{\mathcal{I}} \equiv \underline{\mathcal{I}}$; we observe ACA computing this class of $\mathcal{I}$ for 9.7% of the subjects. The least accurate $\mathcal{I}$ is given when $\mathcal{I} \equiv \mathit{true}$. While we observe this case for 17.6% of the subjects, we found through a brief sampling that a quarter of these subjects have most paths leading to a ψ-state—in which case $\mathcal{I} \equiv \mathit{true}$ is a reasonable overapproximation. Slicing improves the accuracy of $\mathcal{I}$ from below by raising $\underline{\mathcal{I}}$, because the input space described by $\underline{\mathcal{I}}$ becomes more comprehensive with each conjunct sliced. We observe slicing improve the accuracy of $\underline{\mathcal{I}}$ on more than half of the subjects in this study, with some subjects slicing away hundreds of conjuncts across a run of ACA.

RQ3: *How do the ACA components that ensure safety influence the efficiency and accuracy of ACA?*

The safety of intervals computed by ACA is based on the soundness of analysis tools run within *analyze* and the use of generalization. We take the former as a given and present data on the number of times *generalize* is invoked in Fig. 6.5.

At least some generalization is required on 320 of the 380 subjects (84.2%). This is a clear

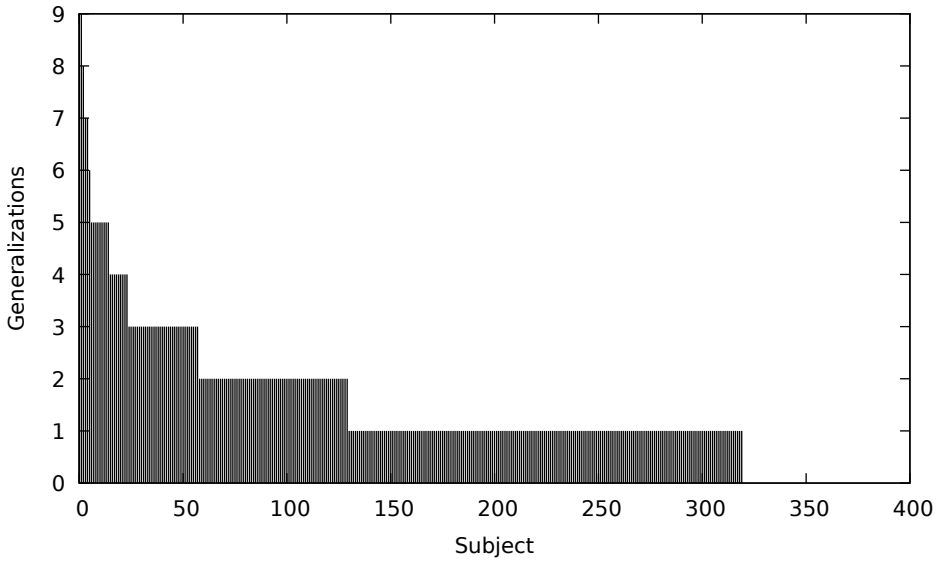


Figure 6.5: Impulse plot of generalizations

indication of the necessity of generalization for computing safe intervals. We use the number of calls to *generalize* as a proxy to measure the efficiency of the processes within ACA that ensure safety. Under this measure, the most "efficient" use of *generalize* is to call it as little as possible, as each invocation of *generalize* can be expensive. If *generalize* does need to be invoked, the most efficient implementation would immediately move $\overline{\mathcal{I}}$ to the top value in the lattice—*true*, but this can lose an inordinate amount of accuracy. With this proxy measure of efficiency, we find that 190 of the 320 subjects (59%) requiring generalization establish a safe $\overline{\mathcal{I}}$ after only one invocation. A safe $\overline{\mathcal{I}}$ is found after two to three invocations in 106 of the 320 subjects (33%), and the remaining 8% require four to nine calls to *generalize*.

We look at the two subjects with the highest number of calls to generalize (9 invocations): `minepump_spec2_product41` and `minepump_spec2_product44`. In both cases there was the pattern that, after a number of ACA rounds of collecting ψ-state reachability, conditioning is not effective, and generalization is triggered. Generalization is then required in each successive round of ACA, and each time the new $\overline{\mathcal{I}}$ produced by *generalize* yields effective conditioning. In this case, the component that ensures safety (*generalize*) actually helps to increase the *must* information computed, by effectively conditioning analysis tools to find new ψ-state reachability outside of $\overline{\mathcal{I}}$, and thus increasing $\underline{\mathcal{I}}$. Both analyses end with $\overline{\mathcal{I}} \equiv \mathit{true}$, but as we see below, this does not

necessarily mean it is a poor approximation.

How do the calls to *generalize* influence the accuracy of ACA? Any call to *generalize* will always produce an overapproximation of $\overline{\mathcal{I}}$, so unless the overapproximation is exact in describing ψ-state reachability, there will always be some degree of inaccuracy. We estimate the amount of inaccuracy introduced by *generalize* by looking at how much input space does *not* reach some ψ-state within some noncoinciding interval. We do so by running SSE—introduced in the discussion of **RQ2**—that is conditioned to stay within each of the noncoinciding intervals (i.e., does not reexamine the lower bound and does not look "outside" the upper bound) for 30 seconds, reporting the amount of input mass measured that avoids a ψ-state. These are all the program paths that do not reach ψ-states.

To clarify what we are reporting on, we present pseudocode in Algorithm 4 for our use of conditioned SSE to measure inaccuracy.

Algorithm 4 Estimating inaccuracy of $\overline{\mathcal{I}}$

```
1: for each subject where Ī ≢ I̲ do
2:     pr_mass = 0.0
3:     for each I ∈ 𝓘 where Ī ≢ I̲ do
4:         pr_mass += conditioned_SSE_¬ψ(subject, ψ, Ī, I̲)
5:     print pr_mass of subject
```

The number of subjects with noncoinciding bounds, i.e., where $\overline{\mathcal{I}} \not\equiv \underline{\mathcal{I}}$, is 320—this is the pool of subjects we will sample to estimate the probability mass of paths leading to $\neg\psi$. Of these 320 subjects, within 227 (70.9%) there was discovered less than 0.05% of probability mass reaching $\neg\psi$-states. This means that the generalized intervals that allowed for ACA's termination introduced very little inaccuracy with these 227 subjects, according to the SSE sample. As with the previous use of SSE in **RQ2** to measure programs whose $\overline{\mathcal{I}}$ is the top value in the lattice, the discovery of probability mass tended to be binary—either the vast majority miss a ψ-state, or a vast majority reaches a ψ-state. For 64 subjects (20.0%), there was discovered greater than 99.95% of probability mass reaching $\neg\psi$-states. That is, the components ensuring safety for ACA did so at the cost of introducing a large amount of inaccuracy on 20% of subjects. A small amount of subjects, 29 (9.1%) reported only grey probability mass within 30 seconds of running SSE—this is due either to the SSE tool struggling with solving a path condition—e.g., due to

embedded assumptions from $\overline{I}$ or $\underline{I}$, or from a timeout. Note that some SSE examples that collected less than 0.1% of $\neg\psi$-state mass in **RQ2** subjects may overlap with subjects here that found a small portion of ψ-state mass—i.e., these numbers are not always complementary because we only sample for 30 seconds.

> **RQ3 Findings** We observed that ACA requires generalization on 84% of subjects, and that, of these 320 subjects, most (92%) only need to invoke *generalize* three or fewer times across a run of ACA. When generalization is needed, we observed that on more than 70%, a small amount of inaccuracy is introduced by *generalize*, according to a brief sample of SSE. On 20% of the subjects, a large amount of inaccuracy is introduced in $\mathcal{I}$ by *generalize*.

RQ4: *How does the accuracy of ACA-computed intervals compare to prior work?*

By construction, ACA is more expensive in terms of time and memory than prior work—because ACA employs multiple analyzers as black-box components of its framework—, but program intervals offer the possibility of large improvements in the accuracy of ψ-state reachability characterizations. This research question compares the accuracy of program intervals generated by ACA with the accuracy of reachability characterizations generated by individual analyzers.

Recall from Chapter 4 that prior work can compute essentially two kinds of program intervals: one guaranteeing that no ψ-state can be reached, i.e., $\{[\mathit{false}, \mathit{false}]\}$; and another giving one or more proofs of ψ-state reachability, e.g., $\{[c, \mathit{true}]\}$, where a ψ-state is reachable under the input constraints of c. The subjects considered in this chapter contain at least one reachable ψ-state, so this research question will only examine the second kind of program interval produced by prior work: some accumulation of reachability characterizations in the lower bound, $\underline{\mathcal{I}}$.

We approach accuracy by measuring its dual—inaccuracy. Inaccuracy is measured in terms of an estimate of the size of the input domain for which the analysis results are inaccurate. For any given individual analyzer, inputs that are not implied by $\underline{\mathcal{I}}$ are not given a definite characterization, so inaccuracy is given by the size of the input space outside of $\underline{\mathcal{I}}$. We use the count measure (i.e., an input domain modeled by a uniform distribution) to estimate the size of input subdomains—specifically, the size of *inaccurate* subdomains. The count measure computes the ratio of the size of the input space comprising a target subdomain to the size of the total

input domain. Note that as we use the count measure, the count of $\underline{\mathcal{I}}$ must be between 0 and 1. We now describe how we measure the inaccuracy of program intervals produced by individual analyzers.

Given a program P and an analyzer A, we run A for 90 minutes using its default settings in search of reachability information, i.e., we do not modify A's internals. This run will produce a set of reachability characterizations, some of which may be false positives if A is an overapproximator. We quantify and sum this set of reachability characterizations—$\#(\underline{\mathcal{I}})$—using the model counter Barvinok [204] (see Sec. 8.2.2 for more on quantification using model counting). The inaccuracy of A on program P is then given as 1 minus this sum, i.e., $1 - \#(\underline{\mathcal{I}})$. False positives are not included in $\#(\underline{\mathcal{I}})$ by definition; their inaccuracy is implicitly characterized by the above difference. We enforce a bound on the number of inputs read at a particular program location—e.g., a read statement within an unbounded while loop—for each subject according to the maximum number of inputs read across a full run of ACA. We quantify inaccuracy within this finite domain.

The inaccuracy of ACA-produced program intervals is measured by the size of input subdomains given between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$. When ACA produces a program interval with noncoinciding bounds, i.e., $\overline{\mathcal{I}} \not\equiv \underline{\mathcal{I}}$, the inaccuracy is the difference of their quantifications: $\#(\overline{\mathcal{I}}) - \#(\underline{\mathcal{I}})$. We will use the standard notation in measure theory [199] for the "length of an interval"—$|\mathcal{I}|$—when referring to the difference in size between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$. When ACA produces a program interval with coinciding bounds, i.e., $\overline{\mathcal{I}} \equiv \underline{\mathcal{I}}$, the characterization of ψ-state reachability is exact, so the inaccuracy is 0.

Across the study, two analyzers did not produce reachability proofs: SeaHorn and ESBMC. These two analyzers are not considered in this research question, as they will always be wholly inaccurate. Of the 380 subjects in this study, nine subjects have ψ-state reachability involving floating point arithmetic. While ACA can exactly characterize these subjects, the Barvinok model counter does not support floating point constraints, so we do not consider these nine subjects. This leaves a pool of 371 subjects and seven analysis tools for this research question.

Table 6.2 reports the percentages of subjects on which each analyzer yields either a less accurate or an equivalent characterization of ψ-state reachability compared to ACA. For instance, CPAchecker computes a less accurate description of ψ-state reachability than ACA on 338 of the 371 subjects (91.1%), and computes the same accuracy on 33 subjects (8.9%). Note that

Analyzer	% subjects less acc.	% subjects same acc.
CBMC	98.7	1.3
CPAchecker	91.1	8.9
PeSCo	94.3	5.7
Symbiotic	86.8	13.2
UAutomizer	93.3	6.7
UTaipan	99.5	0.5
VeriAbs	98.7	1.3

Table 6.2: Percentage of subjects on which prior techniques yield a less accurate or equivalent characterization of ψ-state reachability compared to ACA.

no single analyzer computes a more accurate program interval than ACA. This is because the reachability information collected by a single analyzer will be included in the $\underline{\mathcal{I}}$ produced by ACA, as the underlying analyzers are employed in ACA's tool portfolio.

The benchmark includes a variety of programs including some for which existing analysis techniques, such as symbolic execution, are known to perform well. We compare with two symbolic execution based techniques: CBMC and Symbiotic—which is a wrapper for the KLEE symbolic execution tool. As Table 6.2 reports, ACA is more accurate than CBMC and Symbiotic on 98.7% and 86.3% of the experimental subjects, respectively. This is likely due to the fact that from the set of 371 subjects studied, the symbolic execution techniques performed well on programs containing on average 6.5 symbolic inputs, while the majority (58%) have an average of 14 symbolic inputs and complex control flow that is dependent on symbolic inputs.

The main takeaway from Table 6.2 is that across the considered subjects, prior work when employed on its own yields strictly less accurate program intervals than ACA the vast majority of the time. The large improvement in accuracy across subjects is both encouraging and expected, because any time ACA can either produce an $\overline{\mathcal{I}}$ that is sufficiently below the top value of *true*, or can accumulate a large amount of *must* information in $\underline{\mathcal{I}}$, the accuracy is strictly more than prior work can characterize on its own (see Chapter 4). On some subjects, ACA finds only one or two pieces of ψ-state reachability before needing to generalize to the top *true* value. In these cases, if some analyzer on its own (e.g., Symbiotic) finds some ψ-state reachability, then this analyzer can characterize the same space as ACA does with $\mathcal{I}$, whose accurate information is given only by $\underline{\mathcal{I}}$.

Table 6.3 reports on the average accuracy improvements—according to the count measure—in

	Average accuracy improvement of ACA			
Analyzer	$\lvert\mathcal{I}\rvert \equiv 0$	$\lvert\mathcal{I}\rvert < 0.01$	$\lvert\mathcal{I}\rvert > 0.99$	$\lvert\mathcal{I}\rvert \approx 0.5$
CBMC	0.9907	0.9932	3.1e−10	0.2500
CPAchecker	0.9722	0.9072	2.8e−10	0.2500
PeSCo	0.9907	0.9479	2.6e−10	0.2500
Symbiotic	0.8426	0.7534	2.4e−10	0.2500
UAutomizer	0.9352	0.9706	3.1e−10	0.5000
UTaipan	0.9352	0.9706	3.1e−10	0.5000
VeriAbs	0.9722	0.9344	3.0e−10	0.2500

Table 6.3: Average accuracy improvements—according to the count measure—in ACA over single analyzers across four sizes of the gap between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$, denoted by $|\mathcal{I}|$.

ACA over single analyzers across subjects grouped by four distinct program interval sizes, denoted by $|\mathcal{I}|$. Recall that $|\mathcal{I}|$ refers to the difference in size between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$ produced by ACA. We separate $|\mathcal{I}|$ into four groups: (1) exact program intervals where $|\mathcal{I}| \equiv 0$—this occurs when $\overline{\mathcal{I}} \equiv \underline{\mathcal{I}}$ and is observed on 54 subjects; (2) program intervals for which the difference between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$ is very small, i.e., $|\mathcal{I}| < 0.01$—observed on 167 subjects; (3) program intervals with a very large difference between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$, i.e., $|\mathcal{I}| > 0.99$—observed on 148 subjects; and (4) program intervals whose difference in size is around 0.5, observed on just 2 subjects. For instance, when $|\mathcal{I}| < 0.01$, the information in $\mathcal{I}$ produced by ACA accurately characterizes on average 94.79% more of a subject's ψ-state reachability than PeSCo can do on its own. We will now discuss accuracy gains observed in each type of $|\mathcal{I}|$.

The most accurate program interval produced by ACA is given when $\overline{\mathcal{I}} \equiv \underline{\mathcal{I}}$, whose inaccuracy is 0. Prior work cannot produce this kind of interval on its own, because for subjects with ψ-state reachability, the only possible value of $\overline{\mathcal{I}}$ is the extremal *true*. As a program interval with coinciding bounds is novel to ACA, the large increase in the count measure makes sense. For instance, suppose CBMC either finds no valid reachability evidence or its ψ-state characterization describes a small portion of the input space. If ACA produces a program interval with coinciding bounds, then $\mathcal{I}$ exactly describes the *must* and the *not may* portions of the whole input space, making the improvement in the count measure close to 1 on this subject. When a subject has large portions of its input characterized by $\underline{\mathcal{I}}$, then the increase in accuracy provided by an exact $\mathcal{I}$ is less dramatic; this was the case for many subjects on which Symbiotic was successful in finding ψ-state reachability on its own.

When the size of the "gap" between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$ is small—as with the subjects whose $|\mathcal{I}| < 0.01$—, there is again a large increase in improvement in the count measure of accuracy. This is for similar reasons as the increase in accuracy described in the previous paragraph: the majority of the input space can be accurately characterized by ACA when the "gap" denoting inaccuracy is small, whereas an individual tool can only make contributions to the lower bound.

When the "gap" between $\overline{\mathcal{I}}$ and $\underline{\mathcal{I}}$ is large—as in $|\mathcal{I}| > 0.99$—, there is still an increase in accuracy given by ACA, but the measure of this increase is minuscule: in the best case ACA yields an average increase of 3.1e−10 per subject. A large $|\mathcal{I}|$ occurs when ACA's $\overline{\mathcal{I}}$ for a subject is either the maximal *true* value, or close to *true*. In these cases, the accuracy given by ACA is comparable to prior work run on its own. The increase in accuracy can be no greater than $1 - 0.99 = 0.01$, so we cannot expect a large increase in accuracy when the gap is large. A large gap represents inaccuracy insofar as it characterizes a large portion of the input space as *may* information. But as discussed in **RQ3**, the large gap denoting *may* information often offers a decent approximation of subjects' true ψ-state reachability.

The final case of program interval sizes are for two subjects on which $|\mathcal{I}| \approx 0.5$. For these two subjects, either the individual analyzer did not find valid ψ-state reachability, making the increase in accuracy the size of the program interval; this was the case with UAutomizer and UTaipan. Or the individual analyzer found reachability evidence for one of the subjects but not the other, halving the average increase in accuracy; this was the case with the remaining analyzers.

> **RQ4 Findings** The program intervals produced by ACA offer a more accurate characterization of ψ-state reachability compared to running the component analyzers on their own. For any of the seven individual analyzers, an ACA-produced program interval $\mathcal{I}$ strictly improves the accuracy of ψ-state characterization on 86–99% of the considered subjects. The average improvement in the count measure of accuracy is dramatic for the 60% of subjects whose $|\mathcal{I}| \equiv 0$ or $|\mathcal{I}| < 0.01$. The improvement in accuracy is necessarily slight when $|\mathcal{I}|$ is large.

6.4 Threats to Validity

Internal validity

The tools underlying ALPACA 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 factors that may influence their performance and this may impact the performance of ACA. This is a challenging problem in general—benchmarking the performance of static analysis tools remains an open problem [88, 183]. We have conducted extensive testing and post-analysis of the computed interval data to ensure that it is safe and have only used static analyses that have proven to be robust in the SV-COMP competitions [196, 197]. The SV-COMP benchmark suite, while limited, is updated annually to reflect the challenges to static analyses found in the broader population of C programs.

We observed that on a few subjects the simplification time in slicing increased significantly when certain disjunctive forms appeared. This performance issue inflates runtimes, so improving the implementation would only positively impact the findings in this chapter.

External validity

We caution the reader in making conclusions about the external validity of our study. Using SV-COMP programs constitutes a common sample used in evaluating C static analysis tools, and its use promotes the replicability and reproduction of our study. The subjects included in the benchmark, and the tools taken from the corresponding competition, represent a diverse set of programs and analyzers, but clearly a broader amount of subjects and tools is needed to make a strong claim on the generalizability of our results to other C programs.

Chapter 7

Case Study of `chrony`

This chapter describes an exploratory case study in applying ALPACA to portions of a medium-sized system outside of the SV-COMP benchmarks.

The SV-COMP benchmarks include a large number of diverse C programs, but as a fairly stable benchmark, the competing tools may overfit their competition contribution to these benchmark subjects. As all tools in ALPACA's portfolio come from the set of SV-COMP tools, we want to explore the feasibility of computing program intervals in a software system not included in the benchmarks. We choose the `chrony` [1] system of software—an implementation of the network time protocol—as it is a medium-sized C system (55 C files with 21,926 SLOC) embedded with `assert` statements. Assertions contain conditions expected to hold on all program executions. The semantics of an `assert(c)` statement, where `c` is some boolean predicate, are: `if(c): skip; else: error;`, i.e., an assertion violation is equivalent to reaching an error state. We define the ψ-state to be a violation of an assertion, and explore how we can compute informative program intervals within `chrony`.

We summarize `chrony` in Sec. 7.1, explain our methodology for analyzing portions of its codebase in Sec. 7.2, discuss what we observe applying this methodology to four subsystems of `chrony` in Sec. 7.3, discuss threats to validity in Sec. 7.4, and conclude in Sec. 7.5.

7.1 chrony

The chrony codebase is an implementation of the network time protocol (NTP), which is a protocol for synchronizing clocks between computer systems. Implementations of NTP have been deployed on systems since the mid 1980's and are still in use today. The chrony implementation is the default NTP client in the Red Hat Linux and SUSE Linux distributions, and is available on many Linux distributions as well as macOS.

The chrony codebase has been rigorously tested and has passed recent a Critical Information Infrastructure audit [2]. However, this audit is limited in scope—involving three testers performing manual code checks and applying fuzzing techniques to try to crash the software, which can only exercise a small portion of chrony's input space. The input space of chrony is large, as it must handle nondeterminism from a range of environmental factors, such as "intermittent network connections, heavily congested networks, changing temperatures (ordinary computer clocks are sensitive to temperature), and systems that do not run continuously, or run on a virtual machine." [1] There is also a daemon that can update system parameters at runtime. Each of these nondeterministic sources makes the space to consider in chrony massive.

As a well-designed modular piece of software among a multitude of other NTP implementations, we expect portions of its code to be reused in contexts outside of its designed intent. To safely reuse these modules in another codebase, it is necessary to discover the implicit invariants that these interfaces define. We examine chrony version 3.2.

7.2 Methodology

This section describes how we run a modular ACA to reason about portions of the chrony codebase. We employ modular ACA because chrony is too large and complex for effective whole-program analysis with any combination of the tools in ALPACA, so we examine subsystems of chrony. We discuss how program intervals can be interpreted in this modular context, how we choose the subsystems to examine, how we model nondeterminism in these subsystems, how we propagate information in $\mathcal{I}$ across function boundaries, and conclude with the hardware and software on which the case study was run.

7.2.1 Program intervals in modular ACA

A top-down, whole-program analysis is not feasible with ALPACA over `chrony`, so we attempt a bottom-up approach of first analyzing the "leaf" functions that contain an assertion, then expanding the calling context and analyzing this expanded context given information from the leaf function's computed program interval.

If we run ACA in function f to compute $\mathcal{I}_f$, and function g calls f, how can we use $\mathcal{I}_f$ while analyzing g? First of all, we do not need to reanalyze f, as the formulae in $\overline{\mathcal{I}}_f$ characterize the *may* conditions on ψ-state reachability in f. The means all inputs implied by $\overline{\mathcal{I}}_f$ can be ignored in the analysis of g.

Suppose now the caller function g computes program interval $\mathcal{I}_g$—how do we interpret its bounds $\overline{\mathcal{I}}_g$ and $\underline{\mathcal{I}}_g$? It depends. If the analysis of f yields a program interval with coinciding bounds, i.e., $\underline{\mathcal{I}}_f \equiv \overline{\mathcal{I}}_f$, then $\mathcal{I}_g$ encodes a standard program interval: $\overline{\mathcal{I}}_g$ characterizes *may* information and $\underline{\mathcal{I}}_g$ *must* information. If the analysis of f yields a program with noncoinciding bounds, i.e., $\underline{\mathcal{I}}_f \not\equiv \overline{\mathcal{I}}_f$, and we do not reanalyze $\overline{\mathcal{I}}_f$, then $\overline{\mathcal{I}}_g$ still characterizes *may* information, but the meaning of its lower bound $\underline{\mathcal{I}}_g$ is less clear. The lower bound $\underline{\mathcal{I}}_g$ encodes conditions under which there *may* exist ψ-state reachability in f, and so the lower bound in this case is meaningless with respect to computing definite *must* information.

Propagating the upper bound $\overline{\mathcal{I}}_f$ across calling contexts amounts to a form of weakest precondition reasoning [17, 130], where $\overline{\mathcal{I}}_f$ encodes the necessary but not sufficient conditions of reaching a ψ-state in f from the context of g. We use this approach in the case study, and describe how we propagate $\overline{\mathcal{I}}$ in Sec. 7.2.4. It is also possible to propagate $\underline{\mathcal{I}}$ across calling contexts, though in this case the meaning of any final $\overline{\mathcal{I}}$ becomes undefined—we look at this possibility at the end of Sec. 7.3.

To summarize, program intervals with coinciding bounds can propagate both *may* and *must* across calling contexts, but in the presence of a noncoinciding program interval, either *may* or *must* information can be propagated, but not both simultaneously.

7.2.2 Sample selection

We want to explore using modular ACA starting from the leaves containing potential ψ-state reachability—those functions that contain an assertion—and successively expanding calling contexts. To this end, we choose from a pool of 1008 candidates in `chrony` whose callchains are of length four and whose final called function is `assert`. For instance, function f calls function g, which calls function h, which calls `assert`, is a valid 4-length callchain. We discard callchains whose outermost function is `main`, as this would amount to a (near) whole-program analysis. From these candidates, we select four callchains randomly by calling `shuf -n 4 callchains.txt`, where `callchains.txt` contains quadruples of function calls (and associated file names). The extracted functions and type definitions of the sample sum to 637 SLOC.

7.2.3 Modeling nondeterminism

Nondeterminism is used to express arbitrary choices. This can be used to model, for instance, that the value of some input from a keyboard could be any ASCII character. An analysis tool could then reason about each distinct possibility resulting from this nondeterminism. The nondeterminism needs to be communicated to verification tools in some way, as there is no nondeterministic primitive in C. We communicate this nondeterminism via annotation primitives made up of C expression that tools in ALPACA understand. These annotation primitives are of the form `__VERIFIER_nondet_{type}`, e.g., when some tool sees a call to `__VERIFIER_nondet_char()`, it can assume the return value is any of the values of type `char`.

Given a function f, we model its parameters, any global variables referenced, and any functions external to f, nondeterministically. Any external function is assumed side-effect free and returns a nondeterministic value of its return type. This is a "soundiness" [148] assumption taken from SV-COMP rules on externally defined functions; we mitigate this threat to validity by looking at the source of the modeled functions and validating that any side effects do not mutate variables involved in ψ-state reachability.

To model a nondeterministic `struct`, we give any fields that are used in f a nondeterministic value, and leave the remaining fields uninitialized. An arbitrary binary choice, such as a pointer being null or not, can be modeled with an `if-else` statement guarded by a nondeterministic

conditional variable.

LLAMA, described in Chapter 5 is used to automatically prepare all level 1 functions (those containing calls to `assert`) for ALPACA.

7.2.4 Embedding $\overline{\mathcal{I}}$ at callsites

As discussed in Sec. 7.2.1, within a modular ACA context, only $\overline{\mathcal{I}}$ or $\underline{\mathcal{I}}$ may be propagated across calling contexts in the presence of a program interval with noncoinciding bounds. We choose to propagate $\overline{\mathcal{I}}_f$ by embedding `assert(`$\neg\overline{\mathcal{I}}_f$`)` above any of f's callsites in some calling function g. What this does is communicate within g that if the input variables satisfy $\overline{\mathcal{I}}$, then a ψ-state *may* be hit.

The procedure we follow to run ACA in successively expanding calling contexts is given by these steps:

1. $d \to 1$
2. Compute $\mathcal{I}_d$ for depth-d function f_d.
3. If $d \equiv 3$, return $\mathcal{I}_3$.
4. Embed `assert(`$\neg\overline{\mathcal{I}}_d$`)` before all calls to f_d in f_{d+1}.
5. Go to Step 2.

For each subsystem discussed in the following section, we will look at the intermediate $\mathcal{I}$'s computed in Step 2, along with the final $\mathcal{I}$.

7.2.5 Setup

This case study was run on Ubuntu 20.04 using Linux kernel 5.8.0-7642, with an AMD Ryzen 7 2700X eight-core processor. Each subsystem was run on the `exp` branch of ALPACA from revision:

`c835094c52c2019232d379bd97e5c0928ba5d72b`.

ALPACA was run on each subsystem without any parameters, that is, as `alpaca subsystem1.c`.

7.3 Discussion

In this section we look at four different subsystems in `chrony` and make observations about program intervals computed in different calling contexts. We consider the function that calls `assert` to be a "depth 1" function, and so use the notation that $\mathcal{I}_1$ is the program interval computed at depth 1, $\mathcal{I}_2$ the interval at depth 2, etc. We conclude with observations over all four subsystems.

7.3.1 Subsystem 1

The first subsystem we consider is the callchain: `CLG_initialize` (in `clientlog.c`) calls `expand_hashtable` (in `clientlog.c`) calls `ARR_GetElement` (in `array.c`) calls `assert`.

The assertion within the function `ARR_GetElement` is checking that an index references a valid position in a bespoke array data type defined in `chrony`. ALPACA is able to compute a program interval with coinciding bounds, exactly characterizing *may* and *must* ψ-state reachability for all contexts in which `ARR_GetElement` is called.

In the function `expand_hashtable`, there are two calls to `ARR_GetElement` within two separate unbounded `for` loops. There are two additional assertions made in the scope of `expand_hashtable`: one before any calls to `ARR_GetElement` and the other in the second `for` loop—checking for the non-nullity of a hash table record. ALPACA is able to find reachability conditions leading to the first assertion, which involves variables not mentioned in $\overline{\mathcal{I}}_1$. ALPACA is not able to find effective evidence after conditioning analyzers away from the first reachability condition and must generalize its program interval. After generalization, unreachability is proved and $\mathcal{I}_2$ is returned with noncoinciding bounds.

In the function `CLG_Initialise` there is one call to `expand_hashtable`. Possibly hitting a ψ-state at this callsite requires two conditions to hold: that a certain file exists and that a variable be nonnegative. That the second condition holds is guaranteed by its typing as an `unsigned int`. ALPACA computes a program interval $\mathcal{I}_3$ with coinciding bounds that denotes a ψ-state *may* be reached when the `ClientLog` file exists and variable `max_slots` is nonnegative (the characterization in CIVL includes the redundant constraint on `unsigned int` types).

7.3.2 Subsystem 2

The second subsystem we consider is made up of the callchain: `post_init_ntp_hook` (in `main.c`) calls `NSR_ResolveSources` (in `ntp_sources.c`) calls `resolve_sources` (in `ntp_sources.c`) calls `assert`.

The function `resolve_sources` includes an assertion that checks if a pointer to a `struct` has been initialized. This can be exactly characterized—running ALPACA yields a program interval $\mathcal{I}_1$ with corresponding bounds.

Within the function `NSR_ResolveSources`, a single call to `resolve_sources` is given within two `if` branches guarded by two distinct nondeterministic variables. ALPACA determines that $\mathcal{I}_1$ can never be satisfied in the context of `NSR_ResolveSources`, yielding a program interval of $\mathcal{I}_2 \equiv \{[\mathit{false}, \mathit{false}]\}$ What does $\mathcal{I}_2$ tell us in the context of this subsystem? It guarantees that no invocation of `NSR_ResolveSources` will result in an assertion violation.

This guarantee holds for invocations of `post_init_ntp_hook`, which—containing no other assertions itself—also yields the program interval encoding ψ-state unreachability: $\mathcal{I}_3 \equiv \{[\mathit{false}, \mathit{false}]\}$.

7.3.3 Subsystem 3

The third subsystem we consider is is made up of the callchain: `post_init_ntp_hook` (in `main.c`) calls `NSR_RemoveAllSources` (in `ntp_sources.c`) calls `clean_source_record` (in `ntp_sources.c`) calls `assert`.

Within `clean_source_record` there is an assertion checking for the existence of a pointer field in a `struct`. As with the depth 1 function in the preceding subsystem, ALPACA computes an exact $\mathcal{I}_1$—one with coinciding bounds.

In the context of `NSR_RemoveAllSources`, the call to `clean_source_record` is made within a `for` loop that is nondeterministically bounded. But ALPACA can prove that $\mathcal{I}_1$ cannot be satisfied at the callsite within the `for` loop and returns the same interval as with Subsystem 2: $\mathcal{I}_2 \equiv \{[\mathit{false}, \mathit{false}]\}$. And as with Subsystem 2, ALPACA run over the caller function `post_init_ntp_hook` yields this same interval for $\mathcal{I}_3$.

7.3.4 Subsystem 4

The fourth subsystem we consider is the callchain: `SRC_DumpSources` (in `sources.c`) calls `open_dumpfile` (in `sources.c`) calls `source_to_string` (in `sources.c`) calls `assert`.

The function `source_to_string` uses a `switch` statement to select an appropriate subfunction for the enumerated types of a `struct` field. If the field does not hold a value specified by the `enum`, the fall-through `default` case is an assertion violation. This assertion may appear strange, because all enumerator values are given in the cases—so how could the `struct` field hold a value outside of this range? But in C, it is legal to assign a value to some enumerator type that is not found in the enumerator list. ALPACA characterizes this strange possibility of the default case exactly, yielding program interval $\mathcal{I}_1$ with coinciding bounds.

Within function `open_dumpfile.c`, there are two callsites for `source_to_string`. ALPACA discovers three different pieces of reachability evidence to $\overline{\mathcal{I}}_1$, each at the second callsite. Each reachable characterization involves a symbolic variable not taking the value of '`r`' (ASCII code 114), denoting read-only access to a file. The full formulae for the three reachability conditions are given in the lower bound $\underline{\mathcal{I}}_2$ in Table 7.1.

No more effective evidence is found, and the program interval is generalized to a non-Top value. After this generalization, the remainder of the state space is proven to not reach a ψ-state, and ALPACA returns with $\overline{\mathcal{I}}_2$ encoding three different ways of reaching a ψ-state in `open_dumpfile.c`.

In the calling context of `SRC_DumpSources`, the single callsite of `open_dumpfile` is within an unbounded `for` loop. ALPACA computes a program interval $\mathcal{I}_3$ whose upper bound denotes that when there are one or more sources to save to file, and when $\overline{\mathcal{I}}_2$ is satisfied, a ψ-state *may* be reached.

7.3.5 Observations

We summarize the program intervals computed at the function boundaries for each subsystem in Table 7.1. Subsystems 1–4 are given in the columns. For each subsystem, the upper bound computed by ALPACA in the depth i function is given in the $\overline{\mathcal{I}}_i$ row, and its corresponding lower bound, $\underline{\mathcal{I}}_i$, is given in the row below. As Subsystem 4 has a lengthy $\overline{\mathcal{I}}_2$, we just use its name in

	Subsys. 1	Subsys. 2	Subsys. 3	Subsys. 4
$\overline{\mathcal{I}}_1$	$A \leq B$	$F \not\equiv 0$	$G \equiv 0$	$H \not\equiv 1 \wedge H \not\equiv 0$
$\underline{\mathcal{I}}_1$	$A \leq B$	$F \not\equiv 0$	$G \equiv 0$	$H \not\equiv 1 \wedge H \not\equiv 0$
$\overline{\mathcal{I}}_2$	$2 * C \leq D$	*false*	*false*	$(H \equiv 0 \wedge I \not\equiv 0 \wedge (1 \leq J - K)$ $\wedge L \not\equiv 0 \wedge M \not\equiv 114)$ $\vee(N \not\equiv 0 \wedge H \equiv 0 \wedge L \not\equiv 0 \wedge M \not\equiv 114)$ $\vee(P \not\equiv 0)$
$\underline{\mathcal{I}}_2$	$2 * C \leq D$ $\wedge D \leq 0$	*false*	*false*	$(H \equiv 0 \wedge I \not\equiv 0 \wedge (1 \leq J - K)$ $\wedge L \not\equiv 0 \wedge M \not\equiv 114)$ $\vee(N \not\equiv 0 \wedge H \equiv 0 \wedge L \not\equiv 0 \wedge M \not\equiv 114)$ $\vee(P \not\equiv 0 \wedge H \not\equiv 0 \wedge L \not\equiv 0 \wedge M \not\equiv 114)$
$\overline{\mathcal{I}}_3$	$E \not\equiv 0 \wedge 0 \leq D$	*false*	*false*	$\overline{\mathcal{I}}_2 \wedge Q \geq 1$
$\underline{\mathcal{I}}_3$	$E \not\equiv 0 \wedge 0 \leq D$	*false*	*false*	$\overline{\mathcal{I}}_2 \wedge Q \geq 1$

Table 7.1: Computed upper and lower bounds for each depth i function across subsystems.

$\overline{\mathcal{I}}_3$ and $\underline{\mathcal{I}}_3$ rather than relisting the formulae. Each capital letter in a formula refers to a distinct symbolic variable. For instance, variable H in Subsystem 4 is found in $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$; while variable A in Subsystem 1 only appears in $\mathcal{I}_1$. Although the subsystems examined included the C data types `int`, `char`, and `unsigned int`, we do not make the distinction between types in the table.

Across the four subsystems in this case study, an application of modular ACA was able to propagate nontrivial ψ-state reachability information across function boundaries in most cases. In Subsystem 1, the information relating symbolic variables A and B in the depth 1 function is lost, as a new assertion in the depth 2 function—formulated by variables C and D—ends up composing its $\mathcal{I}_2$; this leaves A and B as free variables.

In two cases—Subsystems 2 and 3—ALPACA proves that assertion violations are unreachable in the two outermost function contexts. This is the best case, assuming an assertion violation always reachable in a depth 1 function.

In the other two cases, the program interval computed on the depth 2 function, $\mathcal{I}_2$, has noncoinciding bounds. In Subsystem 1, the upper bound of $\mathcal{I}_3$ depends on some file existing. Depending on the expected probability of that file existing, the approximation in $\overline{\mathcal{I}}_3$ computed by ALPACA is more or less helpful. For instance, if the file usually exists, the characterization in $\overline{\mathcal{I}}_3$ is not very informative, as we are told that with the usual case of $\overline{\mathcal{I}}_3$, there *may* be an

assertion violation. But if the file never exists when the function is called; then $\overline{\mathcal{I}}_3$ tells us we will never hit an assertion from the context of the depth 3 function. In Subsystem 4, the upper bound $\overline{\mathcal{I}}_3$ becomes slightly more constrained in the outermost function, and the upper bound can be satisfied if there is one or more sources to write to file, and any of the conditions in $\overline{\mathcal{I}}_2$ are satisfied.

For ACA to produce informative program intervals in the modular context of this case study, it is desirable to propagate intervals with coinciding bounds to other contexts. Noncoinciding bounds represent some degree of overapproximation in the characterization given by $\overline{\mathcal{I}}$. In all four cases, the first computed program interval, $\mathcal{I}_1$, has coinciding bounds. This is helpful, as it allows calling contexts to reason over the exact conditions describing ψ-state reachability at the callsite.

This case study applied program intervals in such a way that sound *may* information is propagated across boundaries via $\overline{\mathcal{I}}$, as described in Sec. 7.2.4. But it is also possible to imagine propagating the sound *must* information in $\underline{\mathcal{I}}$ across function boundaries. To do so, you must give full definitions (or exact output specifications) of functions—so that no unsound overapproximation is introduced in $\underline{\mathcal{I}}$, and embed `assert(`$\neg\underline{\mathcal{I}}$`)` at the callsite. If that assertion is violated in an outer context, there is definite proof of reachability in the outer context. Complementary to our case study, this would produce a succession of propagated lower bounds: $\underline{\mathcal{I}}_1$, $\underline{\mathcal{I}}_2$, $\underline{\mathcal{I}}_3$.

7.4 Threats to validity

We examined a small sample of the subsystems that make up the `chrony` codebase, and do not claim that the four we examined are representative of all of `chrony`. This case study does suggest that ALPACA can be applied in a modular way to portions of medium-sized systems outside of SV-COMP.

One threat to validity is the assumption we make that functions external to the one being examined are side-effect free. This is a "soundiness" [148] assumption made to make the application of analysis tools more effective. To mitigate this threat, we examined by hand that no variables relevant to ψ-state reachability are mutated, but this check could be automated by a cheap static analysis that tracks sets of variables that have been written to. However, if

there is a large amount of mutation in global state involving ψ-state reachability, then a modular approach is necessarily limited in its effectiveness.

Another threat is in modeling nondeterminism for the ALPACA tools. While we try to apply correct transformations to model arbitrary choices with SV-COMP primitives, as the process is semi-automated it is possible we have altered the semantics of the programs. To mitigate for this threat we check that the program intervals computed make sense applied to their respective original functions.

The computed constraints are fairly simple and in all but one case only a few conjuncts long. We need to see if this application of modular ACA is effective when contexts encode more complex constraints.

7.5 Conclusion

This chapter presented an exploratory case study in applying modular ACA to the software system `chrony`. We looked at subsystems of `chrony` consisting of function callchains ending with an assertion. We reused the upper bound computed by ACA in one calling context to compute informative program intervals in an expanded calling context.

On a small sample of four subsystems, we found that nontrivial ψ-state reachability can be propagated across function boundaries in most cases. Using our modular approach, either one or the other of *may* or *must* information can be passed from one context to another.

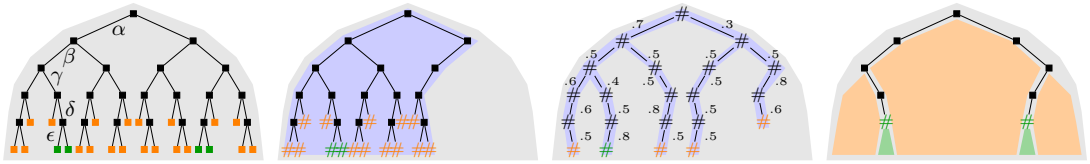
Chapter 8

Conditional Quantitative Analysis

In this chapter of the dissertation,[1] we leverage the framework of ACA to address the simultaneous challenges of scalability and accuracy in computing likelihood estimates of reaching a program state. By factoring out the parts of the state space that do not need to be explicitly quantified from what remains, a fine-grained quantitative analysis can be focused on a potentially much smaller portion of a program's state space. The portions that do not need to be explicitly quantified can be characterized as such using guarantees computed by overapproximating analyzers; these characterizations serve as a conditioning to drive underapproximators to explore the remaining state space. The scalability of the former helps focus the accuracy of the latter.

Imagine a program that contains 24 distinct execution paths, and you are interested in quantifying the likelihood of reaching a set of ψ-states that—unbeknownst to you—is shared by 4 of these paths. To solve this problem, we introduce the idea of *conditional quantitative analysis* (CQA). This analysis takes as input a program interval $\mathcal{I}$ (output by ACA), which delineates a subspace of the program; this characterization is used to condition precise analyses to consider only the portions of the program where these 4 paths may lie. If other portions of the program are guaranteed to avoid ψ-states, we do not want to waste time exploring or quantifying each path in this subspace.

[1] This chapter is a variant of the 2020 TSE paper "Conditional Quantitative Program Analysis" [100] that has been integrated into the dissertation by adopting common terminology and reducing redundancy. This work was done in collaboration with colleagues from Imperial College London, Mateus Borges and Antonio Filieri. Mateus and Antonio developed the quantitative analyses. Mateus extended the CIVL symbolic execution engine to implement probabilistic and statistical exploration strategies described in [99] and [95].



(a) Example State Space (b) PSE (c) SSE (d) CQA

Figure 8.1: Comparing state-space exploration and path quantification costs

Fig. 8.1a depicts the state space of this program. The program is visualized as an execution tree, where nodes represent branch points, and edges one of the true or false branches taken. Each path from the root node to a leaf node represents a feasible program execution; each subpath in the tree is associated with a conjunction of constraints denoting the set of inputs that follow the same branch sequence during program execution. The execution path common to all diagrams in Fig. 8.1 is given by the inputs satisfying $\alpha \wedge \beta \wedge \gamma \wedge \delta \wedge \epsilon$, annotated in Fig. 8.1a. Execution paths reaching a ψ-state have green leaves; those that do not are colored orange.

The operation of three different forms of quantitative program analysis on the state space in Fig. 8.1a is depicted in Fig. 8.1b–8.1d. These analyses will be sketched here and discussed in more detail in Sec. 8.2. The first two analyses we will consider are probabilistic symbolic execution [99] and statistical symbolic execution [95].

One possible run of probabilistic symbolic execution (PSE) is shown in Fig. 8.1b. At the termination of each feasible program execution, the collected path condition is quantified using a model counter, which counts the number of distinct solutions that satisfy a propositional formula (e.g., a path condition). In our context, calling a model counter yields the number of inputs that flow down an execution path. The count measure of executing this path is given by the resulting count divided by the count of the entire input domain. In Fig. 8.1, calls to a model counter are given by a '#' (green representing the count of inputs on a path reaching a target state; orange, the count on a non-reaching path).

There is a systematic sweep of the state space in Fig. 8.1b, which allows much of the left subtree—including two paths reaching target states—to be quantified; the explored state space is shaded in blue. But there is a significant portion of the right subtree that cannot be analyzed due to exhaustion of resources. There are also a large amount of calls to the model counter (one

for every path condition), each of which can be expensive.

A possible run of statistical symbolic execution (SSE) is given in Fig. 8.1c where computed branch probabilities annotate both explored outcomes and the states along paths reaching those outcomes. Most importantly, SSE *biases* the exploration towards the paths with the most *unexplored* input mass first. Note that some states missed by the systematically exhaustive technique, namely the rightmost path in Fig. 8.1c, are quantified using a statistical approach.

Three of the four distinct path conditions leading to the target states are missed, but the path reaching a target state that has the most input mass is quantified. Though the statistical procedure halted before the program was explored exhaustively, the fraction of the total input mass is significantly higher due to path sampling and pruning.

In constrast to both PSE and SSE, the framework of CQA only needs to explicitly quantify the green regions of Fig. 8.1d and in many cases the remainder of the state-space can be quantified implicitly—using the laws of probability and simple arithmetic. Thus motivated, we now look to the kinds of guarantees required in safety-critical systems.

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 [186], industrial robotics [127], personal care robotics [129], railway [92], automotive [128], and medical software [125]. Meeting these standards is essential, but they present substantial verification and validation challenges above and beyond those of traditional software [117].

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.[164] 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 [186], robustness evidence, such as that which is

achieved using fault-injection to meet automotive standards [128], and reliability evidence, such as that which is required to certify functions to IEC 61508 safety integrity levels (SIL) [126].

The increasing cost-effectiveness of automated formal methods and static analyses has led certification standards, e.g., DO-333[187], and researchers to explore the types of evidence they can contribute to safety arguments to complement evidence from testing, e.g., [65]. 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 [60], or they cannot.

In this chapter, 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 [139, 140]. 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,[2] 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 [46] 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 [170], 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.

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

[2]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.

observe the internal behavior of software, equivalence can be detected. This is precisely what symbolic execution techniques do [135] and reliability-focused extensions to symbolic execution can quantify the probability mass of a set of equivalent inputs [99, 94, 95]. 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 [117]. 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* [101]. 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 chapter, 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 [122, 216], 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,[3] the static analyses we focus on in our work 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., [187].

Quantitative static program analysis has been studied for more than two decades, e.g., [176, 162], 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., [133, 48, 51, 215], and increasingly scalable techniques for model counting of logical formulae, e.g., [18, 15, 108, 205, 152, 13, 37], to produce several families of techniques which we term *probabilistic symbolic execution* (PSE) [99, 94] and *statistical symbolic execution* (SSE) [95].

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

[3]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., [144, 4, 216]).

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., [207]. We use the ACA framework 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 [99, 94, 95], 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 [195]. 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 Sec.8.3 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 [46], 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. Sec.8.2 presents the foundations of the CQA framework.

Sec.8.3 presents an evaluation that explores the algorithmic tradeoffs between CQA and existing approaches and demonstrates that CQA extends the state-of-the-art.

8.1 Background

8.1.1 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 10 (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:* $Pr : 2^{\Omega} \rightarrow [0, 1]$ *that satisfies the Kolmogorov's probability axioms [173]:*

- $Pr(e) \geq 0$ *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 11 (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 12 (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.

8.1.2 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 [18]; in other cases the accuracy of estimates are probabilistically bounded [39, 33].

8.2 Conditional Quantitative Analysis

The problem this chapter 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 [99]. Another approach is to fuzz the input space to get a statistical bound on the probability of reaching a ψ-state [111, 103, 36]. 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 chapter 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 5 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

Algorithm 5 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:         end if
11:     return [l, u]
12: end procedure
```

Specifications for CQA Functions

generate_intervals(P, ψ)
Input: Program P, reachability property ψ
Output: Set of disjoint logical intervals (see Defs. 3–5)

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 $Pr(\overline{I})$]

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 $\neg\alpha \wedge \gamma$.

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, a logical interval from the set $\mathcal{I}$, 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 3 (Termination)**.** *Algorithm 5 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 5 will terminate. All functions called within CQA are required to terminate due to both time and space bounds, guaranteeing that CQA terminates. □

Theorem 4 (Correctness)**.** *Algorithm 5 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 5. These instantiations are used in the evaluation of CQA, discussed in Sec.8.3.

8.2.1 Instantiation of *generate_intervals*

One non-trivial instantiation of *generate_intervals* is given by the *alternating conditional analysis* framework. 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.

8.2.2 Instantiation of *estimate*

Inputs can be assumed distributed uniformly over their domains or according to a given input distribution called a *usage profile* [94]. For simplicity, a uniform distribution over the input domains will be assumed throughout the chapter; extension to arbitrary usage profiles is orthogonal to our contributions and can be straightforwardly implemented as in [94] and [38].

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 [202]. 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 [18] (with off-the-shelf implementations including Latte [15] and Barvinok [205]). Nonlinear constraints over numerical variables can rely on progress in convex analysis [45], interval constraint paving [108, 169], and the approximate methods developed in both program analysis [38, 89, 189] and statistical machine learning [184]. Model counting over string domains includes exact counters for regular languages [13], exact bound computation [152], and mixed string/numerical counters [200].

More general—though usually more expensive—♯SAT and ♯SMT solvers also exist for model counting over mixed theories (e.g., [52, 106, 55]). 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., [120, 53]) 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 chapter we will focus on linear integer constraints.

8.2.3 Instantiations of *quantify_in_bounds*

Following the principle of conditional program analysis, because all behaviors outside the program interval $\mathcal{I}$ have already been analyzed by *generate_intervals*,[4] quantification techniques can focus only on the residual behaviors, i.e., program paths satisfying the *assumption* $\alpha \equiv \neg\underline{\mathcal{I}}_\psi \wedge \overline{\mathcal{I}}_\psi$.

We require each logical interval I composing $\mathcal{I}$ to be disjoint, and due to this, each I can be reasoned about separately. 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 11 (conditional probability), for each disjoint interval I_i and its corresponding assumption α_i we obtain:

$$Pr(I_\psi \,|\, \alpha_i) = \frac{Pr(I_\psi \wedge \alpha_i)}{Pr(\alpha_i)} \ . \tag{8.1}$$

[4] Recall that *generate_intervals* characterizes behaviors that *must* reach a ψ-state, i.e., $\underline{\mathcal{I}}_\psi$; and upon termination guarantees that all inputs outside the upper bound, i.e., $\neg\overline{\mathcal{I}}_\psi$, *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*.

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 12.

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 $\text{CQA}_{\#}$, CQA_{pse}, and CQA_{sse}, respectively.

8.2.4 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. We discuss two techniques that offer this kind of improved precision in Sec. 8.2.5 and Sec. 8.2.6.

8.2.5 Probabilistic Symbolic Execution

Probabilistic symbolic execution (PSE) extends symbolic execution by computing the probability of each execution path being triggered by a program input [99]. 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 ψ-state (denoted with a ψ superscript), (b) missing a ψ-state (denoted with a $\neg\psi$ superscript), or (c) truncated (denoted with a ? superscript) because the execution along a path failed to reach a ψ-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 ψ-state is given by

$$Pr^{\psi}(P) = \sum_i Pr(PC_i^{\psi}). \tag{8.2}$$

The probability of missing a ψ-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 $\underline{\mathcal{I}}$'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 $\overline{\mathcal{I}}$'s probability mass. Thus PSE satisfies the postcondition of *quantify_in_bounds*.

8.2.6 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) [95] 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*.

8.3 CQA 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?*

8.3.1 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 ALPACA, described in Chap. 5, 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 [205] to count computed intervals. ALPACA uses the CIVL symbolic

executor for C programs [192]; 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 [138], CPA-BAM-BnB [11], CPA-Seq [78], ESBMC-incr [163], PeSCo [181], Symbiotic [194], UltimateAutomizer [118], UltimateTaipan [109], and VeriAbs [79].

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 [95], 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 [205] 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 [80] and then sliced into independent subproblems that do not share symbolic variables (following the slicing rules in [94]). The simplification step helps to mitigate the impact of Barvinok's internal transformation into Disjunctive Normal Form for inputs containing nested disjunctions.

8.3.2 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 [126].[5] 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 [195] 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. We begin with the filtering process as described in Chapter 6.

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.[6]

For subjects passing the initial filter 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., [139, 140, 46], 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.

[5] More stringent probabilities are required for *high demand* contexts.

[6] Arbitrary discrete distributions can be reduced to mixtures of uniform ones over a partition of the finite domain, as in [94] 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.

8.3.3 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.[7] 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 8.1 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.** $\underline{\mathcal{I}}$ and **Most acc.** $\overline{\mathcal{I}}$ 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 $\overline{\mathcal{I}}$ or $\underline{\mathcal{I}}$), then both are counted as having produced the most-accurate bound; thus the columns reporting on accuracy in the table

[7] Available at `bitbucket.org/mgerrard/cqa`.

	Most	Most	Avg.	Run characteristics		
	acc. $\underline{\mathcal{I}}$	acc. $\overline{\mathcal{I}}$	Time (s)	Complete	T/O	Bounded
$\text{CQA}_{\#}$	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 8.1: Summary of evaluation by technique

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. 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, $\text{CQA}_{\#}$ 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 $\overline{\mathcal{I}}$. 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 $\text{CQA}_{\#}$ 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 8.2 and 8.3 use linear diagrams [185] 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. 8.2'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. 8.2 does not necessarily correspond to the same subject as the ith stripe in Fig. 8.3.

We highlight in gold those subjects on which some exhaustive technique (i.e., symbolic-execution-based) could complete. No technique always produces the most-accurate bound, so we give the average distance to that best bound as colored annotations laid over the linear diagram, e.g., whenever $\text{CQA}_{\#}$ does not produce the most-accurate lower bound, it is on average 1.7×10^{-11} away from that best bound.

The main takeaway from Figs. 8.2 and 8.3 is that, in the context of this evaluation, conditioned and unconditioned quantitative analyses can be seen as having complementary strengths in bounding the probability of reaching a ψ-state. This is especially apparent from Fig. 8.3, where the set of subjects for which CQA variants compute the most accurate upper bound are nearly disjoint from those on which PSE/SSE perform the best.

The differences in magnitude between the average distance to the most-accurate bounds between Figs. 8.2 and 8.3 are due to the fact that this study is done in the context of finding ψ-states that have a low probability of occurring. This means that even if a technique like PSE does not find any probability mass reaching a ψ-state, if the most-accurate lower bound is 10^{-27}, then PSE's probability mass of 0 is still relatively close to the most-accurate lower bound. In contrast, if some technique times out or hits a depth bound, the vast majority of the probability

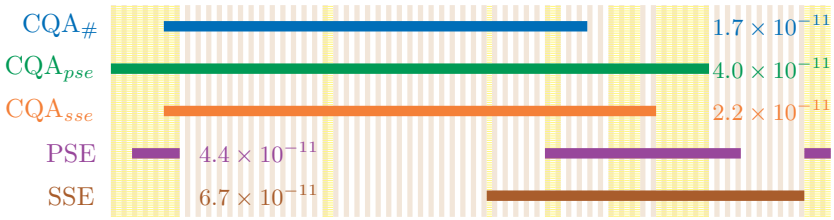


Figure 8.2: Linear diagram of overlap for most-accurate $\underline{\mathcal{I}}$.

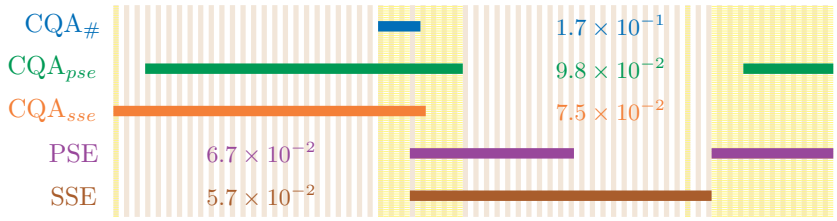


Figure 8.3: Linear diagram of overlap for most-accurate $\overline{\mathcal{I}}$. Sets of the most-accurate lower (Fig. 8.2) and upper (Fig. 8.3) bound for each technique are depicted as horizontal lines, and their intersection by overlapping vertical segments. Each subject is given by a vertical stripe; gold stripes are subjects on which an exhaustive technique completes. The numbers give a technique's average distance to the best bound.

mass involves paths that do *not* reach a ψ-state may remain unaccounted for—e.g., in the worst case for CQA techniques, all time is spent within *generate_intervals* and the upper bound is not lowered at all, as with `email_spec3_product29`. So if one technique does not complete for a subject on which another technique is particularly effective in collecting mass reaching $\neg\psi$-states, the distance to the most-accurate upper bound can be many magnitudes more than to that of the most-accurate lower bound.

The disjointness of each interval returned by *generate_intervals* also offers the potential for parallelism through 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. 8.2.4)—we observed a 2.9× speedup within CQA$_{pse}$ by quantifying each interval in parallel compared to doing so all at once.[8]

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

[8]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.

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 [99, 95], 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., $\underline{I} \equiv \overline{I}$, 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., [39].

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 $\text{CQA}_{\#}$, 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., $\underline{I}_\psi \not\equiv \overline{I}_\psi$, 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 $\neg\underline{\mathcal{I}}_\psi \wedge \overline{\mathcal{I}}_\psi$, as discussed in 8.2.4. Within this focused subspace of the subject, any probability mass proven to lead to a ψ-state effectively raises $\underline{\mathcal{I}}_\psi$ (the lower bound), and mass proven to miss a ψ-state reduces $\overline{\mathcal{I}}_\psi$ (the upper). We will refer to the reduction of uncertainty as *tightening* an interval.

Figure 8.4: Signatures of conditioned PSE raising/reducing the lower/upper bound across different subjects.

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 8.4 depicts how $\mathcal{I}_\psi$ is tightened from below and from above across a sample of four subjects[9] whose bounds 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 8.4, 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

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

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: $(1 - \#\overline{\mathcal{I}}) - \#\underline{\mathcal{I}}$)—how long such an interval will take to explore.

8.3.4 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 nonviolating 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}.

8.3.5 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) [85]. Our model counting interface delegates counting of linear integer constraints to Barvinok [205], after basic simplifications of the constraints via Z3 [80]; 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 [10] 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., [21, 206, 171, 88, 183]. 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.

8.3.6 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 $\overline{\mathcal{I}}$.)

Figure 8.5 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

[10] 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.

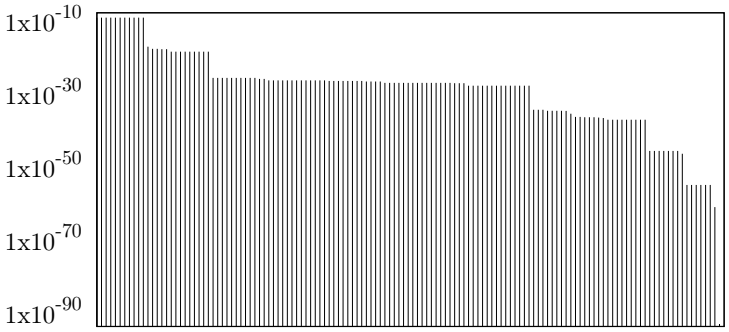


Figure 8.5: Least upper bounds per subject across techniques.

safe. Fig. 8.5 demonstrates that the techniques in this study are able 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.

8.4 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 ψ-state, and use this information to either bound the probability of reaching a ψ-state, 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. 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 [60, 121, 112, 20, 104, 8]; 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 [167], 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 [24]. 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 [24, 31]; combining verification and systematic testing [76, 58]; transforming one program to another containing fewer execution paths while retaining the possible property violations of the original program [93, 27]. 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 [131] or on the definition of probabilistic abstract interpretation methods [137, 162, 73]. The former can take advantage of efficient probabilistic model checkers [131], 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 [99] is a recent technique exploiting symbolic execution to extract conditions on the input leading to ψ-state. We presented techniques in this family in Section 8.2. Variations of PSE and SSE have also been used for exact/approximate reliability analysis [94, 151], performance analysis [54], and detection and automated exploitation of side-channel vulnerabilities in both regular and probabilistic programs [44, 153]. In this chapter, we focused on non-probabilistic programs; investigating possible extensions of CQA to general probabilistic programs [107] and approximate computing [188, 40] is left as future work.

Testing [36, 103] 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 [111]. As already discussed, statistical techniques can be coupled with uniformly random testing to obtain statistical bounds on error probability (e.g., [4, 144, 216]), 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 [35]. 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 [158].

Chapter 9

Algorithmic Diversity in ACA

A static program analyzer reasons about a program's behavior without actually running the program. To do so, an analyzer will first construct a model of the program, and an analysis is then performed on the model. The choice of model and the manner of its analysis can differ widely from one program analyzer to another. This results in an array of program analyzers that have complementary strengths and weaknesses, e.g., one may approximate some loop's behavior, reducing time at the cost of precision; while another may examine each iteration of the loop, gaining precision at the cost of using more resources.

Typically, the performance of one analyzer is pitted against another's—to show, for example, that one technique is superior in a given domain. This is the general style of academic research, crystallized in the motivating statement for the annual Competition on Software Verification (in which static analysis tools are compared against each other across a set of benchmarks): "Competition is a driving force for the invention of new methods, technologies, and tools." [198]

Our work attempts to recast this competitive spirit into a cooperative one—if multiple program analyzers work together, each may emphasize their complementary strengths while mitigating the weaknesses. In an ideal setting, this cooperation would proceed without a hitch: a program's recursive function calls would be handled by analyzers tailored to do so, while its straightline code would be handled by another, etc. But if the program analyzers are treated as black boxes, there can be a number of bumps along the road.

The main challenge lies in getting program analyzers to effectively communicate their (possibly partial) analysis results to each other. Because each analyzer models a given program in a different way, communicating the facts derived from distinct models is not straightforward. We will use the ACA framework to explore some of the challenges and opportunities involved in treating program analyzers as communicating black boxes. Do we observe effective communication from a diversity of program analyzers, or are just a few tools used? Are pairs of analysis tools correlated in effectiveness across a given program and its conditioned variants? We use the metric of "effective evidence"—the results of effective conditioning (see Def. 6), described below—as a way to observe how these black box analyzers communicate with each other in ACA.

9.1 Context of diversity study

In this chapter, we explore algorithmic diversity in the context of the subproblems generated by ACA. We conjecture that is is advantageous to use an algorithmically diverse set of tools while performing an alternating conditional analysis. To compute a nontrivial program interval as is done in ACA, a single cleverly-designed analysis tool is not enough. Given a multiplicity of difficult verification tasks, no single tool will always outperform the others. A portfolio of diverse analysis tools is needed.

We first look at the nature of the subproblems that are generated by ACA, and then discuss the space of analyzers with which we instantiate the ACA tool portfolio.

9.1.1 Subproblems generated by ACA

The subjects we consider in this chapter come from the SV-COMP set of benchmarks described in Chapter 6. Each round of analysis in ACA yields a new verification problem for each analyzer, so n iterations of ACA corresponds to showing the analyzers in the portfolio at least n new problems (possibly more when generalization is employed). Though each new verification problem can be thought of as a decomposition of the original problem—i.e., the program minus the input space described by $\overline{\mathcal{I}}$—, because we treat the analyzers as black boxes, the decomposition must be passed as an entirely new program. The difference between the original program and a conditioned one with an *assume* statement may appear small, but this *assume* could be cutting

out a large semantic space of the program. For instance, if reaching a ψ-state depends on the value of integer variable x, then a conditioned program with the statement $assume(x \equiv 1)$ yields a much simpler verification problem.

9.1.2 Space of analyzers in portfolio

This study instantiates the ACA framework with a tool portfolio of nine different static analysis tools:

- CBMC [138]
- CPA-Seq [78]
- ESBMC [163]
- PeSCo [181]
- SeaHorn [113]
- Symbiotic [194]
- UAutomizer [118]
- UTaipan [109]
- VeriAbs [79]

Each tool takes as input a C program and a specification in linear temporal logic, and reports if the specification is violated or not. We provide brief summaries of these tools below, and specify whether they are an overapproximator ($\overline{A}$), an underapproximator ($\underline{A}$), or both.

CBMC stands for "C Bounded Model Checker."

CBMC first converts the C program into a GOTO program (where control flow statements are translated to guarded goto statements). As a standard application of bounded model checking, CBMC explores the program model up to a configurable depth of loop unrollings, and all of the collected paths are represented by a large disjunctive logical formula. The formula can be passed to a backend SAT or SMT solver. If the solver returns a satisfying model, this represents a reachable path to a ψ-state, and can be translated back to a series of program statements that lead to the ψ-state.

As CBMC only unrolls loops a bounded number of times and could miss reasoning about program executions beyond this bound, CBMC returns only sound reachability proofs, and so is typed as an $\underline{A}$.

CPA-Seq is a sequential configuration—the competition contribution for SV-COMP—of CPAchecker, which stands for "Configurable Program Analysis Checker."

CPAchecker is based on the flexible analysis framework of a configurable program analysis (CPA), described in [25]. The CPA framework allows a hybrid of model checking and abstract interpretation to be created by providing one or more abstract interpreters and defining a termination check and merge operators. Where does the hybrid-ness come in? When exploring a graph representation of a program in classical abstract interpretation, information is merged at nodes that refer to the same program location; whereas in model checking nodes are never merged. CPA allows you to specify the degree to which merging should occur. The abstract domains may also be composed through a product lattice.

CPA-Seq is typed as both an $\underline{A}$ and an $\overline{A}$.

ESBMC stands for an "Efficient SMT-based Bounded Model Checker."

ESBMC extends bounded model checking with k-induction. ESBMC integrates the formulae produced by CBMC with a backend employing multiple SMT-solvers. Whereas CBMC is typically run with one SAT solver as its backend, ESBMC takes advantage of richer theories than propositional logic, such as theories of non-linear arithmetic, bit-vectors, arrays, etc.

To prove correctness in the presence of loops, ESBMC uses k-induction [97]. This is a method to demonstrate that if no ψ-state is reached after k loop unwindings, then it will not be reached in subsequent unwindings.

ESBMC is typed as both an $\underline{A}$ and an $\overline{A}$.

PeSCo stands for "Predicting Sequential Combination of Verifiers."

PeSCo tries to predict the best sequential combination of verifiers on a given subject, and then launches this combination on the task. The underlying verifiers are six different configurations of CPAchecker. The prediction is based on machine learning using support vector machines. PeSCo learns rankings of verifiers based on the SV-COMP scoring scheme.

PeSCo extracts features of a program by encoding the program as a control flow automaton extended by data and control dependencies. PeSCo uses CPA's *restart* algorithm: if one verifier fails, it can still pass on its already computed (partial) results to the next verifier (as in [24]). The training phase is done on the SV-COMP verification tasks. The details of PeSCo's feature extraction and ranking procedure are given in [75]. If there is no prediction in a reasonable amount of time, PeSCo defaults back to the CPA-Seq fixed configuration.

PeSCo is typed as both an $\underline{A}$ and an $\overline{A}$.

SeaHorn's name comes from the fact that its verification conditions are encoded as Constrained Horn Clauses; the "Sea" may stand for the "C" in Constrained or the "C" language.

SeaHorn is made up of a front-, middle-, and back-end. The front-end translates a C program (or other language for which there is an LLVM front-end) into LLVM bitcode. The middle-end emits verification conditions encoded as Constrained Horn Clauses. A Horn clause is a way to say "assume that, if p and q and ... and t all hold, then u also holds"; this encoding is natural for describing program semantics, and can be handed off to many automated solvers.

The back-end employs several SMT-based model checking engines based on PDR/IC3, along with abstract interpretation (via IKOS [43]) to supply program invariants. The inferred invariants can help the model checker to prove safety. SeaHorn can be configured to reason about small-step semantics or a large-block encoding, and its abstract domains can also be configured.

SeaHorn is typed as an $\overline{A}$.

Symbiotic is named for its synergistic combination of instrumentation, slicing and symbolic execution [193].

Symbiotic is primarily a symbolic execution engine with a few preprocessing steps to reduce the size of the program-to-be-verified. A preprocessing step of program slicing can remove large portions of the program, including loops, that are not related to the reachability of a ψ-state. Symbiotic uses KLEE [48] as its underlying symbolic execution engine.

Symbiotic is typed as an $\underline{A}$.

The next two tools are built upon the "Ultimate" architecture [3], which has plugins for program parsing, transforming, and analyzing.

Ultimate Automizer is named for its automata-based approach [119].

This approach represents the program as an automata built from sets of hoare triples, i.e., the program statements are the nodes, and the edges are annotated with pre- and postconditions of given nodes. Sets of traces are represented by these automata.

A CEGAR-based exploration is performed on this program model. The difference with classical CEGAR is that when an infeasible trace is found in the automata representation, the

infeasible trace is excluded from the model by simply subtracting this trace from the automata, instead of performing costly SMT calls to partition the predicate domain such that the infeasible trace will be excluded. The refinement step employs different techniques to obtain interpolants from the infeasible trace, such as Craig interpolation and constraint-based invariant synthesis.

Ultimate Automizer is typed as both an $\underline{A}$ and an $\overline{A}$.

Ultimate Taipan has a surname whose origin is unknown.

Ultimate Taipan follows its preceding cousin as a CEGAR-based model checker. Taipan refines its model by performing an abstract interpretation on a fragment of the program they call a path program, which is a projection of the control flow graph onto a selected sequence of program statements, described in [110].

Ultimate Taipan is typed as both an $\underline{A}$ and an $\overline{A}$.

VeriAbs stands for "verification by abstraction and test generation."

VeriAbs contains a portfolio of different analysis techniques, and decides which to employ based on a lightweight analysis of the input program. VeriAbs uses fuzz testing to find inputs that lead to a ψ-state, and uses a combination of bounded model checking and k-induction (as with ESBMC) to find proofs of the unreachability of ψ-states.

VeriAbs is typed as both an $\underline{A}$ and an $\overline{A}$.

9.2 Experimental Setup

In this chapter, we explore two questions: (1) how does employing multiple diverse analyzers advantage ACA, and (2) is there a correlation between pairs of analyzers and effective evidence in a run of ACA. Before looking to these questions, we describe the experimental setup.

Metric of effective evidence

The metric we will use to get at these questions is that of effective evidence. Effective evidence is the result of effective conditioning, given in Def. 6. Effective evidence means either that the inputs to a ψ-state that are characterized are guaranteed to be executable and have not been seen on a previous ACA round of analysis, or that evidence to the ψ-state's unreachability (given a

conditioned program) are given by an overapproximate analysis, whose proof is therefore assumed to be valid. For example, if the analyzer Symbiotic (which is typed as an $\underline{A}$) reports a new path to a ψ-state, this is effective evidence; if ESBMC (which is typed as both an $\overline{A}$ and an $\underline{A}$) gives a proof of the unreachability of a ψ-state—thus terminating ACA, this is also effective evidence.

Pieces of evidence are counted on each round of analysis in ACA, and multiple analyzers can provide effective evidence on each of these rounds. Assuming no generalization, all tools providing either effective reachability or unreachability evidence are counted as providing one piece of evidence. During generalization, we count each analyzer as contributing a maximum of one piece of evidence per lattice level, to mitigate overcounting when the lattice is wide (see 5.3). At most one piece of evidence is ultimately considered at each lattice level, even if one tool finds multiple pieces of evidence among lattice elements on the same level.

Running ALPACA

The instantiation of ACA used in this study is ALPACA, described in Chapter 5. The experiments in this study were run on the `exp` branch of ALPACA from revision:

`c835094c52c2019232d379bd97e5c0928ba5d72b`.

Each `subject.c` was run using the command:

```
timeout --signal=SIGKILL 90m $ALPACA --docker -t 300
 --generalize-timeout 300 -p allDock --known-reach
 --exit-strategy {eager|patient}
 --gen-exit-strategy {eager|patient} subject.c
```

This command is the same as in Chapter 6, with the addition of the strategy flags, explained below.

The `exit-strategy` flag determines whether, on each iteration of ALPACA, the portfolio of tools stops searching after the first valid reachability (or unreachability) condition is confirmed—the *eager* strategy, or continues until all analyzers either complete or exhaust their timebound—the *patient* strategy. We use both the *eager* and *patient* strategy in this study because we want to examine the relation amongst analyzers both when ACA attempts to reduce runtime (via

the *eager* strategy) and when ACA allows analyzers to collect as much evidence as possible (via *patient*), up to a time bound.

The `gen-exit-strategy` is analogous to the above flag, but is applied within the generalization phase of ACA: the termination of any one of the many possible generalizations—each of which is a distinct lattice element as defined in Chap. 5, Sec. 5.3—of an interval can occur either eagerly or patiently. The `generalize-timeout` is set to 300 seconds in order to be significantly less than the ALPACA default time of 900 seconds to run the portfolio on any given iteration. The generalization time is reduced because the lattice of conjuncts may be large, and all of its elements time-intensive to explore.

Either both exit strategies are set to *eager* or both are set to *patient*.

Subject selection

The subjects selected are those coming from the SV-COMP corpus of benchmarks discussed in Chapter 6, using the same selection criteria. The hardware and setup is also the same as in Chapter 6.

9.3 Evaluation of Algorithmic Diversity

In this section, we will explore how employing multiple diverse analyzers advantages ACA. We discuss how how different subsets of analyzers contribute to characterizing reachable and unreachable ψ-state spaces, and see if pairs of analyzers are correlated across a run of ACA. To this end, we look at two research questions.

RQ1 *How does a diversity of analysis tools contribute to finding effective evidence within ACA?*

RQ2 *Is there a correlation among analyzer effectiveness on subproblems of the same subject?*

9.3.1 RQ1—contribution of diversity

We approach this question by seeing first if more than one tool is needed to compute $\mathcal{I}$, and if so what are the relative contributions of tools in terms of effective evidence. We answer this on two different ACA configurations: when rounds of analysis are short-circuited upon discovery of some

TOOL	$\#(E_r)$	MAX	MIN	AVG	MED
VeriAbs	1394	305	13	119	79
CPA-Seq	773	270	5	34	32
UAutomizer	209	302	10	117	102
Pesco	162	155	8	25	15
Symbiotic	101	519	5	21	8
CBMC	13	11	3	4	4
UTaipan	12	336	11	93	34

Table 9.1: Reachability evidence count and times (s) run with `eager` strategy

TOOL	$\#(E_u)$	MAX	MIN	AVG	MED
Seahorn	336	293	2	27	5
UAutomizer	116	268	28	83	70
VeriAbs	69	305	116	237	250
ESBMC	62	34	3	6	5
CPA-Seq	14	75	8	28	26

Table 9.2: Unreachability evidence count and times (s) run with `eager` strategy

effective evidence; and when each round allows all tools some fixed time to explore. We first look at ACA when run with the short-circuiting—`eager`—strategy.

The task of ACA would be simplified if a single analysis tool could provide the required reachability and unreachability evidence to construct a program interval. But among the study, no single tool could comprehensively characterize the state space of all programs when running with the `eager` strategy.

Across 380 C benchmarks, each of the 9 tools in the portfolio provides effective evidence; some provide reachability paths, others unreachability proofs, and a few analyzers provide both kinds of evidence. On average, four distinct tools per program provide effective evidence; with the maximum program collecting evidence from five distinct tools.

We present the data of evidence provided by analyzers in two separate tables: one for those that are able to provide effective reachability evidence—shown in Table 9.1; and another for analyzers that are able to provide safe unreachability evidence—shown in Table 9.2. The tables are sorted by counts of effective evidence. The set of analyzers among the two tables is not disjoint, e.g., VeriAbs is able to provide both reachability and safe unreachability evidence. But some tools can only underapproximate a program's state space, such as CBMC, and some can only provide effective unreachability evidence, such as SeaHorn.

The $\#(E_r)$ and $\#(E_u)$ columns in Tables 9.1 and 9.2 denote the number of pieces of effective reachability and unreachability evidence, respectively, provided by each analyzer, and the MAX, MIN, AVG, and MED, refers to the maximum, minimum, average, and median times (in seconds), to find a piece of evidence. Because there are multiple pieces of evidence collected per subject in general, the # column sums to greater than 380.

The main takeaway from Tables 9.1 and 9.2 is that *all* analysis tools are used within ACA to construct a more precise program interval across subjects, even with short-circuiting when the first tool finds effective evidence on analysis rounds. Each tool contributed at least 12 pieces of combined reachability and unreachability evidence across 380 subjects. Note that a trivial interval can be built without any analyzers—this interval would imply the whole program in possibly reaching a ψ-state and would be wholly uninformative. But if many analyzers can be employed, then each piece of evidence provided only strengthens the precision of an interval. On many subjects, a single iteration would yield multiple distinct pieces of evidence from different analyzers, which significantly accelerated convergence within ACA.

All tools in the portfolio contributed to building a more precise $\mathcal{I}$, but they did not all contribute equally in terms of pieces of evidence. For reachability, the leading two tools—VeriAbs and CPA-Seq—account for the bulk of effective evidence found (81%). Similarly, for unreachability proofs, the leading two tools—SeaHorn and Ultimate Automizer—find the majority of evidence (76%). The collection of the four leading tools is noteworthy in that each uses employs very different strategies to find effective evidence—combinations of fuzzing, k-induction, abstract interpretation with composite domains, solving constrained horn clauses, running a CEGAR-style model check. In any case, no one tool can be singled out as the most effective.

What is happening within ACA when a large number of distinct tools find effective evidence? We look at the subject with the largest number of contributing tools for the `eager` strategy: `minepump_spec3_product59`. On the first round of analysis in ACA, CPA-Seq quickly (in 13 seconds) finds reachability to a ψ-state; on the second round, both PeSCo and Symbiotic find two new distinct pieces of reachability within seconds of each other; no effective evidence could be found on the third round, so generalization allows VeriAbs to find a new reachability condition; Symbiotic, PeSCo, and VeriAbs continue to collect new evidence for ψ-state reachability, and SeaHorn declares unreachability for the final subproblem.

It is interesting to observe how the first three tools found three different reachability paths in a very short time (less than 25 seconds), after which no tool was effective until generalization steered tools away from some portion of the state space that was difficult to analyze. For this subject, the creation of $\mathcal{I}$ necessitates a diverse toolset, but tool diversity is not enough to build $\mathcal{I}$, and an effective generalization mechanism must also be able to condition analyzers to explore

TOOL	$\#(E_r)$	MAX	MIN	AVG	MED
VeriAbs	1296	415	13	87	62
CPA-Seq	735	334	5	34	27
UAutomizer	399	592	9	118	96
Symbiotic	389	784	4	28	8
Pesco	322	339	8	26	14
UTaipan	93	643	9	105	53
CBMC	68	820	3	57	6

Table 9.3: Reachability evidence count and times (s) run with `patient` strategy

TOOL	$\#(E_u)$	MAX	MIN	AVG	MED
Seahorn	294	235	2	32	6
UAutomizer	124	304	24	84	65
ESBMC	75	32	3	5	5
VeriAbs	54	305	139	227	238
CPA-Seq	15	70	7	23	16

Table 9.4: Unreachability evidence count and times (s) run with `patient` strategy

reduced subspaces.

We now look at ACA when run with the strategy to allow all tools some time bound to find effective evidence—the `patient` strategy. Data for the reachability and unreachability counts of each analysis tool using the `patient` strategy is given in Tables 9.3 and 9.4, respectively.

As in the `eager` strategy case, all analysis tools are used to construct a more precise $\mathcal{I}$. Unsurprisingly, when each tool is given more time to explore a subproblem, there are more pieces of effective reachability evidence that can be collected. This makes the distribution of effective reachability evidence among tools in the portfolio more even than in the `eager` case: the same two leading tools only account for 62% of evidence, rather than 81%.

There is actually a decrease in the total number of pieces of unreachability evidence found from the `eager` strategy to the `patient` strategy, by 35. This is because allowing each round of analysis to run until some timeout causes the overall runtime of ACA to grow and thus hit a timeout faster, and there are no unreachability reports in the presence of a timeout. The distribution of contributions among tools is still mostly from the leaders SeaHorn and Ultimate Automizer (74%).

What is happening within ACA in the `patient` case when a large number of distinct tools find effective evidence? We look at the subject with the largest number of contributing tools (8) for the `patient` strategy: `array`. In this instance tool diversity does not necessarily help in the construction of $\mathcal{I}$: six tools all find the same piece of reachability evidence on the first round of analysis, and SeaHorn finds unreachability evidence on the second. A subject with the next largest number of contributing tools (6)—`floppy_simpl4`—does not find any redundant evidence: on the first round of analysis, five different tools find five distinct pieces of evidence;

and after generalization in the second round, SeaHorn finds unreachability evidence.

We conjecture that the diversity in both the `eager` and `patient` cases is helpful because varying tools have strengths in different areas. For instance, VeriAbs was fairly successful in collecting unreachability evidence (69 and 54 pieces running `eager` and `patient`, respectively), while it was one of the highest-performing tools as far as collecting reachability evidence (1394 and 1296 pieces)—this can in part be explained because it employs a random fuzzer to search for reachable paths, and the fuzzing was able to quickly reach some ψ-state across these subjects.

> **RQ1 Findings.** Diversity is needed to produce a non-trivial $\mathcal{I}$ on a variety of SV-COMP benchmarks—no tool or pair of tools can do so on its own. The relative contributions of each tool in the portfolio is heavily dependent on the strategy ACA employs to wait for multiple pieces of evidence to come in or not, though four tools tended to collect the majority of effective evidence regardless of strategy.

9.3.2 RQ2—correlation among analyzer pairs

This section explores whether or not certain analyzer pairs are correlated in finding effective evidence over a given analysis run. What would correlation between tool pairs tell us in the context of ACA? Correlation could be the result of four (not necessarily distinct) causes: two tools can share evidence effectively within ACA (or not), and two tools are able to effectively verify similar subproblems (or not).

For example, if many subjects have reachable ψ-states that occur after unbounded loops, two CEGAR-based model checkers may be able to handle this class of programs and would have resulting high correlations, whereas a bounded model checker may deal poorly with these programs, and be negatively correlated with the first two. The other possibility of correlation could occur if, altering the preceding example, evidence found by the CEGAR-based model checker can be used to condition the bounded model checker on a subsequent round of analysis—allowing it to bypass the "blockage points" of unbounded loops and find new ψ-state reachability—, then these two tools could be correlated. Because we do not enter the black boxes of analysis tools, we measure correlation from high level ACA statistics.

CB = CBMC	SH = SeaHorn	CS = CPA-Seq
UA = UltimateAutomizer	ES = ESBMC	VA = VeriAbs
UT = UltimateTaipan	SY = Symbiotic	PE = PeSCo

Table 9.5: Two-character abbreviations for analyzers.

We will measure correlation in the following way. Given a subject and some blocking clause, an analyzer either finds effective evidence or not, so we use the ϕ-coefficient to examine correlation, which is a measure of association over binary variables [214]. Two analyzers are considered correlated if they both find (or cannot find) evidence for the same subject and same blocking clause; they are considered negatively correlated if one finds evidence but the other does not. The ϕ-coefficient is derived from a 2×2 contingency table for binary variables X and Y.

We consider correlation only among analysis runs that yield reachability evidence, because due to a short-circuit termination condition in the implementation of ALPACA, a tool providing unreachability evidence will be the only tool appearing for that run. As a result of this implementation detail, tools providing mostly unreachability evidence have weak negative correlations with other tools, which would likely go away if the short-circuiting were removed. SeaHorn provides *only* unreachability evidence, thus it does not appear in this subsection's tables.

According to [177], a rough estimate for interpreting the strength of correlation (positive or negative) given by the ϕ-coefficient can be broken down into five categories: very strong (0.8+), strong (0.6–0.79), relatively strong (0.4–0.59), moderate (0.2–0.39), and weak to negligible (0–0.19). These qualifiers are given under two assumptions. The first is that the sample size is sufficient, which holds with 3261 and 3864 samples, for `eager` and `patient` strategies, respectively. There are more samples than the 380 base programs, because we consider correlation within each program and an associated blocking clause, which constitutes an independent analysis run; ACA will in general have multiple analysis runs for a given program (see Chapter 3). The second assumption is that each cell of the underlying 2×2 contingency table contains an expected frequency of at least five; this assumption is satisfied.

The tables in the remainder of this chapter reference many individual analysis tools together, so their full names are associated with the two-character abbreviations given in Table 9.5.

Table 9.6 gives an upper triangular matrix for the ϕ-coefficient among each analyzer pair, when run with the `eager` strategy. We use an upper triangular matrix because correlation is

symmetric, and we do not include a tool's relation with itself (which is always 1).

	CB	CS	ES	PE	SY	UA	UT	VA
CB		-0.05	0.02	-0.09	-0.08	0.15	0.3	-0.12
CS			-0.08	-0.41	-0.09	0.04	-0.1	0.16
ES				0.03	-0.16	-0.16	0.11	-0.08
PE					-0.19	-0.23	-0.1	0.03
SY						-0.21	-0.08	-0.06
UA							0.17	-0.29
UT								-0.13
VA								

Table 9.6: Upper triangular matrix (correlation is symmetric) of ϕ-coefficient among individual analyzers when run with the `eager` strategy.

	CB	CS	ES	PE	SY	UA	UT	VA
CB		0.13	0.08	-0.04	-0.01	0.18	0.14	-0.11
CS			0.08	0.31	0.33	-0.06	-0.03	0.39
ES				0.09	-0.14	-0.15	0.01	0.14
PE					0.2	-0.48	0.02	0.65
SY						-0.29	-0.27	0.33
UA							0.62	-0.49
UT								-0.05
VA								

Table 9.7: Upper triangular matrix (correlation is symmetric) of ϕ-coefficient among individual analyzers when run with the `patient` strategy.

The main observation from Table 9.6 is that almost all tool pairs under the `eager` strategy have a weak to negligible correlation. The only relatively strong correlation is a negative one between PeSCo and CPA-Seq (-0.41). A negative correlation could suggest that the tools cannot share evidence effectively. This is unlikely to be the case for PeSCo and CPA-Seq, because they share the same input and output interfaces as instantiations of the CPAchecker toolchain, and even employ common abstract domains.

A negative correlation could also suggest that two tools are complements to one another, that is, that one is only effective in the state spaces in which the other is ineffective, and vice versa. The fact these two tools have a negative correlation is interesting because PeSCo and CPA-Seq are built within the same toolchain and run many of the same algorithms, the main difference is that PeSCo runs them in a permuted order. We conjecture that this negative correlation is a result of the variability in the quality of PeSCo's ordering predictions on a given subject and its corresponding subproblems.

Table 9.7 gives an upper triangular matrix for the ϕ-coefficient among each analyzer pair, when

run with the `patient` strategy. The first observations from Table 9.7 is that more pairs exhibit moderate to strong correlations compared to pairs in Table 9.6. Why is there this difference? We conjecture that the `eager` strategy, while reducing overall runtime for ALPACA, masks possible correlations by its short-circuiting mechanism. For instance, if VeriAbs finds effective evidence after 16 seconds, and CBMC would have taken 18 seconds, in the `eager` strategy, CBMC will not have its evidence counted even though it was relatively well-suited for that verification problem.

The strongest positive correlations in Table 9.6 are between PeSCo and VeriAbs (0.65) and between Ultimate Automizer and Ultimate Taipan (0.62). The strongest negative correlations are between Ultimate Automizer and VeriAbs (-0.49) and Ultimate Automizer and PeSCo (-0.48). Because we cannot see the internal details of the analysis tools, we will look at ACA log files on subjects with strong tools pair correlations to see what is happening within ACA.

We look at the two pairs with the strongest correlations discussed in the previous paragraph. We will select one random subject from the pool of subjects in which both tools provide effective evidence. Our random selection is done by placing the pool of filenames in a text file and running the Unix command `shuf -n 2 filenames.txt` to select two subjects.

The selected subjects from the first pair—(PeSCo,VeriAbs)—are `minepump_spec4_product45` and `transmitter.02`. On the first subject, both tools find reachability evidence on the first round of analysis, after which only VeriAbs can find effective evidence. On the second subject, PeSCo finds effective on the first round of analysis, and VeriAbs finds effective evidence during generalization after the fourth round of analysis in ACA. The selected subjects from the second positive pair—(Ultimate Automizer,Ultimate Taipan)—are `Problem01_label38` and `token_ring.04`. For both of these subjects, we again find the pattern of both tools being effective on the first round, and only Ultimate Automizer is thereafter. Over these four samples, the positive correlation appeared both when tools are simultaneously effective over a given subproblem, and also when the conditioning from one is effective for the other.

We also considered the correlation among families of analyzers. However, seven of the nine tools fell into either the CEGAR or the BMC families, while more classical Symbolic Execution and Abstract Interpretation were only represented by single tools. A more detailed evaluation of how algorithmic families relate to each other—using an ACA instantiated with more representatives from each family—is left to future work.

RQ2 Findings. We did not find strong correlation between effective evidence and tool pairs when each round of analysis in ACA is terminated eagerly. When given a fixed time on each round of analysis in ACA, there are strong correlations (both positive and negative) among tool pairs finding effective evidence. Positive correlation appeared when tools are effective over the same subproblem, and also when the conditioning from one is effectively used by the other.

9.3.3 Discussion

This subsection is a more anecdotal discussion of analyzer diversity observed within ACA. We briefly look at ordering effects among analyzers as ACA proceeds, how long different tools take to find the time to find effective evidence, and why choosing a downsized portfolio is difficult.

Ordering Effects

We first look at the ordering effects among tools within the context of ACA. Do certain tools often appear before others in previous rounds of ACA? Do some tools never appear in later rounds? These relations can suggest how one analyzer can cover a portion of the state space that may be difficult for another analyzer, freeing the latter to explore state spaces that align better with its internal abstract domain.

Tables 9.8 and 9.9 list the tools that produce effective evidence in later iterations of ACA, under the `eager` and `patient` strategies, respectively. A *later iteration* is defined as any iteration of ACA after the first iteration, that is, we consider evidence collected when *analyze* of Algorithm 1 has been executed two or more times. The "Later #" column counts the number of times a tool appears in a later iteration, and the "Precursor counts" column shows the top three tools that appeared most often in previous iterations—each of the tools in this column lists the number of preceding appearances.

The biggest takeaway from Tables 9.8 and 9.9 is that only one tool was its own maximal precursor for each strategy (CPA-Seq and Symbiotic). This means that for the other tools in Tables 9.8 and 9.9, the exploration of the program's state space in later iterations of ACA largely depended on the previous efforts of *other* analysis tools.

Tool	Later #	Precursor counts
VA	143	CS:95, VA:75, PE:61
CS	99	CS:84, PE:45, SY:6
UA	66	CS:46, UA:45, PE:12
ES	62	CS:32, PE:27, VA:18
SY	31	CS:31, SY:21, VA:20
PE	9	CS:8, PE:4, VA:3
UT	6	UA:3, CS:3, CB:2
CB	4	CS:4, CB:2, PE:1

Table 9.8: Display of ordering effects by showing the number of times an analyzer appears in a later iteration along with its top three most frequent precursors when run with the `eager` strategy.

Tool	Later #	Precursor counts
VA	133	SY:133, PE:133, CS:130
CS	82	SY:82, PE:80, CS:79
ES	75	CS:71, PE:66, SY:65
UA	67	UA:67, SY:59, CS:59
SY	28	SY:28, PE:28, CS:28
UT	18	UA:19, SY:19, UT:19
PE	11	SY:11, PE:11, CS:11
CB	4	UT:4, UA:4, PE:4

Table 9.9: Display of ordering effects by showing the number of times an analyzer appears in a later iteration along with its top three most frequent precursors when run with the `patient` strategy.

Analyzers built around a fixed toolchain, such as Ultimate and CPA-based tools, are found preceding and following each other, unsurprisingly. But some tools are found preceding the exploration efforts of *most* tools, such as CPA-Seq, which appears as a top precursor for six tools under the `eager` strategy, and Symbiotic, a top precursor for five tools under the `patient` strategy.

Table 9.8 contains only eight of the nine tools. SeaHorn does not appear because it is only used for unreachability evidence, and so by definition will only appear in a later iteration.

Variation in time to find evidence

The diversity in techniques in relation to different program state spaces is also reflected in the time taken by each analyzer to find effective evidence. Figures 9.1 and 9.2 present the impulse plots of analyzers collecting 54 pieces of evidence or more in either reachability or unreachability; the vertical axis is time in seconds and goes up to 630, each "impulse," or line, indicates the time an analyzer takes to find a piece of evidence, and these impulses are sorted by ascending time.

The associated analyzers are given in the caption to Figures 9.1 and 9.2, but most details have been suppressed in the figure so that the reader focuses only on the varying *shapes* of analysis times across analyzers. Some analyzers such as Symbiotic and SeaHorn have long, skinny tails. This indicates that effective evidence can usually be computed very quickly. This makes sense, as abstract interpretation is fast when the abstract domains used are simple; and symbolic execution is fast when the portion of the state space searched during the analysis does not involve much unbounded looping behavior. Because state space partitioning is performed on-the-fly by ACA,

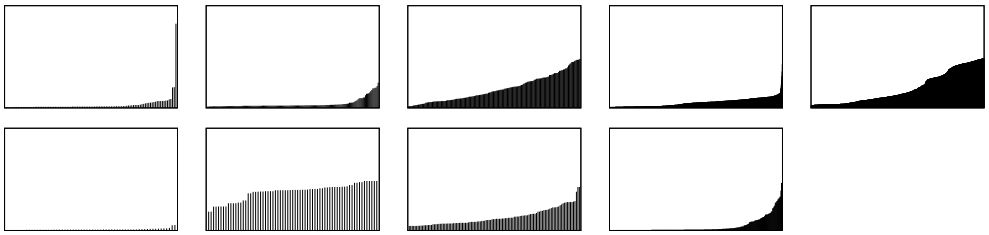


Figure 9.1: Impulse plots of effective evidence sorted by time for analyzers collecting 54 pieces of evidence or more when run with the `eager` strategy. Time in seconds on the vertical axis goes up to 630. Tools providing evidence in top row, from left to right: reachability from Symbiotic, Pesco, UAutomizer, CPA-Seq, VeriAbs; evidence in bottom row: unreachability from ESBMC, VeriAbs, UAutomizer and SeaHorn.

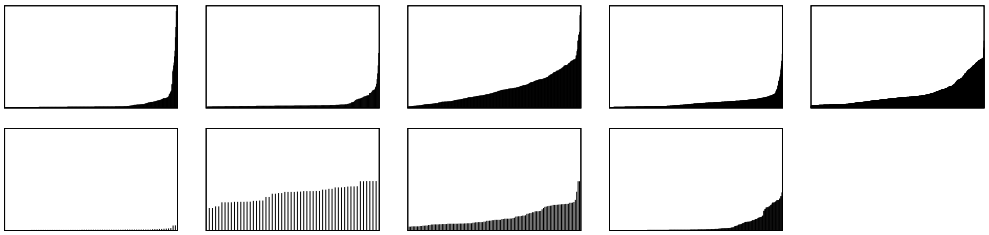


Figure 9.2: Impulse plots of effective evidence sorted by time for analyzers collecting 62 pieces of evidence or more when run with the `patient` strategy. Time in seconds on the vertical axis goes up to 630. Tools providing evidence in top row, from left to right: reachability from Symbiotic, Pesco, UAutomizer, CPA-Seq, VeriAbs; evidence in bottom row: unreachability from ESBMC, VeriAbs, UAutomizer and SeaHorn.

employing a diverse portfolio of analyzers improves the chances of some partition aligning with an analysis tool's sweet spot. Others such as the Ultimate Automizer and VeriAbs tend to take longer on any given subject: their tails are shorter and fatter.

Considerations in Downsizing

If it were possible to predict *a priori* which tool or tools may be best suited for a particular program, then just one or two analyzers could be run at a time, and the portfolio could be significantly downsized. However, it is very difficult to predict which tool will be best in analyzing a program, in general.

Unlike some strategies used in portfolio-based SAT solvers [211], where the structure of formulae allow heuristics to guide which solvers to run in parallel or which configurations to apply to several instances of a solver, the structure of a general program, the reachability property,

and the abstract domain as modeled by an analyzer all must align; whether this may happen or not is hard to predict. These three factors interact in complex ways that can only be indirectly reflected in a black-box study such as this one. It is possible to have misalignment of structure and abstract domains in parts of the program state space unrelated to the property and the analyzer can still successfully characterize parts of its state space. There is a causality dilemma in that the alignment of the factors happens in the modeled state space, so the analyzer must first construct the state space before you can see the alignment—this is essentially running the analyzer to see if you should indeed run the analyzer.

Across the benchmarks in this study, there was no strict subset of tools that could effectively characterize each. So if the computing resources are available, running all static analyzers is recommended, in the chance that the analysis of its abstract domain will align nicely with a given program.

Some tools clearly provide more evidence across more subjects than others. Unsurprisingly, the tools that performed the best in the reachability category for SV-COMP provide the most evidence across the largest number of subjects. CPA-Seq provided evidence for 207 distinct subjects, VeriAbs for 125, and UAutomizer for 59. So downsizing the portfolio can be effective if there is good evidence (such as competition rankings) that a subset of analyzers can perform well across a subject set.

But in general, it is difficult to predict the best static analyzers that should instantiate an ACA tool portfolio. Even among programs whose general structure is very similar—a read-eval-print-loop whose reachability or unreachability depends on a series of specific inputs (found in the `eca-rers` benchmark category)—there is no tool that dominates the others. In fact, the need of an enlarged portfolio is not reduced by much: only two of the tools are not used; the other *seven* provide effective evidence.

If we cannot predict which tools to use beforehand on any given subject, then maybe we can predict which tool to use within ACA after some observations. The state space of a single program may be uniform in some way, so that if some tools can find characterizing evidence on the first iteration of ACA, they can likely do so for other parts of the state space. If this is the case, then instead of continuing to run each tool in parallel, we could just use the tool or set of tools that was first successful. This held true to some extent when using the *patient* exit strategy,

but not when using the *eager* strategy (see Section 5.1).

Among programs yielding multiple pieces of reachability evidence, at least one tool from among the set of tools that report evidence on the first iteration later found evidence 60% of the time. Put in another way, if you were to downsize the portfolio based on the success of the first iteration of ACA, 40% of the time ACA would not yield more effective evidence. When running with the eager strategy, meaning only a single tool will provide evidence per iteration, these numbers are more drastic: the tool of the first iteration appears in later iterations only 19% of the time.

There are instances when a set of benchmarks can be efficiently solved by a known technique, and in this case it is reasonable to run on a downsized portfolio. For instance, a set of assertions may be known to be proven unreachable using invariants derived from a specific abstract domain, such as the octagon domain. In this case, instantiating an abstract-interpretation based tool with this domain—rather than a faster, but more imprecise one, such as the interval domain—would make sense. But this requires expert knowledge both of the problem domain and of how to configure the tools in the portfolio; this predictive power is not available in general.

9.4 Limitations and Threats to Validity

We chose the SV-COMP benchmark suite because it defines a standard specification format for properties embedded in C programs along with violation witnesses provided by analyzers, which allowed us to use a large number of analysis tools within the framework of ACA. The set of 380 SV-COMP artifacts that made it through ALPACA cover a range of varied program state spaces, but we caution the reader in making conclusions about the external validity of our findings. While this study is a starting point, clearly a broader set of both analyzers and programs is needed to construct a more comprehensive assessment of the interaction amongst programs, analyzers, and reachability properties.

Some of the bottlenecks in analysis time come from how the analyzers share reachability information, which in our case was the error witness specification given by an error automaton [31]. Error automata can specify the constraints on inputs leading to a property violation in different ways, e.g., branch directives indicating the decision taken at each branch, or concrete inputs

at the entry of a program. Because the tool used to characterize reachability within ALPACA accepts only the error specification of branch directives as its form of directed symbolic execution, there was an extra translation step involved—using CPAchecker's witness validator [23]—when a witness was given as a set of concrete inputs. In the future, the tool to characterize reachability should directly accept initial concrete inputs as another form of direction.

Finally, only two among the many configurations of ALPACA were run during this study. The choice of different parameters—e.g., instantiating ALPACA with a different set of tools, running with a downsized portfolio, changing timeout bounds—on a run of ALPACA can significantly affect runtime as well as the produced interval. A broader study involving multiple configurations is needed in order to compare findings across varying instantiations of ACA.

Chapter 10

Conclusion and Future Work

10.1 Summary of Contributions

The contributions of this dissertation are in generating more informative program correctness proofs by combining a diverse set of program analyzers within a meta-analysis framework. The four main contributions of this dissertation are in:

- enriching the results of a program analysis to encode overapproximate and underapproximate information
- developing a novel program meta-analysis framework that combines artifacts produced by over- and underapproximate analyzers to compute such a result
- empirically evaluating that framework
- applying the framework to improve the scalability and accuracy of downstream analyses

We found that nontrivial program intervals can be effectively computed over a broad class of programs, and that these $\mathcal{I}$ depend on a wide diversity of program analyzers computing combinations of *may* and *must* information in order to be assembled. The quality of a returned program interval is dependent on the instantiation of analyzers in ACA, but given an effective set of $\overline{A}$ and $\underline{A}$, the sweet spot for ACA seems to be when there is a relatively small portion of *must* information to be characterized which can be effectively communicated to some $\overline{A}$. Large

sets of *must* information will often cause some generalization to occur; this may occur because the assumptions we use to condition analyzers can overly complicate their abstract domain.

To support more effective meta-analysis, it would be helpful if other analysis frameworks have clearly defined interfaces in which tools can exchange partial results. The work of input interfacing, e.g., with nondeterministic primitives such as `__VERIFIER_nondet_int()`, and output interfacing, e.g., with error automata, has been started by the originators of SV-COMP and has been adopted by many state-of-the-art analysis tools for C and Java. While error (and correctness) witnesses are a good start, we would like to see more generalized witness automata that describe subspaces of a program, not just deterministic paths (or overapproximate invariants). We used embedded C expressions in order to condition analyzers away from already-explored portions of a program, but we could imagine analysis tools also standardizing an `assume` interface to allow for more expressive conditions as are expressed with ACSL [19]. In this way analysis tools can be designed to account for these conditions in a way that helps them explore the remainder of the state space most effectively.

10.2 Future Work

In this section we suggest lines of future work, grouping them into two different directions: improvements to the ACA framework, and uses of its computed program intervals.

10.2.1 Improving ACA

Adaptive Interval Refinement

The disjoint intervals computed by ACA could have a large "gap" between their respective upper and lower bounds. The set of input points between the gap in these bounds denotes inputs that *may* reach the ψ-state, but about which nothing more definite can be said. In general, ACA cannot always produce exact intervals, i.e., ones where the upper and lower bound coincide, as no analyzer (or meta-analyzer) can be sound and complete for *all* programs. But it is possible to refine the accuracy of individual inexact intervals using techniques that are complementary to the ACA framework. In other cases, the intervals may actually be too numerous, and refining in this

case would mean coming up with more general (and simpler) formulae to describe the intervals.

The pseudocode for refinement is outlined in Algorithm 6. There could be multiple inexact intervals, and the SELECT function in line 2 chooses which of these to refine according to a given heuristic (discussed below); because we may want to consider more than one interval at a time, e.g., to generalize semantic similarities, I could be a set of intervals. The function REFINE-INTERVAL in line 3 attempts to refine the interval(s) contained in I using one of the strategies proposed below. The ACCUMULATE function in line 4 is the same as in Algorithm 1. Refinement terminates when a user-defined threshold is reached; this could be as simple as a time bound, or as involved as checking if the reachability of ψ-states in $\mathcal{I}$ sufficiently covers some measure space. The refinement is adaptive in the sense that, depending on the SELECT heuristic, the chosen interval could vary at each recursive call of REFINE.

Algorithm 6 Adaptive Interval Refinement

```
1: function REFINE(I)
2:     I ← SELECT(I)
3:     I' ← REFINE-INTERVAL(I)
4:     I' ← ACCUMULATE(I, I')
5:     if threshold reached then
6:         return I'
7:     else
8:         REFINE(I')
```

Selecting Intervals

The selection heuristic could be (1) random, (2) based on the probability mass associated with the interval, or (3) dependent on characteristics of the formulae defining the interval. The second heuristic allows us to prioritize the analysis of the different intervals based on the value of $Pr(\alpha_i)$ by targeting only profitable intervals. Since $Pr(\alpha_i)$ imposes a bound on the maximum contribution to the final analysis result, smaller intervals can be discarded entirely if their contribution is deemed insufficient. The third heuristic would allow us to focus on formulae that may be good candidates for generalization (e.g., due to a large number of logically "similar" terms), to disregard others as too difficult (e.g., formulae with complex heap manipulation), or to run PSE supported with specialized solvers (e.g., string or numerical solvers). This can tailor the analysis of each interval to its specific features, without relying on more general but less efficient procedures.

Interval Refinement Strategies

We discuss three possible strategies to refine the intervals: the first based on harnessing statistical symbolic execution (SSE) to characterize rare events within the bounds, the second on finding interpolants between the upper and lower bound, and third on employing a semantic generalization of formulae.

Statistical Refinement

The first strategy would build off of the CQA framework discussed above. In particular, we could use the results of ACA to direct SSE to sample only within the space between the interval bounds. Conditioning the analysis within a restricted interval increases the probability of sampling a low-probability execution path leading to a ψ-state.

Consider a Monte Carlo sampling of the execution paths. Assume there exists a path to a ψ-state corresponding to the path condition PC^* whose probability in the original program is a small value ϵ. An unbiased Monte Carlo sampler is expected to take $1/\epsilon$ samples before sampling PC^*. For example, for $\epsilon = 10^{-7}$, 10^7 samples are expected before hitting the ψ-state.

If $PC^* \implies \alpha_i$ (otherwise PC^* would have been already included in $\underline{\mathcal{I}}$ before SSE analysis), then the probability of sampling it for conditional SSE would be increased by a factor $1/Pr(\alpha_i)$. In other words, the smaller the subset of behaviors satisfying α_i, the more likely for the conditional sampler to hit the rare ψ-state. For example, if α_i accounts for 10^{-3} of the domain, the expected number of samples required to hit PC^* is reduced by three orders of magnitude. The strategy of harnessing SSE would help to increase the accuracy in intervals that delineate program spaces containing *extremely* rare events.

Interpolating Inexact Intervals

If you are given two formulae, α and β, and $\alpha \implies \beta$, it's conceivable to imagine some formula $\mathcal{F}$ that lies "in between" the two, so that $\alpha \implies \mathcal{F} \implies \beta$. There is an amazing theorem stating that if $\alpha \implies \beta$, and the two share a common variable, then there *always* exists this "in between" formula $\mathcal{F}$—such a formula is called an *interpolant*. We defined a disjoint interval as having a lower bound that implies its upper bound; and as an interval delineates a logical space that reasons over common variables, interpolation seems relevant. The ideal case of an interval with coinciding bounds is *precisely* an interpolant between $\underline{I}$ and $\overline{I}$.

There are existing tools that return an interpolating formula, given two formulae satisfying the above conditions. But if we find such an interpolant between some interval's upper and lower bounds, we cannot simply replace the interval with this formula. As the interpolant "splits" the interval in some sense, we must either prove that all program behavior "below" the interpolant (i.e., between the lower bound and the interpolant) *definitely* reaches a ψ-state, or that all behavior "above" (i.e., between the interpolant and the upper bound) *definitely* does *not* reach a ψ-state. We consider the first case as "raising the lower bound" and the second case as "lowering the upper bound." If both can be done, the result is an exact interval, i.e., the lower bound coincides with the upper bound.

To lower the upper bound safely, we will need to use the guarantees of an overapproximator. If some $\overline{A}$ did not prove the unreachability of a ψ-state in the space between an interpolant and an upper bound on an initial run of ACA, why should some $\overline{A}$ be able to prove it this time around? Because an interpolant can be significantly simpler than either of its two bounding formulae. The abstract domain of an $\overline{A}$ could become overly coarse with complex terms in the upper and lower bound, but may handle the interpolant with ease. Searching the space between the interpolant $\mathcal{F}$ and the upper bound $\overline{I}$ amounts to calling the overapproximators on $\neg\mathcal{F} \wedge \overline{I}$, and looking for unreachability evidence.

To raise the lower bound safely, we must examine all paths between the lower bound and the interpolant and confirm that they reach a ψ-state. For large state spaces, even this restricted search will not finish before running out of resources; but the lower bound can be enriched if any new reachable paths to a ψ-state are characterized. Searching the space between the lower bound $\underline{I}$ and the interpolant $\mathcal{F}$ amounts to running a symbolic execution on $\neg\underline{I} \wedge \mathcal{F}$, and collecting any evidence of ψ-state reachability.

The choice of an appropriate interpolant plays a big role in whether either of the bounds can be made more accurate, i.e., moved closer to the interpolant.

Inferring Interval Invariants

The third strategy is motivated by situations where we actually *want* to create an inexact interval. Why would we want this? There are cases where $\mathcal{I}$ grows large in the number of intervals due to reachable ψ-state paths that are semantically similar, but syntactically distinct.

In principle this set of paths can be unbounded.

Consider the program fragment: $\ldots$**while** $(x > 0)\{\psi\}\ldots$, where x is some symbolic integer. After a few iterations, ACA could compute a series of exact, disjoint intervals; the succession of coinciding lower/upper bounds may be: $[x = 1, x = 1]$, followed by $[x = 2, x = 2]$, followed by $[x = 3, x = 3]$, and so on. This overly-fine granularity arises because the bounding condition of the **while** loop is not explicit in the given path conditions produced by $\underline{A}$. Though the generalization needed $(x > 0)$ is obvious in this case, the patterns we hope to automatically infer will often be much more complex.

So for the third strategy, we want to compute a disjoint interval I' such that $I \implies I'$. (Note that this implication conceptually "raises" the upper bound of I while leaving the lower bounds intact; we only want to generalize the upper bound.) We would like I' to be small in the sense that all or mostly all behavior described by I' actually leads to a ψ-state. To produce a minimal generalization of this kind, we plan to build on recent work that infers program invariants using symbolic states—SymInfer [166].

SymInfer is conceptually similar to the dynamic inference of formulae pioneered by Daikon [91]; but unlike Daikon, the newer work validates or invalidates the candidate invariants using symbolic states—meaning no spurious invariants are produced. Also, by using a counterexample-guided algorithm, more accurate invariants are produced. This makes SymInfer ideal for generating our desired "minimally"-generalized intervals. The SELECT function would choose a set of syntactically-similar formulae (e.g., all formulae involving an equality constraint on the same variable), and REFINE-INTERVAL would run SymInfer to see if a more general interval can be inferred to cover the numerous "similar" intervals.

Statically Decomposing Programs

The current presentation of ACA assumes a program with the existence of a reachable ψ-state, and as ACA proceeds, this program is dynamically decomposed with `assume` statements. We can also use ACA to statically decompose a program with or without ψ-state reachability. Each of these decompositions can then be again given to ACA, or to some other analyzer.

We can imagine partitioning a program that contains some line with a call to `psi()`—which may or may not be reachable—, by chopping the program domain in two in the following manner:

1. make a copy of the original program, call the copy `p'`
2. comment out the call to `psi()` in `p'`
3. inject a new error state in `p'` at some:
   - loop header
   - loop exit
   - just before a function callsite
   - just after a function callsite
4. try to collect a single reachability condition within `p'`, call it `r`
5. insert either `assume(r)` or `assume(!r)` at the top of the original program

If ACA successfully finds a path to an injected error, it can produce a file that contains an assumption and mappings from input variables to their counts and types so that a later run of ACA can create initial instrumentation that makes sense, i.e., all input variables mentioned in the assumption have been initialized. Now ACA (or some other analyzer) chews again on the original program with the embedded assumptions, with the hope being that it is easier to reason about the smaller program pieces.

Reconstructing $\mathcal{I}$ in modular ACA

The end of Chapter 7 discussed the possibility of propagating the sound *must* information in $\underline{\mathcal{I}}$ across function boundaries. This would produce a succession of propagated lower bounds: $\underline{\mathcal{I}}_1$, $\underline{\mathcal{I}}_2$, $\underline{\mathcal{I}}_3$.

We imagine this list of lower bounds can be paired with the list of upper bounds (whose generation is described in Chapter 7), e.g., $(\overline{\mathcal{I}}_1, \underline{\mathcal{I}}_1)$, $(\overline{\mathcal{I}}_2, \underline{\mathcal{I}}_2)$, ..., such that a program interval with both bounds defined can be reconstructed at each depth. Unlike a typical run of ACA, here the $\underline{\mathcal{I}}_i$ and $\overline{\mathcal{I}}_i$ are produced independently of each other. We hypothesize that two complementary runs of modular ACA—one computing series of $\underline{\mathcal{I}}$, the other series of $\overline{\mathcal{I}}$—can reconstruct subsystem program intervals sound in both bounds.

10.2.2 Using Program Intervals

We presented promising applications of program intervals in focusing the results of expensive quantitative techniques in Chapter 8, and in computing a weakest precondition in a modular setting in Chapter 7. Here we look at two other possible applications of $\mathcal{I}$.

Predictive Failure Avoidance

Due to the complexity of modern software, we cannot always have a proof that a program is free of potential failures before deploying it. But if we are given an idea of how a failure may occur, it is possible to detect a possible failure on a given execution path in order to steer execution away from the failure. The hope would be to avoid the failure with minimally affecting program behavior.

If the ψ-states are defined as error locations, then the program interval produced by ACA could be used as a runtime monitor for predicting a failure. The logical formulae given by $\overline{\mathcal{I}}$ can be used to monitor the runtime constraints on inputs. By performing branch-lookahead to see what the upcoming constraint will be, you can check if it is implied by $\overline{\mathcal{I}}$; if so, there's the possibility of hitting an error state. If this possibility is detected, we would need to define some neighborhood of "similar" execution paths that could avoid avoid being implied by $\overline{\mathcal{I}}$, and force execution to take one of these alternate paths.

Using program intervals for comprehensive bug fixes

We mentioned at the beginning of Chapter 3 the possibility of using $\mathcal{I}$ to provide developers with a broader summary of conditions on inputs that *must* lead to a failure. If there are multiple paths to some failure, this more comprehensive summary could be helpful in guiding the developer to make a single general bug fix, rather than having to discover, debug, and patch each individual path to a failure over time.

In the context of bug fixes, we only want to show developers known failures, so we would output the *must* information collected in $\underline{\mathcal{I}}$. While the slicing within *characterize* has already attempted to safely generalize ψ-state reachability conditions by removing conjuncts that are irrelevant to reaching a ψ-state, there could still be a large number of distinct conditions in $\underline{\mathcal{I}}$.

We want to group these in some meaningful way so that the developer is not overwhelmed by a large set of complex logical formulae.

Some of these conditions are grouped by logical implication under their disjoint interval's upper bound $\overline{I_i}$, but we can imagine other grouping schemes, e.g., conjunctions with the longest shared prefix, conjunctions containing only disequalities, conjunctions containing nonlinear constraints, etc. We could also visualize the paths in $\underline{\mathcal{I}}$ by projecting its conditions onto a simplified program model such as a control flow graph. Developing different groupings by similarity metrics, or making commonalities apparent visually would show a developer multiple perspectives on how a ψ-state could be reached. This could provide deeper insight than a single failure report, leading to a more comprehensive bug fix.

Bibliography

[1] Homepage of chrony. `https://chrony.tuxfamily.org/`, Accessed Jun 13, 2021.

[2] Pentest-report chrony 08.2017. `https://web.archive.org/web/20171005123643/https://wiki.mozilla.org/images/e/e4/Chrony-report.pdf`, Accessed Jun 13, 2021.

[3] Ultimate toolchain. `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/`, Accessed Jun 5, 2021.

[4] G. Agha and K. Palmskog. A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.*, 28(1):6:1–6:39, Jan 2018.

[5] H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[6] E. Alatawi, T. Miller, et al. Leveraging abstract interpretation for efficient dynamic symbolic execution. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 619–624. IEEE, 2017.

[7] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *International Static Analysis Symposium*, pages 300–316. Springer, 2012.

[8] A. Albarghouthi, A. Gurfinkel, and M. Chechik. From under-approximations to over-approximations and back. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 157–172, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *International Conference on Computer Aided Verification*, pages 313–329. Springer, 2013.

[10] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[11] 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*, pages 355–359. Springer, 2017.

[12] async: Run IO operations asynchronously and wait for their results. `https://hackage.haskell.org/package/async`.

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

[14] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.

[15] 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.

[16] T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172. Springer, 2015.

[17] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, 2005.

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

[19] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. Acsl: Ansi c specification language. *CEA-LIST, Saclay, France, Tech. Rep. v1*, 2, 2008.

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

[21] D. Beyer. Automatic verification of c and java programs: Sv-comp 2019. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–155, Cham, 2019. Springer International Publishing.

[22] D. Beyer and M. Dangl. Strategy selection for software verification based on boolean features. In *International Symposium on Leveraging Applications of Formal Methods*, pages 144–159. Springer, 2018.

[23] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 721–733. ACM, 2015.

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

[25] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, pages 504–518. Springer, 2007.

[26] D. Beyer and M.-C. Jakobs. Cooperative verifier-based testing with coveritest. *International Journal on Software Tools for Technology Transfer*, pages 1–21, 2021.

[27] 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*, pages 1182–1193. ACM, 2018.

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

[29] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, pages 1–29, 2017.

[30] D. Beyer and H. Wehrheim. Verification artifacts in cooperative verification: Survey and unifying component framework. In *International Symposium on Leveraging Applications of Formal Methods*, pages 143–167. Springer, 2020.

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

[32] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

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

[34] D. W. Binkley and K. B. Gallagher. Program slicing. *Advances in computers*, 43:1–50, 1996.

[35] 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*, pages 632–642. ACM, 2014.

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

[37] 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*, PLDI '14, pages 123–132, New York, NY, USA, 2014. ACM.

[38] M. Borges, A. Filieri, M. d'Amorim, and C. S. Păsăreanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2015, pages 866–877. ACM, 2015.

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

[40] 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*, ASPLOS '14, pages 51–66, New York, NY, USA, 2014. ACM.

[41] A. R. Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[42] A. R. Bradley. Understanding ic3. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 1–14. Springer, 2012.

[43] G. Brat, J. A. Navas, N. Shi, and A. Venet. Ikos: A framework for static analysis based on abstract interpretation. In *International Conference on Software Engineering and Formal Methods*, pages 271–277. Springer, 2014.

[44] 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*, ISSTA 2018, pages 27–37, New York, NY, USA, 2018. ACM.

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

[46] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. Softw. Eng.*, 19(1):3–12, Jan. 1993.

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

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

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

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

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

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

[53] 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*, IJCAI'16, pages 3569–3576. AAAI Press, 2016.

[54] B. Chen, Y. Liu, and W. Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 49–60, New York, NY, USA, 2016. ACM.

[55] D. Chistikov, R. Dimitrova, and R. Majumdar. Approximate counting in smt and value estimation for probabilistic programs. *Acta Informatica*, 54(8):729–764, Dec 2017.

[56] M. Christakis. On narrowing the gap between verification and systematic testing. *it-Information Technology*, 59(4):197–202, 2017.

[57] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *International Symposium on Formal Methods*, pages 132–146. Springer, 2012.

[58] 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*, pages 144–155, 2016.

[59] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

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

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

[62] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[63] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[64] L. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[65] 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*, pages 1–15, 2014.

[66] A. Cortesi. Widening operators for abstract interpretation. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40. IEEE, 2008.

[67] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, 1978.

[68] P. Cousot. The calculational design of a generic abstract interpreter. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:421–506, 1999.

[69] 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*, pages 238–252. ACM, 1977.

[70] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2-3):103–179, 1992.

[71] P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

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

[73] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *Programming Languages and Systems*, pages 169–193. Springer, 2012.

[74] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[75] M. Czech, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. Predicting rankings of software verification tools. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*, pages 23–26, 2017.

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

[77] P. Daca, A. Gupta, and T. A. Henzinger. Abstraction-driven concolic testing. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 328–347. Springer, 2016.

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

[79] 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*, pages 457–462. Springer, 2018.

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

[81] L. De Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *International Conference on Computer Aided Verification*, pages 14–26. Springer, 2003.

[82] Y. Demyanova, T. Pani, H. Veith, and F. Zuleger. Empirical software metrics for benchmarking of verification tools. *Formal methods in system design*, 50(2-3):289–316, 2017.

[83] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

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

[85] 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.

[86] M. B. Dwyer and S. Elbaum. Unifying verification and validation techniques: relating behavior and properties through partial evidence. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 93–98, 2010.

[87] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser. Formal software analysis emerging trends in software model checking. In *Future of Software Engineering (FOSE'07)*, pages 120–136. IEEE, 2007.

[88] M. B. Dwyer, S. Person, and S. G. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In M. Young and P. T. Devanbu, editors, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pages 92–104. ACM, 2006.

[89] M. Dyer, A. Frieze, and R. Kannan. A random polynomial-time algorithm for approximating the volume of convex bodies. *J. ACM*, 38(1):1–17, Jan. 1991.

[90] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, pages 169–181, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[91] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

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

[93] 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*, pages 174–185, 2017.

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

[95] 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*, FSE 14, pages 437–448, New York, NY, USA, 2014. ACM.

[96] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[97] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro. Handling loops in bounded model checking of c programs via k-induction. *International Journal on Software Tools for Technology Transfer*, 19(1):97–114, 2017.

[98] S. Gao, M. Ganai, F. Ivančić, A. Gupta, S. Sankaranarayanan, and E. M. Clarke. Integrating icp and lra solvers for deciding nonlinear real arithmetic problems. In *Formal Methods in Computer Aided Design*, pages 81–89. IEEE, 2010.

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

[100] M. Gerrard, M. Borges, M. Dwyer, and A. Fillieri. Conditional quantitative program analysis. *IEEE Transactions on Software Engineering*, 2020.

[101] M. J. Gerrard and M. B. Dwyer. Comprehensive failure characterization. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 365–376, Piscataway, NJ, USA, 2017. IEEE Press.

[102] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. 40(6):213–223, 2005.

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

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

[105] H. H. Goldstine, J. Von Neumann, and J. Von Neumann. Planning and coding of problems for an electronic computing instrument. 1947.

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

[107] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 167–181, New York, NY, USA, 2014. ACM.

[108] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: An interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, Mar. 2006.

[109] 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*, pages 399–403. Springer, 2017.

[110] M. Greitschus, D. Dietsch, and A. Podelski. Loop invariants from counterexamples. In *International Static Analysis Symposium*, pages 128–147. Springer, 2017.

[111] 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*, ICSE '07, pages 621–631, Washington, DC, USA, 2007. IEEE Computer Society.

[112] 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*, pages 117–127, 2006.

[113] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[114] Á. Hajdu and Z. Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020.

[115] J. Haltermann and H. Wehrheim. Covegi: Cooperative verification via externally generated invariants. *Fundamental Approaches to Software Engineering*, 12649:108, 2021.

[116] language-c: Analysis and generation of C code. `https://hackage.haskell.org/package/language-c-0.9`.

[117] M. P. E. Heimdahl. Safety and software intensive systems: Challenges old and new. In *2007 Future of Software Engineering*, pages 137–152, 2007.

[118] 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*, pages 447–451. Springer, 2018.

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

[120] 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*, 471(2179):20150142, 2015.

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

[122] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 73–84, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[123] S. Heule, M. Sridharan, and S. Chandra. Mimic: Computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 710–720, 2015.

[124] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

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

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

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

[129] International Organization for Standards. ISO 13482: Robots and robotic devices – safety requirements for personal care robots, 2014.

[130] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest precondition reasoning for expected run–times of probabilistic programs. In *European Symposium on Programming*, pages 364–389. Springer, 2016.

[131] J.-P. Katoen. The probabilistic model checking landscape. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 31–45, New York, NY, USA, 2016. ACM.

[132] B. W. Kernighan and D. M. Ritchie. *The C programming language*. 2006.

[133] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *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*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[134] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[135] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[137] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[138] D. Kroening and M. Tautschnig. Cbmc–c bounded model checker. In *Proc. TACAS*, pages 389–391. Springer, 2014.

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

[140] 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.

[141] A. Lai and S. Qadeer. A program transformation for faster goal-directed search. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 147–154. IEEE, 2014.

[142] J. Launchbury and S. L. P. Jones. State in haskell. *Lisp and symbolic computation*, 8(4):293–341, 1995.

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

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

[145] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[146] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995.

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

[148] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[149] D. Lohar, C. Jeangoudoux, J. Sobel, E. Darulova, and M. Christakis. A two-phase approach for conditional floating-point verification. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021)*, 12652:43—63, February 2021.

[150] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 75–87, 2020.

[151] 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*, ASE '14, pages 575–586, New York, NY, USA, 2014. ACM.

[152] 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*, PLDI '14, pages 565–576, New York, NY, USA, 2014. ACM.

[153] 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)*, pages 313–327, July 2018.

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

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

[156] S. Marlow. *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming.* " O'Reilly Media, Inc.", 2013.

[157] S. Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[158] 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*, pages 97–104. IEEE, 2011.

[159] K. L. McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.

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

[161] E. Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[162] D. Monniaux. Abstract interpretation of probabilistic semantics. In *Static Analysis*, pages 322–339. Springer, 2000.

[163] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer. Esbmc 1.22. In *Proc. TACAS*, pages 405–407. Springer, 2014.

[164] 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*, 56(7):689–717, 2014.

[165] K. S. Namjoshi and Z. Pavlinovic. The impact of program transformations on static program analysis. In *International Static Analysis Symposium*, pages 306–325. Springer, 2018.

[166] T. Nguyen, M. B. Dwyer, and W. Visser. Symlnfer: Inferring program invariants using symbolic states. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 804–814. IEEE, 2017.

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

[168] H. R. Nielson and F. Nielson. *Semantics with applications*, volume 104. Springer, 1992.

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

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

[171] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 53–68, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[172] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *International Journal on Software Tools for Technology Transfer*, 5(1):34–48, 2003.

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

[174] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, 1993.

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

[176] G. Ramalingam. Data flow frequency analysis. In *ACM SIGPLAN Notices*, volume 31, pages 267–277. ACM, 1996.

[177] L. M. Rea and R. A. Parker. *Designing and conducting survey research: A comprehensive guide*. John Wiley & Sons, 2014.

[178] T. Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

[179] J. R. Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[180] C. Richter, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. Algorithm selection for software validation based on graph kernels. *Automated Software Engineering*, 27(1):153–186, 2020.

[181] 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*, pages 229–233. Springer, 2019.

[182] C. Richter and H. Wehrheim. Attend and represent: A novel view on algorithm selection for software verification. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1016–1028. IEEE, 2020.

[183] 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*, pages 132–143, 2016.

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

[185] P. Rodgers, G. Stapleton, and P. Chapman. Visualizing sets with linear diagrams. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(6):1–39, 2015.

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

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

[188] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. *ACM SIGPLAN Notices*, 49(6):112–122, 2014.

[189] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. *ACM SIGPLAN Notices*, 48(6):447–458, 2013.

[190] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *International Static Analysis Symposium*, pages 351–380. Springer, 1998.

[191] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[192] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers. CIVL: The Concurrency Intermediate Verification Language. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, Proceedings*, SC '15, Piscataway, NJ, USA, Nov 2015. IEEE Press.

[193] J. Slabỳ, J. Strejček, and M. Trtík. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 207–221. Springer, 2012.

[194] J. Slaby, J. Strejček, and M. Trtík. Symbiotic: synergy of instrumentation, slicing, and symbolic execution. In *Proc. TACAS*, pages 630–632. Springer, 2013.

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

[196] SV-COMP 2016 results. `https://sv-comp.sosy-lab.org/2016/results/`, Accessed May 1, 2017.

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

[198] SV-COMP homepage. `https://sv-comp.sosy-lab.org/`, Accessed Apr 15, 2021.

[199] T. Tao. *An Introduction to Measure Theory*. American Mathematical Society, 2011.

[200] M.-T. Trinh, D.-H. Chu, and J. Jaffar. Model counting for recursively-defined strings. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification*, pages 399–418, Cham, 2017. Springer.

[201] A. Turing. Checking a large routine. In *The early British computer conferences*, pages 70–72, 1989.

[202] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[203] R. van Tonder and C. L. Goues. Tailoring programs for static analysis via program transformation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 824–834, 2020.

[204] S. Verdoolaege. Software package barvinok. 2004. *Electronically available at http://freshmeat.net/projects/barvinok*.

[205] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, Mar. 2007.

[206] 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*, ICSE '18, pages 291–302, New York, NY, USA, 2018. ACM.

[207] 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*, POPL '85, pages 291–299, New York, NY, USA, 1985. ACM.

[208] M. Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[209] H. Wong-Toi. Symbolic approximations for verifying real-time systems. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1994.

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

[211] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008.

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

[213] H. Ye, M. Martinez, and M. Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2):1–38, 2021.

[214] G. U. Yule. On the methods of measuring the association between two variables. *Journal of the Royal Statistics Society*, 75:576–642, 1912.

[215] 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*, ASE, pages 830–835, 2015.

[216] P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to stateflow/simulink verification. *Formal Methods in System Design*, 43(2):338–367, Oct 2013.

Appendices

Appendix A

Observational Study Data

This appendix contains performance metrics for runs of ALPACA on the 380 SV-COMP subjects discussed in Chapter 6. The mapping between subject IDs and original SV-COMP names is in Section A.1. Detailed results for each subject are in Section A.2.

A.1 Name mapping

For readability reasons, we shortened the filename of some subjects. Here below is a table mapping each name id to the original filename from the SV-COMP benchmark, which can be found at: `https://github.com/sosy-lab/sv-benchmarks`.

Notably, the SV-COMP organizers renamed some of the subjects. However, the original filename is stored in the yml description of the subject, e.g., see "old file name" in `email_spec0_product31.cil.yml`.

The recommended way to access the benchmark files is searching the SV-COMP repository for the filename using the search function of GitHub.

Subject ID	SV-COMP file name
Ackermann02	Ackermann02˙false-unreach-call˙true-no-overflow˙true-termination.c
Addition02	Addition02˙false-unreach-call˙true-no-overflow˙true-termination.c
array	array˙false-unreach-call˙true-termination.i
BallRajamani-SPIN2000-Fig1	BallRajamani-SPIN2000-Fig1˙false-unreach-call˙true-no-overflow˙true-termination.c
cdaudio_simpl1	cdaudio˙simpl1˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
elevator_spec14_productSimulator	elevator˙spec14˙productSimulator˙false-unreach-call˙true-termination.cil.c
elevator_spec1_productSimulator	elevator˙spec1˙productSimulator˙false-unreach-call˙true-termination.cil.c
elevator_spec2_productSimulator	elevator˙spec2˙productSimulator˙false-unreach-call˙true-termination.cil.c
elevator_spec3_productSimulator	elevator˙spec3˙productSimulator˙false-unreach-call˙true-termination.cil.c
elevator_spec9_productSimulator	elevator˙spec9˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec0_product16	email˙spec0˙product16˙false-unreach-call˙true-termination.cil.c
email_spec0_product21	email˙spec0˙product21˙false-unreach-call˙true-termination.cil.c
email_spec0_product22	email˙spec0˙product22˙false-unreach-call˙true-termination.cil.c
email_spec0_product26	email˙spec0˙product26˙false-unreach-call˙true-termination.cil.c
email_spec0_product31	email˙spec0˙product31˙false-unreach-call˙true-termination.cil.c
email_spec0_product33	email˙spec0˙product33˙false-unreach-call˙true-termination.cil.c
email_spec0_product34	email˙spec0˙product34˙false-unreach-call˙true-termination.cil.c
email_spec0_product35	email˙spec0˙product35˙false-unreach-call˙true-termination.cil.c
email_spec0_productSimulator	email˙spec0˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec11_product15	email˙spec11˙product15˙false-unreach-call˙true-termination.cil.c
email_spec11_product20	email˙spec11˙product20˙false-unreach-call˙true-termination.cil.c
email_spec11_product22	email˙spec11˙product22˙false-unreach-call˙true-termination.cil.c
email_spec11_product26	email˙spec11˙product26˙false-unreach-call˙true-termination.cil.c
email_spec11_product30	email˙spec11˙product30˙false-unreach-call˙true-termination.cil.c
email_spec11_product32	email˙spec11˙product32˙false-unreach-call˙true-termination.cil.c
email_spec11_product33	email˙spec11˙product33˙false-unreach-call˙true-termination.cil.c
email_spec11_product35	email˙spec11˙product35˙false-unreach-call˙true-termination.cil.c
email_spec11_productSimulator	email˙spec11˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec1_product14	email˙spec1˙product14˙false-unreach-call˙true-termination.cil.c
email_spec1_product15	email˙spec1˙product15˙false-unreach-call˙true-termination.cil.c
email_spec1_product16	email˙spec1˙product16˙false-unreach-call˙true-termination.cil.c
email_spec1_product20	email˙spec1˙product20˙false-unreach-call˙true-termination.cil.c
email_spec1_product21	email˙spec1˙product21˙false-unreach-call˙true-termination.cil.c
email_spec1_product22	email˙spec1˙product22˙false-unreach-call˙true-termination.cil.c
email_spec1_product26	email˙spec1˙product26˙false-unreach-call˙true-termination.cil.c
email_spec1_product30	email˙spec1˙product30˙false-unreach-call˙true-termination.cil.c

Subject ID	SV-COMP file name
email_spec1_product31	email˙spec1˙product31˙false-unreach-call˙true-termination.cil.c
email_spec1_product32	email˙spec1˙product32˙false-unreach-call˙true-termination.cil.c
email_spec1_product33	email˙spec1˙product33˙false-unreach-call˙true-termination.cil.c
email_spec1_product34	email˙spec1˙product34˙false-unreach-call˙true-termination.cil.c
email_spec1_product35	email˙spec1˙product35˙false-unreach-call˙true-termination.cil.c
email_spec1_productSimulator	email˙spec1˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec27_product17	email˙spec27˙product17˙false-unreach-call˙true-termination.cil.c
email_spec27_product18	email˙spec27˙product18˙false-unreach-call˙true-termination.cil.c
email_spec27_product19	email˙spec27˙product19˙false-unreach-call˙true-termination.cil.c
email_spec27_product23	email˙spec27˙product23˙false-unreach-call˙true-termination.cil.c
email_spec27_product24	email˙spec27˙product24˙false-unreach-call˙true-termination.cil.c
email_spec27_product25	email˙spec27˙product25˙false-unreach-call˙true-termination.cil.c
email_spec27_product27	email˙spec27˙product27˙false-unreach-call˙true-termination.cil.c
email_spec27_product29	email˙spec27˙product29˙false-unreach-call˙true-termination.cil.c
email_spec27_product30	email˙spec27˙product30˙false-unreach-call˙true-termination.cil.c
email_spec27_product31	email˙spec27˙product31˙false-unreach-call˙true-termination.cil.c
email_spec27_product32	email˙spec27˙product32˙false-unreach-call˙true-termination.cil.c
email_spec27_product33	email˙spec27˙product33˙false-unreach-call˙true-termination.cil.c
email_spec27_product34	email˙spec27˙product34˙false-unreach-call˙true-termination.cil.c
email_spec27_product35	email˙spec27˙product35˙false-unreach-call˙true-termination.cil.c
email_spec27_productSimulator	email˙spec27˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec3_product13	email˙spec3˙product13˙false-unreach-call˙true-termination.cil.c
email_spec3_product17	email˙spec3˙product17˙false-unreach-call˙true-termination.cil.c
email_spec3_product18	email˙spec3˙product18˙false-unreach-call˙true-termination.cil.c
email_spec3_product19	email˙spec3˙product19˙false-unreach-call˙true-termination.cil.c
email_spec3_product23	email˙spec3˙product23˙false-unreach-call˙true-termination.cil.c
email_spec3_product24	email˙spec3˙product24˙false-unreach-call˙true-termination.cil.c
email_spec3_product25	email˙spec3˙product25˙false-unreach-call˙true-termination.cil.c
email_spec3_product27	email˙spec3˙product27˙false-unreach-call˙true-termination.cil.c
email_spec3_product28	email˙spec3˙product28˙false-unreach-call˙true-termination.cil.c
email_spec3_product29	email˙spec3˙product29˙false-unreach-call˙true-termination.cil.c
email_spec3_product30	email˙spec3˙product30˙false-unreach-call˙true-termination.cil.c
email_spec3_product31	email˙spec3˙product31˙false-unreach-call˙true-termination.cil.c
email_spec3_product32	email˙spec3˙product32˙false-unreach-call˙true-termination.cil.c
email_spec3_product33	email˙spec3˙product33˙false-unreach-call˙true-termination.cil.c
email_spec3_product34	email˙spec3˙product34˙false-unreach-call˙true-termination.cil.c

Subject ID	SV-COMP file name
email_spec3_product35	email˙spec3˙product35˙false-unreach-call˙true-termination.cil.c
email_spec3_productSimulator	email˙spec3˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec4_product18	email˙spec4˙product18˙false-unreach-call˙true-termination.cil.c
email_spec4_product19	email˙spec4˙product19˙false-unreach-call˙true-termination.cil.c
email_spec4_product23	email˙spec4˙product23˙false-unreach-call˙true-termination.cil.c
email_spec4_product24	email˙spec4˙product24˙false-unreach-call˙true-termination.cil.c
email_spec4_product25	email˙spec4˙product25˙false-unreach-call˙true-termination.cil.c
email_spec4_product27	email˙spec4˙product27˙false-unreach-call˙true-termination.cil.c
email_spec4_product30	email˙spec4˙product30˙false-unreach-call˙true-termination.cil.c
email_spec4_product31	email˙spec4˙product31˙false-unreach-call˙true-termination.cil.c
email_spec4_product32	email˙spec4˙product32˙false-unreach-call˙true-termination.cil.c
email_spec4_product33	email˙spec4˙product33˙false-unreach-call˙true-termination.cil.c
email_spec4_product34	email˙spec4˙product34˙false-unreach-call˙true-termination.cil.c
email_spec4_product35	email˙spec4˙product35˙false-unreach-call˙true-termination.cil.c
email_spec4_productSimulator	email˙spec4˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec6_product12	email˙spec6˙product12˙false-unreach-call˙true-termination.cil.c
email_spec6_product14	email˙spec6˙product14˙false-unreach-call˙true-termination.cil.c
email_spec6_product15	email˙spec6˙product15˙false-unreach-call˙true-termination.cil.c
email_spec6_product16	email˙spec6˙product16˙false-unreach-call˙true-termination.cil.c
email_spec6_product20	email˙spec6˙product20˙false-unreach-call˙true-termination.cil.c
email_spec6_product21	email˙spec6˙product21˙false-unreach-call˙true-termination.cil.c
email_spec6_product22	email˙spec6˙product22˙false-unreach-call˙true-termination.cil.c
email_spec6_product26	email˙spec6˙product26˙false-unreach-call˙true-termination.cil.c
email_spec6_product28	email˙spec6˙product28˙false-unreach-call˙true-termination.cil.c
email_spec6_product29	email˙spec6˙product29˙false-unreach-call˙true-termination.cil.c
email_spec6_product30	email˙spec6˙product30˙false-unreach-call˙true-termination.cil.c
email_spec6_product31	email˙spec6˙product31˙false-unreach-call˙true-termination.cil.c
email_spec6_product32	email˙spec6˙product32˙false-unreach-call˙true-termination.cil.c
email_spec6_product33	email˙spec6˙product33˙false-unreach-call˙true-termination.cil.c
email_spec6_product34	email˙spec6˙product34˙false-unreach-call˙true-termination.cil.c
email_spec6_product35	email˙spec6˙product35˙false-unreach-call˙true-termination.cil.c
email_spec6_productSimulator	email˙spec6˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec7_product28	email˙spec7˙product28˙false-unreach-call˙true-termination.cil.c
email_spec7_product29	email˙spec7˙product29˙false-unreach-call˙true-termination.cil.c
email_spec7_product30	email˙spec7˙product30˙false-unreach-call˙true-termination.cil.c
email_spec7_product31	email˙spec7˙product31˙false-unreach-call˙true-termination.cil.c

Subject ID	SV-COMP file name
email_spec7_product32	email˙spec7˙product32˙false-unreach-call˙true-termination.cil.c
email_spec7_product33	email˙spec7˙product33˙false-unreach-call˙true-termination.cil.c
email_spec7_product34	email˙spec7˙product34˙false-unreach-call˙true-termination.cil.c
email_spec7_product35	email˙spec7˙product35˙false-unreach-call˙true-termination.cil.c
email_spec7_productSimulator	email˙spec7˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec8_product15	email˙spec8˙product15˙false-unreach-call˙true-termination.cil.c
email_spec8_product16	email˙spec8˙product16˙false-unreach-call˙true-termination.cil.c
email_spec8_product20	email˙spec8˙product20˙false-unreach-call˙true-termination.cil.c
email_spec8_product21	email˙spec8˙product21˙false-unreach-call˙true-termination.cil.c
email_spec8_product22	email˙spec8˙product22˙false-unreach-call˙true-termination.cil.c
email_spec8_product26	email˙spec8˙product26˙false-unreach-call˙true-termination.cil.c
email_spec8_product30	email˙spec8˙product30˙false-unreach-call˙true-termination.cil.c
email_spec8_product31	email˙spec8˙product31˙false-unreach-call˙true-termination.cil.c
email_spec8_product33	email˙spec8˙product33˙false-unreach-call˙true-termination.cil.c
email_spec8_product34	email˙spec8˙product34˙false-unreach-call˙true-termination.cil.c
email_spec8_product35	email˙spec8˙product35˙false-unreach-call˙true-termination.cil.c
email_spec8_productSimulator	email˙spec8˙productSimulator˙false-unreach-call˙true-termination.cil.c
email_spec9_product15	email˙spec9˙product15˙false-unreach-call˙true-termination.cil.c
email_spec9_product16	email˙spec9˙product16˙false-unreach-call˙true-termination.cil.c
email_spec9_product20	email˙spec9˙product20˙false-unreach-call˙true-termination.cil.c
email_spec9_product21	email˙spec9˙product21˙false-unreach-call˙true-termination.cil.c
email_spec9_product22	email˙spec9˙product22˙false-unreach-call˙true-termination.cil.c
email_spec9_product26	email˙spec9˙product26˙false-unreach-call˙true-termination.cil.c
email_spec9_product30	email˙spec9˙product30˙false-unreach-call˙true-termination.cil.c
email_spec9_product31	email˙spec9˙product31˙false-unreach-call˙true-termination.cil.c
email_spec9_product32	email˙spec9˙product32˙false-unreach-call˙true-termination.cil.c
email_spec9_product33	email˙spec9˙product33˙false-unreach-call˙true-termination.cil.c
email_spec9_product34	email˙spec9˙product34˙false-unreach-call˙true-termination.cil.c
email_spec9_product35	email˙spec9˙product35˙false-unreach-call˙true-termination.cil.c
EvenOdd03	EvenOdd03˙false-unreach-call˙true-no-overflow˙true-termination.c
Fibonacci04	Fibonacci04˙false-unreach-call˙true-no-overflow˙true-termination.c
Fibonacci05	Fibonacci05˙false-unreach-call˙true-no-overflow˙true-termination.c
floppy_simpl3	floppy˙simpl3˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
floppy_simpl4	floppy˙simpl4˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
for_bounded_loop1	for˙bounded˙loop1˙false-unreach-call˙true-termination.i
id_b3_o2	id˙b3˙o2˙false-unreach-call˙true-termination˙true-no-overflow.c

Subject ID	SV-COMP file name
kbfiltr_simpl2	kbfiltr˙simpl2˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
kundu2	kundu2˙false-unreach-call˙false-termination.cil.c
McCarthy91	McCarthy91˙false-unreach-call˙true-no-overflow˙true-termination.c
minepump_spec1_product33	minepump˙spec1˙product33˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product34	minepump˙spec1˙product34˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product35	minepump˙spec1˙product35˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product36	minepump˙spec1˙product36˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product37	minepump˙spec1˙product37˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product38	minepump˙spec1˙product38˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product39	minepump˙spec1˙product39˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product40	minepump˙spec1˙product40˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product42	minepump˙spec1˙product42˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product43	minepump˙spec1˙product43˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product44	minepump˙spec1˙product44˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product49	minepump˙spec1˙product49˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product50	minepump˙spec1˙product50˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product51	minepump˙spec1˙product51˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product52	minepump˙spec1˙product52˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product53	minepump˙spec1˙product53˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product54	minepump˙spec1˙product54˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product55	minepump˙spec1˙product55˙false-unreach-call˙false-termination.cil.c
minepump_spec1_product56	minepump˙spec1˙product56˙false-unreach-call˙false-termination.cil.c
minepump_spec1_productSimulator	minepump˙spec1˙productSimulator˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product33	minepump˙spec2˙product33˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product34	minepump˙spec2˙product34˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product35	minepump˙spec2˙product35˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product36	minepump˙spec2˙product36˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product41	minepump˙spec2˙product41˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product42	minepump˙spec2˙product42˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product43	minepump˙spec2˙product43˙false-unreach-call˙false-termination.cil.c
minepump_spec2_product44	minepump˙spec2˙product44˙false-unreach-call˙false-termination.cil.c
minepump_spec2_productSimulator	minepump˙spec2˙productSimulator˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product01	minepump˙spec3˙product01˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product02	minepump˙spec3˙product02˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product03	minepump˙spec3˙product03˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product04	minepump˙spec3˙product04˙false-unreach-call˙false-termination.cil.c

Subject ID	SV-COMP file name
minepump_spec3_product05	minepump˙spec3˙product05˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product06	minepump˙spec3˙product06˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product07	minepump˙spec3˙product07˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product08	minepump˙spec3˙product08˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product09	minepump˙spec3˙product09˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product10	minepump˙spec3˙product10˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product11	minepump˙spec3˙product11˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product12	minepump˙spec3˙product12˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product13	minepump˙spec3˙product13˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product14	minepump˙spec3˙product14˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product15	minepump˙spec3˙product15˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product16	minepump˙spec3˙product16˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product17	minepump˙spec3˙product17˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product18	minepump˙spec3˙product18˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product19	minepump˙spec3˙product19˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product20	minepump˙spec3˙product20˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product21	minepump˙spec3˙product21˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product23	minepump˙spec3˙product23˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product24	minepump˙spec3˙product24˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product25	minepump˙spec3˙product25˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product26	minepump˙spec3˙product26˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product27	minepump˙spec3˙product27˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product28	minepump˙spec3˙product28˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product29	minepump˙spec3˙product29˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product30	minepump˙spec3˙product30˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product31	minepump˙spec3˙product31˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product32	minepump˙spec3˙product32˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product35	minepump˙spec3˙product35˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product36	minepump˙spec3˙product36˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product39	minepump˙spec3˙product39˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product40	minepump˙spec3˙product40˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product43	minepump˙spec3˙product43˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product44	minepump˙spec3˙product44˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product47	minepump˙spec3˙product47˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product48	minepump˙spec3˙product48˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product51	minepump˙spec3˙product51˙false-unreach-call˙false-termination.cil.c

Subject ID	SV-COMP file name
minepump_spec3_product52	minepump˙spec3˙product52˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product55	minepump˙spec3˙product55˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product56	minepump˙spec3˙product56˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product59	minepump˙spec3˙product59˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product60	minepump˙spec3˙product60˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product63	minepump˙spec3˙product63˙false-unreach-call˙false-termination.cil.c
minepump_spec3_product64	minepump˙spec3˙product64˙false-unreach-call˙false-termination.cil.c
minepump_spec3_productSimulator	minepump˙spec3˙productSimulator˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product33	minepump˙spec4˙product33˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product34	minepump˙spec4˙product34˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product35	minepump˙spec4˙product35˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product36	minepump˙spec4˙product36˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product37	minepump˙spec4˙product37˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product38	minepump˙spec4˙product38˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product39	minepump˙spec4˙product39˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product40	minepump˙spec4˙product40˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product41	minepump˙spec4˙product41˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product42	minepump˙spec4˙product42˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product43	minepump˙spec4˙product43˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product44	minepump˙spec4˙product44˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product45	minepump˙spec4˙product45˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product46	minepump˙spec4˙product46˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product47	minepump˙spec4˙product47˙false-unreach-call˙false-termination.cil.c
minepump_spec4_product48	minepump˙spec4˙product48˙false-unreach-call˙false-termination.cil.c
minepump_spec4_productSimulator	minepump˙spec4˙productSimulator˙false-unreach-call˙false-termination.cil.c
newton_1_4	newton˙1˙4˙false-unreach-call˙true-termination.i
newton_1_5	newton˙1˙5˙false-unreach-call˙true-termination.i
newton_1_6	newton˙1˙6˙false-unreach-call˙true-termination.i
newton_1_7	newton˙1˙7˙false-unreach-call˙true-termination.i
pc_sfifo_1	pc˙sfifo˙1˙false-unreach-call˙false-termination.cil.c
pc_sfifo_2	pc˙sfifo˙2˙false-unreach-call˙false-termination.cil.c
Problem01_label15	Problem01˙label15˙false-unreach-call˙false-termination.c
Problem01_label20	Problem01˙label20˙false-unreach-call˙false-termination.c
Problem01_label21	Problem01˙label21˙false-unreach-call˙false-termination.c
Problem01_label32	Problem01˙label32˙false-unreach-call˙false-termination.c
Problem01_label33	Problem01˙label33˙false-unreach-call˙false-termination.c

Subject ID	SV-COMP file name
Problem01_label35	Problem01˙label35˙false-unreach-call˙false-termination.c
Problem01_label37	Problem01˙label37˙false-unreach-call˙false-termination.c
Problem01_label38	Problem01˙label38˙false-unreach-call˙false-termination.c
Problem01_label44	Problem01˙label44˙false-unreach-call˙false-termination.c
Problem01_label47	Problem01˙label47˙false-unreach-call˙false-termination.c
Problem01_label50	Problem01˙label50˙false-unreach-call˙false-termination.c
Problem01_label56	Problem01˙label56˙false-unreach-call˙false-termination.c
Problem01_label57	Problem01˙label57˙false-unreach-call˙false-termination.c
Problem02_label13	Problem02˙label13˙false-unreach-call˙false-termination.c
Problem02_label16	Problem02˙label16˙false-unreach-call˙false-termination.c
Problem02_label43	Problem02˙label43˙false-unreach-call˙false-termination.c
Problem02_label44	Problem02˙label44˙false-unreach-call˙false-termination.c
Problem02_label45	Problem02˙label45˙false-unreach-call˙false-termination.c
Problem02_label50	Problem02˙label50˙false-unreach-call˙false-termination.c
Problem02_label59	Problem02˙label59˙false-unreach-call˙false-termination.c
Problem03_label09	Problem03˙label09˙false-unreach-call.c
Problem03_label13	Problem03˙label13˙false-unreach-call.c
Problem03_label26	Problem03˙label26˙false-unreach-call.c
Problem03_label27	Problem03˙label27˙false-unreach-call.c
Problem03_label28	Problem03˙label28˙false-unreach-call.c
Problem03_label31	Problem03˙label31˙false-unreach-call.c
Problem03_label35	Problem03˙label35˙false-unreach-call.c
Problem03_label37	Problem03˙label37˙false-unreach-call.c
Problem03_label39	Problem03˙label39˙false-unreach-call.c
Problem03_label43	Problem03˙label43˙false-unreach-call.c
Problem03_label45	Problem03˙label45˙false-unreach-call.c
Problem03_label50	Problem03˙label50˙false-unreach-call.c
Problem03_label52	Problem03˙label52˙false-unreach-call.c
Problem04_label19	Problem04˙label19˙false-unreach-call.c
Problem04_label55	Problem04˙label55˙false-unreach-call.c
Problem06_label05	Problem06˙label05˙false-unreach-call.c
Problem06_label11	Problem06˙label11˙false-unreach-call.c
Problem06_label20	Problem06˙label20˙false-unreach-call.c
Problem06_label21	Problem06˙label21˙false-unreach-call.c
Problem06_label24	Problem06˙label24˙false-unreach-call.c
Problem06_label27	Problem06˙label27˙false-unreach-call.c

Subject ID	SV-COMP file name
Problem06_label29	Problem06˙label29˙false-unreach-call.c
Problem06_label48	Problem06˙label48˙false-unreach-call.c
Problem06_label56	Problem06˙label56˙false-unreach-call.c
Problem06_label58	Problem06˙label58˙false-unreach-call.c
Problem06_label59	Problem06˙label59˙false-unreach-call.c
Problem10_label12	Problem10˙label12˙false-unreach-call.c
Problem10_label15	Problem10˙label15˙false-unreach-call.c
Problem10_label24	Problem10˙label24˙false-unreach-call.c
Problem10_label26	Problem10˙label26˙false-unreach-call.c
Problem10_label28	Problem10˙label28˙false-unreach-call.c
Problem10_label29	Problem10˙label29˙false-unreach-call.c
Problem10_label41	Problem10˙label41˙false-unreach-call.c
Problem10_label42	Problem10˙label42˙false-unreach-call.c
Problem10_label47	Problem10˙label47˙false-unreach-call.c
Problem10_label48	Problem10˙label48˙false-unreach-call.c
Problem10_label57	Problem10˙label57˙false-unreach-call.c
Problem10_label58	Problem10˙label58˙false-unreach-call.c
Problem11_label00	Problem11˙label00˙false-unreach-call.c
Problem11_label14	Problem11˙label14˙false-unreach-call.c
Problem11_label15	Problem11˙label15˙false-unreach-call.c
Problem11_label20	Problem11˙label20˙false-unreach-call.c
Problem11_label29	Problem11˙label29˙false-unreach-call.c
Problem11_label31	Problem11˙label31˙false-unreach-call.c
Problem11_label34	Problem11˙label34˙false-unreach-call.c
Problem11_label36	Problem11˙label36˙false-unreach-call.c
Problem11_label39	Problem11˙label39˙false-unreach-call.c
Problem11_label42	Problem11˙label42˙false-unreach-call.c
Problem11_label43	Problem11˙label43˙false-unreach-call.c
Problem11_label49	Problem11˙label49˙false-unreach-call.c
Problem11_label51	Problem11˙label51˙false-unreach-call.c
Problem11_label58	Problem11˙label58˙false-unreach-call.c
Problem13_label04	Problem13˙label04˙false-unreach-call.c
Problem13_label07	Problem13˙label07˙false-unreach-call.c
Problem13_label12	Problem13˙label12˙false-unreach-call.c
Problem13_label16	Problem13˙label16˙false-unreach-call.c
Problem13_label19	Problem13˙label19˙false-unreach-call.c

Subject ID	SV-COMP file name
Problem13_label21	Problem13˙label21˙false-unreach-call.c
Problem13_label24	Problem13˙label24˙false-unreach-call.c
Problem13_label25	Problem13˙label25˙false-unreach-call.c
Problem13_label28	Problem13˙label28˙false-unreach-call.c
Problem13_label29	Problem13˙label29˙false-unreach-call.c
Problem13_label32	Problem13˙label32˙false-unreach-call.c
Problem13_label35	Problem13˙label35˙false-unreach-call.c
Problem13_label40	Problem13˙label40˙false-unreach-call.c
Problem13_label43	Problem13˙label43˙false-unreach-call.c
Problem13_label44	Problem13˙label44˙false-unreach-call.c
Problem13_label45	Problem13˙label45˙false-unreach-call.c
Problem13_label48	Problem13˙label48˙false-unreach-call.c
Problem18_label00	Problem18˙label00˙false-unreach-call.c
Problem18_label01	Problem18˙label01˙false-unreach-call.c
Problem18_label03	Problem18˙label03˙false-unreach-call.c
Problem18_label06	Problem18˙label06˙false-unreach-call.c
Problem18_label08	Problem18˙label08˙false-unreach-call.c
Problem18_label10	Problem18˙label10˙false-unreach-call.c
Problem18_label12	Problem18˙label12˙false-unreach-call.c
Problem18_label19	Problem18˙label19˙false-unreach-call.c
Problem18_label20	Problem18˙label20˙false-unreach-call.c
Problem18_label25	Problem18˙label25˙false-unreach-call.c
Problem18_label33	Problem18˙label33˙false-unreach-call.c
Problem18_label34	Problem18˙label34˙false-unreach-call.c
Problem18_label35	Problem18˙label35˙false-unreach-call.c
Problem18_label36	Problem18˙label36˙false-unreach-call.c
Problem18_label38	Problem18˙label38˙false-unreach-call.c
Problem18_label45	Problem18˙label45˙false-unreach-call.c
Problem18_label52	Problem18˙label52˙false-unreach-call.c
Problem18_label55	Problem18˙label55˙false-unreach-call.c
Problem18_label57	Problem18˙label57˙false-unreach-call.c
rangesum05	rangesum05˙false-unreach-call˙true-termination.i
rangesum10	rangesum10˙false-unreach-call˙true-termination.i
rangesum20	rangesum20˙false-unreach-call.i
recHanoi03	recHanoi03˙false-unreach-call˙true-termination.c
s3_srvr_6	s3˙srvr˙6˙false-unreach-call˙false-termination.cil.c

Subject ID	SV-COMP file name
sine_1	sine˙1˙false-unreach-call˙true-termination.i
sine_2	sine˙2˙false-unreach-call˙true-termination.i
square_1	square˙1˙false-unreach-call˙true-termination.i
square_2	square˙2˙false-unreach-call˙true-termination.i
square_3	square˙3˙false-unreach-call˙true-termination.i
sum01_bug02_sum01_bug02_base.case	sum01˙bug02˙sum01˙bug02˙base.case˙false-unreach-call˙true-termination.i
terminator_01	terminator˙01˙false-unreach-call˙true-termination.i
test_locks_14	test˙locks˙14˙false-unreach-call˙true-valid-memsafety˙false-termination.c
test_locks_15	test˙locks˙15˙false-unreach-call˙true-valid-memsafety˙false-termination.c
token_ring.01	token˙ring.01˙false-unreach-call˙false-termination.cil.c
token_ring.04	token˙ring.04˙false-unreach-call˙false-termination.cil.c
token_ring.08	token˙ring.08˙false-unreach-call˙false-termination.cil.c
token_ring.10	token˙ring.10˙false-unreach-call˙false-termination.cil.c
token_ring.13	token˙ring.13˙false-unreach-call˙false-termination.cil.c
toy1	toy1˙false-unreach-call˙false-termination.cil.c
toy2	toy2˙false-unreach-call˙false-termination.cil.c
transmitter.01	transmitter.01˙false-unreach-call˙false-termination.cil.c
transmitter.02	transmitter.02˙false-unreach-call˙false-termination.cil.c
transmitter.15	transmitter.15˙false-unreach-call˙false-termination.cil.c
transmitter.16	transmitter.16˙false-unreach-call˙false-termination.cil.c

A.2 Detailed results

For each subject, we report the time (in seconds) spent in *characterize*; *generalize*; *analyze*; and the overall (*tot.*) time. We also report on the number of: ACA iterations to convergence; times generalization is invoked; logical intervals with coinciding bounds; logical intervals with noncoinciding bounds; and conjuncts sliced away from reachability conditions expressed as conjunctive formulae.

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`Ackermann02`	3	0	21	25	2	0	1	0	0
`Addition02`	10	23	99	132	6	1	0	1	6
`BallRajamani-SPIN2000-Fig1`	2	0	18	21	2	0	1	0	2
`EvenOdd03`	12	16	83	111	7	1	0	1	0
`Fibonacci04`	3	0	22	25	2	0	1	0	0
`Fibonacci05`	2	0	27	29	2	0	1	0	0
`McCarthy91`	2	0	47	54	2	0	1	0	0
`Problem01_label15`	31	306	1033	1373	6	1	0	1	63
`Problem01_label20`	16	255	634	906	5	1	0	1	44
`Problem01_label21`	24	328	730	1085	5	1	1	1	50
`Problem01_label32`	12	306	456	776	4	1	1	1	21
`Problem01_label33`	17	288	612	918	5	1	0	1	33
`Problem01_label35`	25	306	910	1243	6	1	0	1	63
`Problem01_label37`	15	306	584	906	5	1	0	1	33
`Problem01_label38`	33	307	893	1236	6	1	0	1	63
`Problem01_label44`	30	176	619	825	6	1	0	1	51
`Problem01_label47`	12	306	438	757	4	1	0	1	11
`Problem01_label50`	17	313	617	949	5	1	1	1	50
`Problem01_label56`	17	311	598	927	5	1	0	1	33
`Problem01_label57`	17	306	678	1002	5	1	0	1	43
`Problem02_label13`	29	238	483	752	6	1	0	1	40

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`Problem02_label16`	34	205	751	993	6	1	0	1	55
`Problem02_label43`	28	199	446	674	6	1	0	1	40
`Problem02_label44`	33	156	512	701	6	1	0	1	39
`Problem02_label45`	26	146	670	842	6	1	0	1	45
`Problem02_label50`	29	162	699	893	6	1	0	1	45
`Problem02_label59`	27	258	946	1237	6	1	1	1	69
`Problem03_label09`	31	306	384	724	3	1	0	1	0
`Problem03_label13`	25	306	359	692	3	1	0	1	0
`Problem03_label26`	27	306	343	678	3	1	0	1	0
`Problem03_label27`	47	307	388	750	3	1	0	1	0
`Problem03_label28`	22	306	349	679	3	1	0	1	0
`Problem03_label31`	28	326	384	745	3	1	0	1	0
`Problem03_label35`	18	307	352	680	3	1	0	1	0
`Problem03_label37`	23	222	345	591	3	1	0	1	0
`Problem03_label39`	31	714	379	1128	3	1	0	1	0
`Problem03_label43`	22	307	349	681	3	1	0	1	0
`Problem03_label45`	20	306	353	681	3	1	0	1	0
`Problem03_label50`	24	331	363	726	3	1	0	1	0
`Problem03_label52`	21	5048	326	5400	2	1	0	1	0
`Problem04_label19`	67	4850	478	5400	2	1	0	1	0
`Problem04_label55`	76	4865	454	5400	2	1	0	1	0
`Problem06_label05`	119	307	526	958	3	1	0	1	0
`Problem06_label11`	117	315	517	1023	3	1	0	1	0
`Problem06_label20`	130	306	544	989	3	1	0	1	0
`Problem06_label21`	147	319	575	1049	3	1	0	1	0
`Problem06_label24`	129	306	659	1104	3	1	0	1	0
`Problem06_label27`	124	306	561	1004	3	1	0	1	0
`Problem06_label29`	116	309	482	913	3	1	0	1	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`Problem06_label48`	134	306	579	1029	3	1	0	1	0
`Problem06_label56`	117	307	580	1015	3	1	0	1	0
`Problem06_label58`	135	306	521	967	3	1	0	1	0
`Problem06_label59`	125	307	548	1014	3	1	0	1	0
`Problem10_label12`	229	328	367	928	3	1	0	1	56
`Problem10_label15`	824	397	1120	2341	6	1	0	1	204
`Problem10_label24`	531	524	473	1530	4	1	1	1	132
`Problem10_label26`	265	338	467	1072	3	1	0	1	97
`Problem10_label28`	194	430	475	1099	3	1	0	1	36
`Problem10_label29`	848	563	897	2309	5	1	0	1	252
`Problem10_label41`	470	305	589	1364	4	1	0	1	111
`Problem10_label42`	137	224	330	691	3	1	0	1	16
`Problem10_label47`	252	0	96	350	2	0	1	0	69
`Problem10_label48`	228	268	342	839	3	1	0	1	48
`Problem10_label57`	182	399	338	919	3	1	0	1	30
`Problem10_label58`	243	433	348	1026	3	1	0	1	60
`Problem11_label00`	177	457	360	995	3	1	0	1	54
`Problem11_label14`	322	275	477	1075	4	1	0	1	78
`Problem11_label15`	631	360	834	1826	6	1	0	1	297
`Problem11_label20`	148	396	369	915	3	1	0	1	79
`Problem11_label29`	568	305	917	1792	5	1	0	1	213
`Problem11_label31`	348	340	630	1320	4	1	0	1	126
`Problem11_label34`	349	428	745	1525	4	1	0	1	126
`Problem11_label36`	104	0	152	258	2	0	1	0	25
`Problem11_label39`	149	305	587	1043	3	1	0	1	61
`Problem11_label42`	349	317	506	1173	4	1	0	1	78
`Problem11_label43`	151	305	394	855	3	1	0	1	37
`Problem11_label49`	378	429	585	1393	4	1	0	1	110

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`Problem11_label51`	191	476	354	1022	3	1	0	1	37
`Problem11_label58`	134	325	341	804	3	1	0	1	26
`Problem13_label04`	121	307	409	849	3	1	0	1	0
`Problem13_label07`	130	307	420	864	3	1	0	1	0
`Problem13_label12`	136	617	424	1188	3	1	0	1	0
`Problem13_label16`	128	308	412	869	3	1	0	1	0
`Problem13_label19`	140	306	434	886	3	1	0	1	0
`Problem13_label21`	120	315	399	848	3	1	0	1	0
`Problem13_label24`	112	308	417	860	3	1	0	1	0
`Problem13_label25`	134	308	415	863	3	1	0	1	0
`Problem13_label28`	110	306	373	795	3	1	0	1	0
`Problem13_label29`	118	306	455	888	3	1	0	1	0
`Problem13_label32`	116	307	408	836	3	1	0	1	0
`Problem13_label35`	98	306	371	780	3	1	0	1	0
`Problem13_label40`	111	312	396	829	3	1	0	1	0
`Problem13_label43`	155	307	497	987	3	1	0	1	0
`Problem13_label44`	131	306	389	829	3	1	0	1	0
`Problem13_label45`	115	306	382	811	3	1	0	1	0
`Problem13_label48`	118	306	406	835	3	1	0	1	0
`Problem18_label00`	81	307	385	784	3	1	0	1	0
`Problem18_label01`	80	307	377	768	3	1	0	1	0
`Problem18_label03`	85	307	394	788	3	1	0	1	0
`Problem18_label06`	83	307	384	777	3	1	0	1	0
`Problem18_label08`	88	307	376	778	3	1	0	1	0
`Problem18_label10`	84	307	381	775	3	1	0	1	0
`Problem18_label12`	95	307	433	872	3	1	0	1	0
`Problem18_label19`	85	308	377	790	3	1	0	1	0
`Problem18_label20`	107	307	386	802	3	1	0	1	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`Problem18_label25`	109	307	419	845	3	1	0	1	0
`Problem18_label33`	92	307	390	792	3	1	0	1	0
`Problem18_label34`	94	307	403	836	3	1	0	1	0
`Problem18_label35`	99	323	402	833	3	1	0	1	0
`Problem18_label36`	82	307	384	776	3	1	0	1	0
`Problem18_label38`	90	309	370	794	3	1	0	1	0
`Problem18_label45`	131	307	459	914	3	1	0	1	0
`Problem18_label52`	87	306	365	763	3	1	0	1	0
`Problem18_label55`	84	339	435	866	3	1	0	1	0
`Problem18_label57`	90	308	379	780	3	1	0	1	0
`array`	4	0	32	37	2	0	1	0	0
`cdaudio_simpl1`	29	350	334	715	3	1	0	1	3
`elevator_spec14_productSimulator`	7	306	333	649	3	1	0	1	4
`elevator_spec1_productSimulator`	5	306	330	642	3	1	0	1	3
`elevator_spec2_productSimulator`	3	0	55	59	2	0	1	0	3
`elevator_spec3_productSimulator`	5	306	329	640	3	1	0	1	2
`elevator_spec9_productSimulator`	5	0	61	70	2	0	1	0	4
`email_spec0_product16`	4	0	28	34	2	0	1	0	0
`email_spec0_product21`	3	0	33	38	2	0	1	0	0
`email_spec0_product22`	5	0	28	33	2	0	2	0	0
`email_spec0_product26`	3	917	949	1871	5	3	0	3	0
`email_spec0_product31`	7	612	352	975	3	1	0	1	0
`email_spec0_product33`	3	0	25	29	2	0	1	0	0
`email_spec0_product34`	3	917	941	1863	5	3	0	3	0
`email_spec0_product35`	4	0	29	34	2	0	1	0	0
`email_spec0_productSimulator`	4	612	331	951	3	1	0	1	11
`email_spec11_product15`	13	612	812	1439	7	2	0	2	52
`email_spec11_product20`	4	0	35	41	2	0	1	0	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
email_spec11_product22	13	611	830	1456	7	2	0	2	52
email_spec11_product26	3	0	35	43	2	0	1	0	0
email_spec11_product30	5	611	709	1330	4	2	0	2	0
email_spec11_product32	4	0	29	35	2	0	1	0	0
email_spec11_product33	4	917	942	1869	5	3	0	3	0
email_spec11_product35	8	1223	1004	2236	6	3	0	3	6
email_spec11_productSimulator	3	306	328	639	3	1	0	1	11
email_spec1_product14	4	612	324	941	3	1	0	1	0
email_spec1_product15	6	1191	953	2154	5	3	0	3	0
email_spec1_product16	5	306	327	640	3	1	0	1	0
email_spec1_product20	4	612	326	944	3	1	0	1	0
email_spec1_product21	4	612	332	952	3	1	0	1	0
email_spec1_product22	5	1231	629	1867	4	2	0	2	0
email_spec1_product26	5	612	325	944	3	1	0	1	0
email_spec1_product30	8	306	340	655	3	1	1	1	0
email_spec1_product31	5	306	325	637	3	1	0	1	0
email_spec1_product32	5	619	330	960	3	1	0	1	0
email_spec1_product33	6	611	324	942	3	1	0	1	0
email_spec1_product34	4	306	332	645	3	1	0	1	0
email_spec1_product35	4	1223	1249	2481	6	4	0	4	0
email_spec1_productSimulator	5	612	338	957	3	1	0	1	9
email_spec27_product17	5	0	69	82	2	0	1	0	0
email_spec27_product18	5	938	632	1576	4	2	0	2	0
email_spec27_product19	6	0	80	90	2	0	1	0	0
email_spec27_product23	4	0	34	38	2	0	1	0	0
email_spec27_product24	5	927	633	1566	4	2	0	2	0
email_spec27_product25	7	306	380	700	3	1	0	1	0
email_spec27_product27	4	612	325	943	3	1	0	1	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
email_spec27_product29	5	0	25	31	2	0	1	0	0
email_spec27_product30	5	944	632	1585	4	2	0	2	0
email_spec27_product31	11	0	56	71	2	0	1	0	0
email_spec27_product32	9	0	92	107	2	0	1	0	0
email_spec27_product33	5	1533	633	2172	4	2	0	1	0
email_spec27_product34	4	920	331	1256	3	1	0	1	0
email_spec27_product35	4	0	28	32	2	0	1	0	0
email_spec27_productSimulator	5	612	475	1097	3	1	0	1	9
email_spec3_product13	7	533	650	1196	4	2	1	2	0
email_spec3_product17	4	545	638	1190	4	2	1	2	0
email_spec3_product18	4	544	640	1190	4	2	1	2	0
email_spec3_product19	4	553	632	1190	4	2	1	2	0
email_spec3_product23	28	2173	3194	5400	5	4	0	1	0
email_spec3_product24	5	539	638	1185	4	2	1	2	0
email_spec3_product25	7	1509	939	2472	5	3	0	3	0
email_spec3_product27	3	579	636	1219	4	2	1	2	0
email_spec3_product28	4	1479	935	2421	5	3	0	3	0
email_spec3_product29	4	565	671	1242	4	2	1	2	0
email_spec3_product30	5	2101	1551	3662	7	5	0	5	0
email_spec3_product31	8	543	730	1289	4	2	1	2	0
email_spec3_product32	6	620	761	1392	4	2	1	2	0
email_spec3_product33	4	538	660	1205	4	2	1	2	0
email_spec3_product34	3	614	635	1253	4	2	1	2	0
email_spec3_product35	7	628	643	1281	4	2	1	2	0
email_spec3_productSimulator	4	313	400	722	3	1	0	1	8
email_spec4_product18	4	1221	635	1861	4	2	0	2	0
email_spec4_product19	4	0	75	81	2	0	2	0	0
email_spec4_product23	4	918	629	1552	4	2	0	2	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
email_spec4_product24	4	1226	642	1876	4	2	0	2	0
email_spec4_product25	4	612	328	948	3	1	0	1	0
email_spec4_product27	3	1224	630	1859	4	2	0	1	0
email_spec4_product30	4	0	25	30	2	0	1	0	0
email_spec4_product31	5	0	26	32	2	0	1	0	0
email_spec4_product32	4	0	28	32	2	0	1	0	0
email_spec4_product33	4	1240	745	1992	4	2	0	2	0
email_spec4_product34	3	1166	640	1812	4	2	0	2	0
email_spec4_product35	5	0	29	35	2	0	1	0	0
email_spec4_productSimulator	4	306	478	794	3	1	0	1	9
email_spec6_product12	10	306	456	773	5	1	1	1	27
email_spec6_product14	9	612	730	1357	5	2	0	2	22
email_spec6_product15	9	917	756	1684	6	2	0	2	27
email_spec6_product16	7	306	387	704	4	1	1	1	11
email_spec6_product20	15	918	1099	2033	8	3	0	3	29
email_spec6_product21	7	611	690	1310	5	2	0	2	6
email_spec6_product22	4	306	355	669	3	1	1	1	0
email_spec6_product26	14	917	1142	2078	8	3	0	3	29
email_spec6_product28	7	612	688	1308	5	2	0	2	22
email_spec6_product29	8	918	1048	1979	6	3	0	3	6
email_spec6_product30	12	917	1060	1991	7	3	0	3	17
email_spec6_product31	7	306	604	919	4	1	0	1	16
email_spec6_product32	13	917	1076	2012	7	3	0	3	17
email_spec6_product33	7	1223	1245	2480	6	4	0	4	0
email_spec6_product34	15	918	1121	2058	8	3	0	3	31
email_spec6_product35	12	917	1066	1996	7	3	0	3	17
email_spec6_productSimulator	4	306	329	641	3	1	0	1	9
email_spec7_product28	6	871	650	1531	4	2	1	2	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
email_spec7_product29	8	917	991	1926	5	3	0	3	0
email_spec7_product30	11	918	1062	1991	7	3	0	3	17
email_spec7_product31	8	614	692	1316	5	2	0	2	22
email_spec7_product32	3	345	679	1030	4	2	0	2	0
email_spec7_product33	11	917	1070	2001	7	3	0	3	17
email_spec7_product34	11	612	756	1380	6	2	0	2	35
email_spec7_product35	13	918	1072	2006	7	3	0	3	17
email_spec7_productSimulator	6	306	338	651	3	1	0	1	12
email_spec8_product15	5	286	343	636	3	1	0	1	0
email_spec8_product16	4	308	326	639	3	1	0	1	0
email_spec8_product20	4	306	332	643	3	1	0	1	0
email_spec8_product21	3	306	325	635	3	1	0	1	0
email_spec8_product22	6	310	335	655	3	1	0	1	0
email_spec8_product26	4	612	328	945	3	1	0	1	0
email_spec8_product30	6	306	342	658	3	1	0	1	0
email_spec8_product31	4	331	325	661	3	1	0	1	0
email_spec8_product33	4	936	636	1579	4	2	0	2	0
email_spec8_product34	4	306	325	636	3	1	0	1	0
email_spec8_product35	4	306	328	641	3	1	0	1	0
email_spec8_productSimulator	4	612	337	954	3	1	0	1	8
email_spec9_product15	5	284	327	620	3	1	0	1	0
email_spec9_product16	4	316	328	652	3	1	0	1	0
email_spec9_product20	5	306	327	639	3	1	0	1	0
email_spec9_product21	5	305	327	638	3	1	0	1	0
email_spec9_product22	5	308	330	643	3	1	0	1	0
email_spec9_product26	4	306	445	764	3	1	0	1	0
email_spec9_product30	5	305	325	637	3	1	0	1	0
email_spec9_product31	4	327	327	660	3	1	0	1	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
email_spec9_product32	3	305	328	640	3	1	0	1	0
email_spec9_product33	5	932	661	1604	4	2	0	2	0
email_spec9_product34	5	306	327	639	3	1	0	1	0
email_spec9_product35	4	306	329	641	3	1	0	1	0
floppy_simpl3	32	346	943	1322	6	1	0	1	51
floppy_simpl4	34	306	329	670	3	1	0	1	3
for_bounded_loop1	5	0	42	47	3	0	2	0	5
id_b3_o2	3	0	19	23	2	0	1	0	0
kbfiltr_simpl2	22	0	119	142	4	0	3	0	18
kundu2	22	186	171	382	7	1	0	1	42
minepump_spec1_product33	5	31	331	368	4	1	1	1	5
minepump_spec1_product34	15	324	374	714	7	2	1	2	22
minepump_spec1_product35	7	345	636	990	5	2	0	2	5
minepump_spec1_product36	6	348	635	989	5	2	0	2	6
minepump_spec1_product37	6	30	332	369	4	1	1	1	5
minepump_spec1_product38	17	328	586	939	7	2	1	2	22
minepump_spec1_product39	5	344	634	984	5	2	0	2	5
minepump_spec1_product40	5	346	641	993	5	2	0	2	5
minepump_spec1_product42	4	201	1239	1446	6	4	0	4	20
minepump_spec1_product43	4	785	1541	2332	7	5	0	5	22
minepump_spec1_product44	9	751	1265	2026	8	4	0	4	28
minepump_spec1_product49	5	30	334	370	4	1	1	1	5
minepump_spec1_product50	14	322	378	716	7	2	1	2	22
minepump_spec1_product51	5	346	634	986	5	2	0	2	5
minepump_spec1_product52	6	345	639	991	5	2	0	2	5
minepump_spec1_product53	7	34	396	440	4	1	1	1	5
minepump_spec1_product54	13	322	377	713	7	2	1	2	22
minepump_spec1_product55	6	343	636	986	5	2	0	2	5

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`minepump_spec1_product56`	6	348	638	993	5	2	0	2	5
`minepump_spec1_productSimulator`	8	74	327	412	3	1	0	1	1
`minepump_spec2_product33`	3	34	332	370	3	1	1	1	11
`minepump_spec2_product34`	2	37	326	366	3	1	1	1	5
`minepump_spec2_product35`	4	181	928	1114	5	3	0	3	7
`minepump_spec2_product36`	9	305	355	671	5	1	1	1	17
`minepump_spec2_product41`	4	1129	2762	3897	11	9	0	7	111
`minepump_spec2_product42`	5	240	939	1186	5	3	0	3	18
`minepump_spec2_product43`	3	0	27	31	2	0	1	0	1
`minepump_spec2_product44`	15	1730	2538	4287	14	9	0	9	89
`minepump_spec2_productSimulator`	3	306	324	633	3	1	0	1	3
`minepump_spec3_product01`	8	0	43	52	4	0	1	0	0
`minepump_spec3_product02`	7	86	660	755	6	3	3	2	22
`minepump_spec3_product03`	8	114	972	1095	7	4	0	1	24
`minepump_spec3_product04`	5	199	1846	2052	8	6	0	6	22
`minepump_spec3_product05`	8	0	61	70	4	0	1	0	0
`minepump_spec3_product06`	11	57	371	441	6	2	4	1	23
`minepump_spec3_product07`	5	148	1560	1714	8	5	0	1	27
`minepump_spec3_product08`	3	52	627	683	4	2	0	2	3
`minepump_spec3_product09`	9	0	73	82	5	0	1	0	0
`minepump_spec3_product10`	11	87	667	766	6	3	3	2	22
`minepump_spec3_product11`	5	108	1256	1370	7	4	0	1	24
`minepump_spec3_product12`	3	227	2174	2410	9	7	0	7	25
`minepump_spec3_product13`	8	0	52	61	4	0	1	0	0
`minepump_spec3_product14`	11	83	685	780	7	3	0	1	22
`minepump_spec3_product15`	7	146	1619	1774	9	5	0	1	33
`minepump_spec3_product16`	4	50	623	678	4	2	0	2	3
`minepump_spec3_product17`	11	0	77	89	5	0	1	0	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`minepump_spec3_product18`	7	80	661	748	6	3	3	2	22
`minepump_spec3_product19`	6	127	1343	1488	7	4	0	1	19
`minepump_spec3_product20`	4	48	623	676	4	2	0	2	3
`minepump_spec3_product21`	10	0	67	78	5	0	1	0	0
`minepump_spec3_product23`	8	96	966	1070	7	3	0	1	22
`minepump_spec3_product24`	4	54	626	685	4	2	0	2	3
`minepump_spec3_product25`	8	0	53	62	4	0	1	0	0
`minepump_spec3_product26`	10	58	683	753	7	2	0	1	23
`minepump_spec3_product27`	7	145	1632	1789	8	5	0	1	24
`minepump_spec3_product28`	3	48	625	677	4	2	0	2	3
`minepump_spec3_product29`	7	0	53	60	4	0	1	0	0
`minepump_spec3_product30`	7	83	659	750	6	3	3	2	22
`minepump_spec3_product31`	9	86	979	1075	7	3	0	1	23
`minepump_spec3_product32`	4	52	625	682	4	2	0	2	3
`minepump_spec3_product35`	6	85	952	1045	6	3	1	3	19
`minepump_spec3_product36`	2	72	623	697	4	2	0	2	4
`minepump_spec3_product39`	9	85	969	1064	7	3	0	1	27
`minepump_spec3_product40`	4	77	661	743	4	2	0	2	4
`minepump_spec3_product43`	5	85	947	1038	6	3	1	3	18
`minepump_spec3_product44`	3	69	624	698	4	2	0	2	5
`minepump_spec3_product47`	14	58	407	479	7	2	0	1	28
`minepump_spec3_product48`	3	71	624	699	4	2	0	2	4
`minepump_spec3_product51`	5	305	338	649	4	1	0	1	14
`minepump_spec3_product52`	4	66	624	696	4	2	0	2	4
`minepump_spec3_product55`	7	91	968	1067	7	3	0	1	26
`minepump_spec3_product56`	3	72	621	696	4	2	0	2	4
`minepump_spec3_product59`	14	360	692	1068	7	3	3	2	32
`minepump_spec3_product60`	4	63	653	723	4	2	0	2	6

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
`minepump_spec3_product63`	9	51	354	414	5	2	0	1	22
`minepump_spec3_product64`	4	75	625	704	4	2	0	2	4
`minepump_spec3_productSimulator`	4	89	620	714	4	2	0	2	4
`minepump_spec4_product33`	2	0	20	23	2	0	1	0	3
`minepump_spec4_product34`	2	0	19	22	2	0	1	0	2
`minepump_spec4_product35`	6	214	931	1153	5	3	0	3	4
`minepump_spec4_product36`	13	858	2226	3100	13	8	0	7	75
`minepump_spec4_product37`	8	0	34	43	2	0	1	0	3
`minepump_spec4_product38`	4	817	1544	2367	7	5	0	4	23
`minepump_spec4_product39`	3	0	19	22	2	0	1	0	0
`minepump_spec4_product40`	14	907	1326	2250	10	5	0	5	63
`minepump_spec4_product41`	4	0	21	26	2	0	1	0	3
`minepump_spec4_product42`	2	0	22	25	2	0	1	0	2
`minepump_spec4_product43`	2	230	934	1168	5	3	0	3	4
`minepump_spec4_product44`	4	0	17	22	2	0	1	0	0
`minepump_spec4_product45`	4	1851	2158	4015	9	7	0	1	92
`minepump_spec4_product46`	3	913	1541	2458	7	5	0	5	26
`minepump_spec4_product47`	2	702	1237	1943	6	4	0	4	12
`minepump_spec4_product48`	69	1789	3537	5400	11	5	0	1	0
`minepump_spec4_productSimulator`	3	306	323	632	3	1	0	1	4
`newton_1_4`	2	0	58	60	2	0	1	0	0
`newton_1_5`	2	0	55	58	2	0	1	0	0
`newton_1_6`	3	0	36	39	2	0	1	0	0
`newton_1_7`	2	0	34	37	2	0	1	0	0
`pc_sfifo_1`	13	48	124	187	6	1	1	1	7
`pc_sfifo_2`	11	305	419	737	6	1	0	1	12
`rangesum05`	2	0	107	115	2	0	1	0	0
`rangesum10`	4	0	212	216	3	0	2	0	0

Table A.2: ACA run characteristics by subject

	Time (in sec.)				Count of:				
Subject	*cha.*	*gen.*	*ana.*	*tot.*	Iters	Gens	$\overline{I_i} \equiv \underline{I_i}$	$\overline{I_i} \not\equiv \underline{I_i}$	Sliced
rangesum20	2	305	843	1151	3	1	0	1	0
recHanoi03	38	305	696	1040	6	1	0	1	0
s3_srvr_6	11	305	319	637	3	1	0	1	0
sine_1	2	0	145	152	2	0	1	0	0
sine_2	2	0	316	318	2	0	1	0	0
square_1	2	0	41	44	2	0	1	0	0
square_2	2	0	239	241	2	0	1	0	0
square_3	2	0	349	351	2	0	1	0	0
sum01_bug02_sum01_bug02_base.case	2	8	21	33	3	1	0	1	0
terminator_01	13	139	82	235	7	1	0	1	10
test_locks_14	13	305	390	710	5	1	0	1	46
test_locks_15	14	305	388	708	5	1	0	1	49
token_ring.01	12	97	751	867	6	1	0	1	34
token_ring.04	17	4639	739	5400	4	2	0	1	0
token_ring.08	3	5073	319	5400	2	1	0	1	0
token_ring.10	4	5070	321	5400	2	1	0	1	0
token_ring.13	5	5067	323	5400	2	1	0	1	0
toy1	29	424	718	1174	8	2	0	1	86
toy2	25	429	658	1115	8	2	0	1	86
transmitter.01	14	69	671	761	6	1	0	1	21
transmitter.02	10	80	530	620	5	1	0	1	16
transmitter.15	4	611	633	1248	4	2	0	1	1
transmitter.16	5	611	632	1247	4	2	0	1	1

Appendix B

Conditional Quantitative Analysis Data

This appendix contains results and performance metrics for the different quantitative analyzers and all the 136 subjects of the benchmark extracted from SV-COMP. The mapping between subject IDs and original SV-COMP names is in Section B.1. Detailed results for each subject are in Section B.2.

B.1 Name mapping

For readability reasons, we shortened the filename of some subjects. Here below is a table mapping each name id to the original filename from the SV-COMP benchmark, which can be found at: `https://github.com/sosy-lab/sv-benchmarks`.

Notably, the SV-COMP organizers renamed some of the subjects. However, the original filename is stored in the yml description of the subject, e.g., see "old file name" in `email_spec0_product31.cil.yml`.

The recommended way to access the benchmark files is searching the SV-COMP repository for the filename using the search function of GitHub. File names in the table are clickable (note: Preview for Mac may not support the LaTeXpackage `hyperref`; Acrobat Reader is recommended).

Subject ID	SV-COMP file name
Ackermann02	Ackermann02˙false-unreach-call˙true-no-overflow˙true-termination.c
Fibonacci04	Fibonacci04˙false-unreach-call˙true-no-overflow˙true-termination.c
Fibonacci05	Fibonacci05˙false-unreach-call˙true-no-overflow˙true-termination.c
McCarthy91	McCarthy91˙false-unreach-call˙true-no-overflow˙true-termination.c
Problem01˙label15	Problem01˙label15˙false-unreach-call˙false-termination.c
Problem01˙label32	Problem01˙label32˙false-unreach-call˙false-termination.c
Problem01˙label33	Problem01˙label33˙false-unreach-call˙false-termination.c
Problem01˙label37	Problem01˙label37˙false-unreach-call˙false-termination.c
Problem01˙label44	Problem01˙label44˙false-unreach-call˙false-termination.c
Problem01˙label50	Problem01˙label50˙false-unreach-call˙false-termination.c
Problem01˙label56	Problem01˙label56˙false-unreach-call˙false-termination.c
Problem01˙label57	Problem01˙label57˙false-unreach-call˙false-termination.c
Problem02˙label13	Problem02˙label13˙false-unreach-call˙false-termination.c
Problem02˙label16	Problem02˙label16˙false-unreach-call˙false-termination.c
Problem02˙label44	Problem02˙label44˙false-unreach-call˙false-termination.c
Problem02˙label45	Problem02˙label45˙false-unreach-call˙false-termination.c
Problem02˙label50	Problem02˙label50˙false-unreach-call˙false-termination.c
Problem02˙label59	Problem02˙label59˙false-unreach-call˙false-termination.c
Problem03˙label09	Problem03˙label09˙false-unreach-call.c
Problem03˙label26	Problem03˙label26˙false-unreach-call.c
Problem03˙label28	Problem03˙label28˙false-unreach-call.c
Problem03˙label31	Problem03˙label31˙false-unreach-call.c
Problem03˙label37	Problem03˙label37˙false-unreach-call.c
Problem03˙label39	Problem03˙label39˙false-unreach-call.c
Problem03˙label43	Problem03˙label43˙false-unreach-call.c
Problem03˙label50	Problem03˙label50˙false-unreach-call.c
Problem04˙label19	Problem04˙label19˙false-unreach-call.c
Problem04˙label55	Problem04˙label55˙false-unreach-call.c
Problem06˙label05	Problem06˙label05˙false-unreach-call.c
Problem06˙label11	Problem06˙label11˙false-unreach-call.c
Problem06˙label20	Problem06˙label20˙false-unreach-call.c
Problem06˙label21	Problem06˙label21˙false-unreach-call.c
Problem06˙label24	Problem06˙label24˙false-unreach-call.c
Problem06˙label27	Problem06˙label27˙false-unreach-call.c
Problem06˙label29	Problem06˙label29˙false-unreach-call.c
Problem06˙label48	Problem06˙label48˙false-unreach-call.c

Subject ID	SV-COMP file name
Problem06˙label56	Problem06˙label56˙false-unreach-call.c
Problem06˙label58	Problem06˙label58˙false-unreach-call.c
Problem06˙label59	Problem06˙label59˙false-unreach-call.c
Problem10˙label15	Problem10˙label15˙false-unreach-call.c
Problem10˙label24	Problem10˙label24˙false-unreach-call.c
Problem10˙label28	Problem10˙label28˙false-unreach-call.c
Problem10˙label41	Problem10˙label41˙false-unreach-call.c
Problem10˙label42	Problem10˙label42˙false-unreach-call.c
Problem10˙label47	Problem10˙label47˙false-unreach-call.c
Problem10˙label48	Problem10˙label48˙false-unreach-call.c
Problem10˙label58	Problem10˙label58˙false-unreach-call.c
Problem11˙label00	Problem11˙label00˙false-unreach-call.c
Problem11˙label14	Problem11˙label14˙false-unreach-call.c
Problem11˙label15	Problem11˙label15˙false-unreach-call.c
Problem11˙label20	Problem11˙label20˙false-unreach-call.c
Problem11˙label29	Problem11˙label29˙false-unreach-call.c
Problem11˙label31	Problem11˙label31˙false-unreach-call.c
Problem11˙label34	Problem11˙label34˙false-unreach-call.c
Problem11˙label42	Problem11˙label42˙false-unreach-call.c
Problem11˙label43	Problem11˙label43˙false-unreach-call.c
Problem11˙label49	Problem11˙label49˙false-unreach-call.c
Problem11˙label51	Problem11˙label51˙false-unreach-call.c
Problem11˙label58	Problem11˙label58˙false-unreach-call.c
Problem13˙label04	Problem13˙label04˙false-unreach-call.c
Problem13˙label07	Problem13˙label07˙false-unreach-call.c
Problem13˙label12	Problem13˙label12˙false-unreach-call.c
Problem13˙label16	Problem13˙label16˙false-unreach-call.c
Problem13˙label19	Problem13˙label19˙false-unreach-call.c
Problem13˙label21	Problem13˙label21˙false-unreach-call.c
Problem13˙label24	Problem13˙label24˙false-unreach-call.c
Problem13˙label25	Problem13˙label25˙false-unreach-call.c
Problem13˙label28	Problem13˙label28˙false-unreach-call.c
Problem13˙label29	Problem13˙label29˙false-unreach-call.c
Problem13˙label32	Problem13˙label32˙false-unreach-call.c
Problem13˙label35	Problem13˙label35˙false-unreach-call.c
Problem13˙label40	Problem13˙label40˙false-unreach-call.c

Subject ID	SV-COMP file name
Problem13˙label43	Problem13˙label43˙false-unreach-call.c
Problem13˙label44	Problem13˙label44˙false-unreach-call.c
Problem13˙label45	Problem13˙label45˙false-unreach-call.c
Problem13˙label48	Problem13˙label48˙false-unreach-call.c
Problem18˙label00	Problem18˙label00˙false-unreach-call.c
Problem18˙label01	Problem18˙label01˙false-unreach-call.c
Problem18˙label03	Problem18˙label03˙false-unreach-call.c
Problem18˙label06	Problem18˙label06˙false-unreach-call.c
Problem18˙label08	Problem18˙label08˙false-unreach-call.c
Problem18˙label10	Problem18˙label10˙false-unreach-call.c
Problem18˙label12	Problem18˙label12˙false-unreach-call.c
Problem18˙label19	Problem18˙label19˙false-unreach-call.c
Problem18˙label20	Problem18˙label20˙false-unreach-call.c
Problem18˙label25	Problem18˙label25˙false-unreach-call.c
Problem18˙label33	Problem18˙label33˙false-unreach-call.c
Problem18˙label34	Problem18˙label34˙false-unreach-call.c
Problem18˙label35	Problem18˙label35˙false-unreach-call.c
Problem18˙label36	Problem18˙label36˙false-unreach-call.c
Problem18˙label38	Problem18˙label38˙false-unreach-call.c
Problem18˙label45	Problem18˙label45˙false-unreach-call.c
Problem18˙label52	Problem18˙label52˙false-unreach-call.c
Problem18˙label55	Problem18˙label55˙false-unreach-call.c
Problem18˙label57	Problem18˙label57˙false-unreach-call.c
cdaudio˙simpl1	cdaudio˙simpl1˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
email˙spec0˙product21	email˙spec0˙product21˙false-unreach-call˙true-termination.cil.c
email˙spec0˙product31	email˙spec0˙product31˙false-unreach-call˙true-termination.cil.c
email˙spec1˙product32	email˙spec1˙product32˙false-unreach-call˙true-termination.cil.c
email˙spec1˙product33	email˙spec1˙product33˙false-unreach-call˙true-termination.cil.c
email˙spec1˙product34	email˙spec1˙product34˙false-unreach-call˙true-termination.cil.c
email˙spec1˙product35	email˙spec1˙product35˙false-unreach-call˙true-termination.cil.c
email˙spec1˙productSimulator	email˙spec1˙productSimulator˙false-unreach-call˙true-termination.cil.c
email˙spec27˙product17	email˙spec27˙product17˙false-unreach-call˙true-termination.cil.c
email˙spec27˙product18	email˙spec27˙product18˙false-unreach-call˙true-termination.cil.c
email˙spec27˙product23	email˙spec27˙product23˙false-unreach-call˙true-termination.cil.c
email˙spec27˙product24	email˙spec27˙product24˙false-unreach-call˙true-termination.cil.c
email˙spec27˙product30	email˙spec27˙product30˙false-unreach-call˙true-termination.cil.c

Subject ID	SV-COMP file name
email˙spec27˙product31	email˙spec27˙product31˙false-unreach-call˙true-termination.cil.c
email˙spec27˙product32	email˙spec27˙product32˙false-unreach-call˙true-termination.cil.c
email˙spec3˙product25	email˙spec3˙product25˙false-unreach-call˙true-termination.cil.c
email˙spec3˙product29	email˙spec3˙product29˙false-unreach-call˙true-termination.cil.c
email˙spec4˙product19	email˙spec4˙product19˙false-unreach-call˙true-termination.cil.c
email˙spec8˙product15	email˙spec8˙product15˙false-unreach-call˙true-termination.cil.c
email˙spec8˙product16	email˙spec8˙product16˙false-unreach-call˙true-termination.cil.c
email˙spec8˙product20	email˙spec8˙product20˙false-unreach-call˙true-termination.cil.c
email˙spec8˙product21	email˙spec8˙product21˙false-unreach-call˙true-termination.cil.c
email˙spec8˙product22	email˙spec8˙product22˙false-unreach-call˙true-termination.cil.c
email˙spec8˙product26	email˙spec8˙product26˙false-unreach-call˙true-termination.cil.c
email˙spec9˙product15	email˙spec9˙product15˙false-unreach-call˙true-termination.cil.c
email˙spec9˙product16	email˙spec9˙product16˙false-unreach-call˙true-termination.cil.c
email˙spec9˙product20	email˙spec9˙product20˙false-unreach-call˙true-termination.cil.c
email˙spec9˙product21	email˙spec9˙product21˙false-unreach-call˙true-termination.cil.c
email˙spec9˙product26	email˙spec9˙product26˙false-unreach-call˙true-termination.cil.c
email˙spec9˙product30	email˙spec9˙product30˙false-unreach-call˙true-termination.cil.c
email˙spec9˙product33	email˙spec9˙product33˙false-unreach-call˙true-termination.cil.c
floppy˙simpl3	floppy˙simpl3˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
floppy˙simpl4	floppy˙simpl4˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
id˙b3˙o2	id˙b3˙o2˙false-unreach-call˙true-termination˙true-no-overflow.c
kbfiltr˙simpl2	kbfiltr˙simpl2˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c
nec11	nec11˙false-unreach-call˙false-termination.i
recHanoi03	recHanoi03˙false-unreach-call˙true-termination.c
token˙ring.04	token˙ring.04˙false-unreach-call˙false-termination.cil.c
token˙ring.08	token˙ring.08˙false-unreach-call˙false-termination.cil.c
token˙ring.10	token˙ring.10˙false-unreach-call˙false-termination.cil.c
token˙ring.13	token˙ring.13˙false-unreach-call˙false-termination.cil.c

B.2 Detailed results

Columns. For each implementation, we report the total time (`time`, including both analysis and counting), the counting time (`♯-time`), the guaranteed minimum probability of reaching a target state ($\sharp(\psi)$), the guaranteed minimum probability of not reaching a target state ($\sharp(\neg\psi)$), the number of paths reaching the target state ($p(\psi)$), the number of paths terminating without reaching a target ($p(\neg\psi)$), and the number of grey paths ($p(?)$, i.e., whose execution reached the depth bound before terminating). The values for the number of paths are meaningful only for implementations using PSE or SSE at some stage, i.e., not for $\text{CQA}_{\#}$.

Rows. Each row reports the result of an implementation. `PSE` and `SSE` indicate the results of executing PSE and SSE without conditioning. cqa-pse-interval-n and cqa-sse-interval-n indicate the application of PSE and SSE conditioned within the n-th interval computed by *generate_intervals*, respectively. The size of the interval (as fraction of the input domain) is reported on the left of cqa-pse-interval-n.

Color scheme. Cells are highlighted with according to the following scheme: the time value is highlighted in red in case of timeout (90 minutes), the time value is highlighted in blue in case of OOM (>8Gb for a single tool) or exception raised by some of the tools involved in this case *the results of the highlighted implementation are not reliable and should be ignored*, $p(?)$ is highlighted in pink when greated than zero.

email˙spec27˙product17˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	5144	1	2.3e−187	1 − 2.8e−9	-	-	-
	cqa-pse	5386	240	1.0e−140	1 − 4.7e−10	11	1564	0
	cqa-sse	5386	240	2.3e−187	1 − 4.7e−10	0	377	0
	pse	2001	1577	4.8e−94	1 − 4.8e−94	812	8220	0
	sse	5389	3386	4.8e−94	1 − 4.8e−94	644	7182	0
(4.7e−10)	cqa-pse-interval-1	242	239	0.0	0	0	0	0
	cqa-sse-interval-1	242	239	0.0	0	0	0	0
(4.7e−10)	cqa-pse-interval-2	242	200	2.3e−159	1.0e−28	4	519	0
	cqa-sse-interval-2	242	210	0.0	2.6e−64	0	155	0
(4.7e−10)	cqa-pse-interval-3	242	206	4.9e−150	4.7e−38	4	485	0
	cqa-sse-interval-3	242	218	0.0	3.2e−64	0	137	0
(4.7e−10)	cqa-pse-interval-4	242	219	4.9e−150	4.7e−38	2	324	0
	cqa-sse-interval-4	242	228	0.0	4.8e−55	0	73	0
(4.7e−10)	cqa-pse-interval-5	242	226	1.0e−140	1.0e−28	1	160	0
	cqa-sse-interval-5	242	237	0.0	6.1e−28	0	9	0
(4.7e−10)	cqa-pse-interval-6	242	233	0.0	3.0e−28	0	76	0
	cqa-sse-interval-6	242	239	0.0	4.3e−19	0	3	0

email˙spec27˙product23˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	3873	1	4.8e−122	0	-	-	-
	cqa-pse	5390	1175	4.7e−38	1 − 4.7e−38	2729	14685	0
	cqa-sse	5391	1333	4.7e−38	1 − 4.7e−38	1114	6146	0
	pse	2076	1616	4.7e−38	1 − 4.7e−38	1360	7672	0
	sse	5389	3510	4.7e−38	1 − 4.7e−38	1192	6616	0
(4.7e−10)	cqa-pse-interval-1	1517	1107	2.2e−47	2.2e−47	1119	5498	0
	cqa-sse-interval-1	1517	981	2.2e−47	2.2e−47	298	1753	0
(4.7e−10)	cqa-pse-interval-2	1517	1174	2.2e−47	2.2e−47	687	5300	0
	cqa-sse-interval-2	1517	1199	2.2e−47	2.2e−47	263	1752	0
(4.7e−10)	cqa-pse-interval-3	873	692	2.2e−47	2.2e−47	471	2327	0
	cqa-sse-interval-3	1517	1165	2.2e−47	2.2e−47	362	1732	0
(1 − 1.4e−9)	cqa-pse-interval-4	1111	985	4.7e−38	1.0 − 4.7e−38	452	1560	0
	cqa-sse-interval-4	1518	1332	4.7e−38	1.0 − 4.7e−38	191	909	0

email˙spec4˙product19˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	3836	1	2.2e−47	0	-	-	-
	cqa-pse	5398	1290	2.2e−19	1 − 2.2e−19	1761	13952	0
	cqa-sse	5399	1232	2.2e−19	1 − 2.2e−19	881	5250	0
	pse	1970	1546	2.2e−19	1 − 2.2e−19	821	8211	0
	sse	5389	3407	2.2e−19	1 − 2.2e−19	792	6790	0
(4.7e−10)	cqa-pse-interval-1	1562	1218	1.0e−28	1.0e−28	609	4862	0
	cqa-sse-interval-1	1562	1179	1.0e−28	1.0e−28	300	1757	0
(4.7e−10)	cqa-pse-interval-2	1562	1235	1.0e−28	1.0e−28	577	4579	0
	cqa-sse-interval-2	1562	1158	1.0e−28	1.0e−28	290	1649	0
(4.7e−10)	cqa-pse-interval-3	1562	1289	2.2e−19	2.2e−19	517	3822	0
	cqa-sse-interval-3	1563	1231	2.2e−19	2.2e−19	233	1155	0
(1 − 1.4e−9)	cqa-pse-interval-4	268	221	4.7e−38	1.0 − 4.7e−38	58	689	0
	cqa-sse-interval-4	506	433	4.7e−38	1.0 − 4.7e−38	58	689	0

email˙spec8˙product16˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	3091	1	4.7e−66	1 − 1.9e−9	-	-	-
	cqa-pse	4738	1253	1.0e−28	1 − 1.0e−28	1119	14782	0
	cqa-sse	5388	1596	1.0e−28	1 − 1.0e−28	960	9823	0
	pse	2029	1595	1.0e−28	1 − 1.0e−28	548	8484	0
	sse	5389	3498	1.0e−28	1 − 1.0e−28	548	7491	0
(4.7e−10)	cqa-pse-interval-1	1305	964	4.7e−38	4.7e−38	396	5463	0
	cqa-sse-interval-1	2297	1595	4.7e−38	4.7e−38	343	3592	0
(4.7e−10)	cqa-pse-interval-2	1647	1252	4.7e−38	4.7e−38	389	5714	0
	cqa-sse-interval-2	2297	1536	4.7e−38	4.7e−38	283	2626	0
(4.7e−10)	cqa-pse-interval-3	909	736	4.7e−38	4.7e−38	263	2578	0
	cqa-sse-interval-3	1978	1551	4.7e−38	4.7e−38	263	2578	0
(4.7e−10)	cqa-pse-interval-4	437	371	1.0e−28	1.0e−28	71	1027	0
	cqa-sse-interval-4	928	821	1.0e−28	1.0e−28	71	1027	0

email˙spec9˙product21˙false-unreach-call˙true-termination.cil.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	3163	1	4.8e−122	$1-1.9e-9$	-	-	-
	cqa-pse	4531	984	1.0e−28	$1-1.0e-28$	851	11478	0
	cqa-sse	5385	1499	1.0e−28	$1-1.0e-28$	765	8650	0
	pse	1564	1187	1.0e−28	$1-1.0e-28$	548	8484	0
	sse	5389	3311	1.0e−28	$1-1.0e-28$	548	8223	0
(4.7e−10)	cqa-pse-interval-1	1368	983	4.7e−38	4.7e−38	421	6293	0
	cqa-sse-interval-1	2222	1498	4.7e−38	4.7e−38	335	3465	0
(4.7e−10)	cqa-pse-interval-2	777	573	4.7e−38	4.7e−38	178	3256	0
	cqa-sse-interval-2	1941	1346	4.7e−38	4.7e−38	178	3256	0
(4.7e−10)	cqa-pse-interval-3	394	310	2.2e−75	2.2e−75	90	1311	0
	cqa-sse-interval-3	799	632	2.2e−75	2.2e−75	90	1311	0
(4.7e−10)	cqa-pse-interval-4	295	244	1.0e−28	1.0e−28	162	618	0
	cqa-sse-interval-4	554	470	1.0e−28	1.0e−28	162	618	0

email˙spec27˙product31˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	3898	1	2.3e−159	0	-	-	-
	cqa-pse	5385	1128	1.0e−28	1 − 1.0e−28	974	12731	0
	cqa-sse	5386	1091	1.0e−28	1 − 1.0e−28	574	5517	0
	pse	2105	1641	1.0e−28	1 − 1.0e−28	524	8508	0
	sse	5390	3555	1.0e−28	1 − 1.0e−28	505	7212	0
(4.7e−10)	cqa-pse-interval-1	1304	962	4.7e−38	4.7e−38	361	5499	0
	cqa-sse-interval-1	1488	1090	4.7e−38	4.7e−38	166	1927	0
(4.7e−10)	cqa-pse-interval-2	1487	1127	4.7e−38	4.7e−38	303	5376	0
	cqa-sse-interval-2	1487	1044	4.7e−38	4.7e−38	98	1734	0
(4.7e−10)	cqa-pse-interval-3	327	256	4.7e−38	4.7e−38	155	1000	0
	cqa-sse-interval-3	690	544	4.7e−38	4.7e−38	155	1000	0
(1 − 1.4e−9)	cqa-pse-interval-4	350	285	1.0e−28	1.0 − 1.0e−28	155	856	0
	cqa-sse-interval-4	783	659	1.0e−28	1.0 − 1.0e−28	155	856	0

email˙spec27˙product18˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	3159	1	2.2e−103	1 − 1.9e−9	-	-	-
	cqa-pse	4653	1118	4.7e−38	1 − 4.7e−38	1049	13815	0
	cqa-sse	5397	1618	4.7e−38	1 − 4.7e−38	973	8166	0
	pse	2088	1647	4.7e−38	1 − 4.7e−38	548	8484	0
	sse	5389	3567	4.7e−38	1 − 4.7e−38	548	7295	0
(4.7e−10)	cqa-pse-interval-1	1288	947	2.2e−47	2.2e−47	396	5463	0
	cqa-sse-interval-1	2238	1617	2.2e−47	2.2e−47	363	2846	0
(4.7e−10)	cqa-pse-interval-2	1494	1117	2.2e−47	2.2e−47	389	6025	0
	cqa-sse-interval-2	2238	1505	2.2e−47	2.2e−47	346	2993	0
(4.7e−10)	cqa-pse-interval-3	633	503	2.2e−47	2.2e−47	160	2070	0
	cqa-sse-interval-3	1317	1055	2.2e−47	2.2e−47	160	2070	0
(4.7e−10)	cqa-pse-interval-4	140	115	4.7e−38	4.7e−38	104	257	0
	cqa-sse-interval-4	269	234	4.7e−38	4.7e−38	104	257	0

email˙spec8˙product21˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1689	1	4.8e−122	1 − 1.4e−9	-	-	-
	cqa-pse	3398	1272	1.0e−28	1 − 1.0e−28	585	8975	0
	cqa-sse	5396	2503	1.0e−28	1 − 1.0e−28	552	6625	0
	pse	2014	1559	1.0e−28	1 − 1.0e−28	548	8484	0
	sse	5389	3396	1.0e−28	1 − 1.0e−28	548	7673	0
(4.7e−10)	cqa-pse-interval-1	1709	1271	1.0e−28	1.0e−28	488	7262	0
	cqa-sse-interval-1	3707	2502	1.0e−28	1.0e−28	455	4912	0
(4.7e−10)	cqa-pse-interval-2	192	130	2.2e−47	2.2e−47	50	968	0
	cqa-sse-interval-2	371	267	2.2e−47	2.2e−47	50	968	0
(4.7e−10)	cqa-pse-interval-3	189	137	4.7e−38	4.7e−38	47	745	0
	cqa-sse-interval-3	382	302	4.7e−38	4.7e−38	47	745	0

email˙spec1˙product32˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1521	1	2.2e−47	0	-	-	-
	cqa-pse	3194	1242	1.4e−37	1 − 1.4e−37	1912	7142	0
	cqa-sse	5393	2612	1.4e−37	1 − 1.4e−37	1831	5017	0
	pse	2039	1578	1.4e−37	1 − 1.4e−37	1914	7118	0
	sse	5390	3348	1.4e−37	1 − 1.4e−37	1898	5877	0
(4.7e−10)	cqa-pse-interval-1	1673	1241	4.7e−38	4.7e−38	1678	6072	0
	cqa-sse-interval-1	3872	2611	4.7e−38	4.7e−38	1597	3947	0
(1 − 4.7e−10)	cqa-pse-interval-2	284	208	9.4e−38	1.0 − 9.4e−38	234	1070	0
	cqa-sse-interval-2	577	428	9.4e−38	1.0 − 9.4e−38	234	1070	0

email˙spec1˙product33˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1901	1	1.0e−84	0	-	-	-
	cqa-pse	4007	1593	1.0e−28	1 − 1.0e−28	2230	11840	0
	cqa-sse	5399	2508	1.0e−28	1 − 1.0e−28	1948	7322	0
	pse	2081	1623	1.0e−28	1 − 1.0e−28	1368	7664	0
	sse	5389	3541	1.0e−28	1 − 1.0e−28	1367	6418	0
(4.7e−10)	cqa-pse-interval-1	1207	882	4.7e−38	4.7e−38	965	4895	0
	cqa-sse-interval-1	3498	2507	4.7e−38	4.7e−38	965	4305	0
(1 − 4.7e−10)	cqa-pse-interval-2	2106	1592	1.0e−28	1.0 − 1.0e−28	1265	6945	0
	cqa-sse-interval-2	3498	2363	1.0e−28	1.0 − 1.0e−28	983	3017	0

email˙spec27˙product24˙false-unreach-call˙true-termination.cil.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	2137	1	2.3e−159	0	-	-	-
	cqa-pse	4080	1459	1.0e−28	$1 - 1.0e{-}28$	1581	12490	0
	cqa-sse	5395	2237	1.0e−28	$1 - 1.0e{-}28$	1332	7621	0
	pse	2028	1575	1.0e−28	$1 - 1.0e{-}28$	970	8062	0
	sse	5388	3560	1.0e−28	$1 - 1.0e{-}28$	950	6737	0
(4.7e−10)	cqa-pse-interval-1	1394	1051	4.7e−38	4.7e−38	681	5179	0
	cqa-sse-interval-1	3258	2236	4.7e−38	4.7e−38	649	4062	0
(1 − 4.7e−10)	cqa-pse-interval-2	1943	1458	1.0e−28	$1.0 - 1.0e{-}28$	900	7311	0
	cqa-sse-interval-2	3257	2137	1.0e−28	$1.0 - 1.0e{-}28$	683	3559	0

email˙spec0˙product31˙false-unreach-call˙true-termination.cil.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1331	1	1.0e−56	$1 - 9.3e{-}10$	-	-	-
	cqa-pse	2804	1034	2.2e−19	$1 - 2.2e{-}19$	371	8534	0
	cqa-sse	5399	2899	2.2e−19	$1 - 2.2e{-}19$	365	5177	0
	pse	2095	1649	2.2e−19	$1 - 2.2e{-}19$	372	8660	0
	sse	5390	3506	2.2e−19	$1 - 2.2e{-}19$	372	8015	0
(4.7e−10)	cqa-pse-interval-1	1473	1033	1.0e−28	1.0e−28	356	7773	0
	cqa-sse-interval-1	4068	2898	1.0e−28	1.0e−28	350	4416	0
(4.7e−10)	cqa-pse-interval-2	131	85	2.2e−19	2.2e−19	15	761	0
	cqa-sse-interval-2	298	234	2.2e−19	2.2e−19	15	761	0

email˙spec1˙product35˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1510	1	4.7e−38	0	-	-	-
	cqa-pse	2883	976	1.0e−28	1 − 1.0e−28	2058	7092	0
	cqa-sse	5398	2750	1.0e−28	1 − 1.0e−28	1853	4430	0
	pse	2061	1602	1.0e−28	1 − 1.0e−28	2036	6996	0
	sse	5389	3472	1.0e−28	1 − 1.0e−28	2010	5588	0
(4.7e−10)	cqa-pse-interval-1	1373	975	1.1e−46	1.1e−46	1719	5841	0
	cqa-sse-interval-1	3888	2749	1.1e−46	1.1e−46	1514	3179	0
(1 − 4.7e−10)	cqa-pse-interval-2	386	280	1.0e−28	1.0 − 1.0e−28	339	1251	0
	cqa-sse-interval-2	752	571	1.0e−28	1.0 − 1.0e−28	339	1251	0

email˙spec27˙product30˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1530	1	4.8e−122	1 − 9.3e−10	-	-	-
	cqa-pse	3274	1269	4.7e−38	1 − 4.7e−38	907	12444	0
	cqa-sse	5388	2740	4.7e−38	1 − 4.7e−38	894	9371	0
	pse	1993	1558	4.7e−38	1 − 4.7e−38	548	8484	0
	sse	5389	3445	4.7e−38	1 − 4.7e−38	548	7686	0
(4.7e−10)	cqa-pse-interval-1	1237	915	2.2e−47	2.2e−47	397	5463	0
	cqa-sse-interval-1	3858	2739	2.2e−47	2.2e−47	397	4900	0
(4.7e−10)	cqa-pse-interval-2	1744	1268	4.7e−38	4.7e−38	510	6981	0
	cqa-sse-interval-2	3858	2585	4.7e−38	4.7e−38	497	4471	0

email˙spec9˙product16˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1982	1	4.8e−122	1 − 9.3e−10	-	-	-
	cqa-pse	3756	1313	1.0e−28	1 − 1.0e−28	905	12263	0
	cqa-sse	5390	2420	1.0e−28	1 − 1.0e−28	803	7930	0
	pse	1826	1422	1.0e−28	1 − 1.0e−28	548	8484	0
	sse	5389	3398	1.0e−28	1 − 1.0e−28	548	8063	0
(4.7e−10)	cqa-pse-interval-1	1296	950	4.7e−38	4.7e−38	396	5463	0
	cqa-sse-interval-1	3408	2419	4.7e−38	4.7e−38	374	4105	0
(4.7e−10)	cqa-pse-interval-2	1774	1312	1.0e−28	1.0e−28	509	6800	0
	cqa-sse-interval-2	3408	2310	1.0e−28	1.0e−28	429	3825	0

floppy˙simpl3˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	720	1	1.0e−56	1 − 4.7e−10	-	-	-
	cqa-pse	4715	3973	1.9e−37	1 − 1.9e−37	95	780	0
	cqa-sse	5388	4644	1.9e−37	1 − 1.9e−37	29	237	0
	pse	391	368	1.9e−37	1 − 1.9e−37	96	782	0
	sse	1000	932	1.9e−37	1 − 1.9e−37	96	782	0
(4.7e−10)	cqa-pse-interval-1	3995	3972	1.9e−37	1.9e−37	95	780	0
	cqa-sse-interval-1	4668	4643	1.9e−37	1.9e−37	29	237	0

email˙spec3˙product29˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	5400	1	2.2e−103	0	-	-	-
	cqa-pse	5400	1	2.2e−103	0	0	0	0
	cqa-sse	5400	1	2.2e−103	0	0	0	0
	pse	1994	1565	4.7e−10	1 − 4.7e−10	7203	1829	0
	sse	5389	3493	4.7e−10	1 − 4.7e−10	6224	1766	0
(1 − 2.2e−103)	cqa-pse-interval-1	0	0	0.0	0	0	0	0
	cqa-sse-interval-1	0	0	0.0	0	0	0	0

email˙spec3˙product25˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	5400	1	4.8e−94	0	-	-	-
	cqa-pse	5400	1	4.8e−94	0	0	0	0
	cqa-sse	5400	1	4.8e−94	0	0	0	0
	pse	1985	1559	4.7e−10	1 − 4.7e−10	7721	1311	0
	sse	5389	3432	4.7e−10	1 − 4.7e−10	6821	1262	0
(1 − 4.8e−94)	cqa-pse-interval-1	0	0	0.0	0	0	0	0
	cqa-sse-interval-1	0	0	0.0	0	0	0	0

Problem13˙label43˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1607	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	5396	3779	1.1e−46	1 − 1.1e−18	0	28	16
	cqa-sse	5396	3780	1.1e−46	1 − 7.6e−27	0	8	0
	pse	19	2	0.0	1 − 1.1e−18	0	39	25
	sse	57	29	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	3789	3778	0.0	1.1e−18	0	28	16
	cqa-sse-interval-1	3789	3779	0.0	7.6e−27	0	8	0

Problem13_label19_false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	2110	1	1.1e−46	$1 - 2.3e{-9}$	-	-	-
	cqa-pse	2539	419	1.1e−46	$1 - 1.1e{-18}$	0	28	16
	cqa-sse	2494	374	1.1e−46	$1 - 7.6e{-27}$	0	8	0
	pse	24	3	0.0	$1 - 1.1e{-18}$	0	39	25
	sse	50	23	0.0	$1 - 1.1e{-18}$	0	39	25
(2.3e−9)	cqa-pse-interval-1	429	418	0.0	1.1e−18	0	28	16
	cqa-sse-interval-1	384	373	0.0	7.6e−27	0	8	0

email_spec1_productSimulator_false-unreach-call_true-termination.cil.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1676	1	2.3e−159	$1 - 1.0$	-	-	-
	cqa-pse	5398	3720	2.3e−159	$1 - 1.0$	0	0	0
	cqa-sse	2083	405	2.3e−159	$1 - 1.0$	0	0	0
	pse	5390	4179	2.2e−47	$1 - 1.0$	3364	16341	0
	sse	5389	2872	1.0e−28	$1 - 1.0e{-28}$	976	3851	0
$(1 - 4.7e{-10})$	cqa-pse-interval-1	3722	3719	0.0	0	0	0	0
	cqa-sse-interval-1	407	404	0.0	0	0	0	0

Problem04_label19_false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	3296	1	3.0e−177	0	-	-	-
	cqa-pse	3739	435	3.0e−177	0	0	0	0
	cqa-sse	3694	390	3.0e−177	0	0	0	0
	pse	36	8	0.0	$1 - 1.3e{-18}$	0	117	101
	sse	110	69	0.0	$1 - 1.3e{-18}$	0	117	101
$(1 - 3.0e{-177})$	cqa-pse-interval-1	443	434	0.0	0	0	0	0
	cqa-sse-interval-1	398	389	0.0	0	0	0	0

floppy˙simpl4˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	939	1	1.0e−56	1 − 4.7e−10	-	-	-
	cqa-pse	5398	4453	1.0e−56	1 − 4.7e−10	0	26	0
	cqa-sse	5398	4452	1.0e−56	1 − 2.1e−36	0	25	0
	pse	5389	1394	0.0	1 − 1.0	0	97	0
	sse	5388	2772	0.0	1 − 1.2e−27	0	17	0
(4.7e−10)	cqa-pse-interval-1	4459	4452	0.0	4.7e−10	0	26	0
	cqa-sse-interval-1	4459	4451	0.0	2.1e−36	0	25	0

recHanoi03˙false-unreach-call˙true-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	5400	1	3.3e−9	0	-	-	-
	cqa-pse	5400	1	3.3e−9	0	0	0	0
	cqa-sse	5400	1	3.3e−9	0	0	0	0
	pse	0	0	0.0	0	0	0	0
	sse	0	0	0.0	0	0	0	0
(1 − 3.3e−9)	cqa-pse-interval-1	0	0	0.0	0	0	0	0
	cqa-sse-interval-1	0	0	0.0	0	0	0	0

Problem03˙label37˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	530	1	1.3e−46	1 − 2.8e−9	-	-	-
	cqa-pse	5395	4600	1.3e−46	1 − 6.5e−18	313	25875	32334
	cqa-sse	5394	3585	1.3e−46	1 − 1.3e−46	118	4921	1
	pse	5392	4826	2.9e−93	1 − 2.3e−9	654	54801	65979
	sse	5394	3021	1.3e−46	1 − 1.3e−46	145	5875	0
(2.8e−9)	cqa-pse-interval-1	4865	4599	5.8e−93	6.5e−18	313	25875	32334
	cqa-sse-interval-1	4864	3584	3.7e−55	3.7e−55	118	4921	1

Problem06_label29_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1362	1	6.1e−56	$1-2.8e{-9}$	-	-	-
	cqa-pse	1389	7	6.1e−56	$1-1.4e{-26}$	0	67	76
	cqa-sse	1408	22	6.1e−56	$1-1.4e{-26}$	0	67	76
	pse	58	8	0.0	$1-5.0e{-18}$	0	91	91
	sse	111	51	0.0	$1-5.0e{-18}$	0	91	91
(2.8e−9)	cqa-pse-interval-1	27	6	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	46	21	0.0	1.4e−26	0	67	76

Problem10_label58_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1102	1	2.4e−37	$1-2.3e{-9}$	-	-	-
	cqa-pse	1117	9	2.4e−37	$1-1.3e{-35}$	0	122	188
	cqa-sse	1174	58	2.4e−37	$1-1.3e{-35}$	0	122	188
	pse	17	10	0.0	$1-9.1e{-28}$	0	214	212
	sse	245	215	0.0	$1-9.1e{-28}$	0	214	212
(2.3e−9)	cqa-pse-interval-1	15	8	0.0	1.3e−35	0	122	188
	cqa-sse-interval-1	72	57	0.0	1.3e−35	0	122	188

Problem11_label14_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1032	1	6.1e−28	$1-2.8e{-9}$	-	-	-
	cqa-pse	1117	68	6.1e−28	$1-2.9e{-27}$	23	675	1108
	cqa-sse	1595	419	6.1e−28	$1-2.9e{-27}$	23	675	1108
	pse	73	53	6.1e−28	$1-2.0e{-27}$	25	886	1417
	sse	1369	1111	6.1e−28	$1-2.0e{-27}$	25	886	1416
(2.8e−9)	cqa-pse-interval-1	85	67	3.1e−36	2.3e−27	23	675	1108
	cqa-sse-interval-1	563	418	3.1e−36	2.3e−27	23	675	1108

Problem10_label42_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1175	1	5.0e−28	1 − 2.3e−9	-	-	-
	cqa-pse	1225	43	5.0e−28	1 − 1.9e−27	7	159	187
	cqa-sse	1286	96	5.0e−28	1 − 1.9e−27	7	159	187
	pse	26	16	5.0e−28	1 − 1.4e−27	9	205	212
	sse	208	176	5.0e−28	1 − 1.4e−27	9	205	212
(2.3e−9)	cqa-pse-interval-1	50	42	7.1e−37	1.4e−27	7	159	187
	cqa-sse-interval-1	111	95	7.1e−37	1.4e−27	7	159	187

email_spec1_product34_false-unreach-call_true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	964	1	1.0e−140	0	-	-	-
	cqa-pse	2768	1283	1.0e−28	1 − 1.0e−28	1317	7714	0
	cqa-sse	5392	3020	1.0e−28	1 − 1.0e−28	1101	4470	0
	pse	2095	1620	1.0e−28	1 − 1.0e−28	1318	7714	0
	sse	5389	3507	1.0e−28	1 − 1.0e−28	1300	6438	0
(1 − 1.0e−140)	cqa-pse-interval-1	1804	1282	1.0e−28	1.0 − 1.0e−28	1317	7714	0
	cqa-sse-interval-1	4428	3019	1.0e−28	1.0 − 1.0e−28	1101	4470	0

Problem18_label10_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1289	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1297	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1303	8	1.3e−74	1 − 6.1e−28	0	20	3
	pse	12	2	0.0	1 − 2.2e−19	0	22	5
	sse	23	10	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	8	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	14	7	0.0	6.1e−28	0	20	3

Problem03`label39`false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1249	1	$6.1e{-56}$	$1-2.8e{-9}$	-	-	-
	cqa-pse	1896	584	$6.1e{-56}$	$1-6.5e{-18}$	0	6553	8233
	cqa-sse	1821	469	$6.1e{-56}$	$1-6.1e{-56}$	0	1359	0
	pse	5393	4858	0.0	$1-2.3e{-9}$	0	52853	62997
	sse	5394	3175	$6.1e{-56}$	$1-6.1e{-56}$	1	5938	0
$(2.8e{-9})$	cqa-pse-interval-1	647	583	0.0	$6.5e{-18}$	0	6553	8233
	cqa-sse-interval-1	572	468	0.0	$9.3e{-63}$	0	1359	0

email`spec8`product15`false-unreach-call`true-termination.cil.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	490	1	$1.0e{-84}$	$1-4.7e{-10}$	-	-	-
	cqa-pse	2202	1263	$4.7e{-38}$	$1-4.7e{-38}$	397	7507	0
	cqa-sse	5398	3513	$4.7e{-38}$	$1-4.7e{-38}$	397	5552	0
	pse	1996	1568	$4.7e{-38}$	$1-4.7e{-38}$	398	8634	0
	sse	5389	3484	$4.7e{-38}$	$1-4.7e{-38}$	398	7578	0
$(4.7e{-10})$	cqa-pse-interval-1	1712	1262	$4.7e{-38}$	$4.7e{-38}$	397	7507	0
	cqa-sse-interval-1	4908	3512	$4.7e{-38}$	$4.7e{-38}$	397	5552	0

Problem04`label55`false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	3201	2	$1.4e{-158}$	0	-	-	-
	cqa-pse	3869	661	$1.4e{-158}$	0	0	0	0
	cqa-sse	3793	586	$1.4e{-158}$	0	0	0	0
	pse	35	8	0.0	$1-1.3e{-18}$	0	117	101
	sse	104	64	0.0	$1-1.3e{-18}$	0	117	101
$(1-1.4e{-158})$	cqa-pse-interval-1	668	659	0.0	0	0	0	0
	cqa-sse-interval-1	592	584	0.0	0	0	0	0

email˙spec9˙product33˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	526	1	1.0e−84	1 − 4.7e−10	-	-	-
	cqa-pse	2257	1268	1.0e−28	1 − 1.0e−28	819	7209	0
	cqa-sse	5389	3444	1.0e−28	1 − 1.0e−28	799	5056	0
	pse	1361	1012	1.0e−28	1 − 1.0e−28	820	8212	0
	sse	4512	2650	1.0e−28	1 − 1.0e−28	820	8212	0
(4.7e−10)	cqa-pse-interval-1	1731	1267	1.0e−28	1.0e−28	819	7209	0
	cqa-sse-interval-1	4863	3443	1.0e−28	1.0e−28	799	5056	0

Problem11˙label49˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1388	1	6.1e−28	1 − 2.8e−9	-	-	-
	cqa-pse	1481	61	6.1e−28	1 − 2.9e−27	1	697	1108
	cqa-sse	1909	381	6.1e−28	1 − 2.9e−27	1	697	1108
	pse	73	52	6.1e−28	1 − 2.0e−27	2	908	1417
	sse	1289	1031	6.1e−28	1 − 2.0e−27	2	908	1417
(2.8e−9)	cqa-pse-interval-1	93	60	1.3e−46	2.3e−27	1	697	1108
	cqa-sse-interval-1	521	380	1.3e−46	2.3e−27	1	697	1108

email˙spec27˙product32˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	634	1	2.2e−47	0	-	-	-
	cqa-pse	4656	3416	4.7e−38	1 − 4.7e−38	1358	7672	0
	cqa-sse	5392	3056	4.7e−38	1 − 4.7e−38	661	3642	0
	pse	2011	1574	4.7e−38	1 − 4.7e−38	1360	7672	0
	sse	5389	3433	4.7e−38	1 − 4.7e−38	1195	6676	0
(1 − 2.2e−47)	cqa-pse-interval-1	4022	3415	4.7e−38	1.0 − 4.7e−38	1358	7672	0
	cqa-sse-interval-1	4758	3055	4.7e−38	1.0 − 4.7e−38	661	3642	0

Problem18˙label36˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1320	1	6.2e−84	1 − 4.7e−10	-	-	-
	cqa-pse	1850	525	6.2e−84	1 − 4.7e−10	0	0	0
	cqa-sse	1790	465	6.2e−84	1 − 4.7e−10	0	0	0
	pse	12	2	0.0	1 − 2.2e−19	0	22	5
	sse	23	9	0.0	1 − 2.2e−19	0	22	5
(4.7e−10)	cqa-pse-interval-1	530	524	0.0	0	0	0	0
	cqa-sse-interval-1	470	464	0.0	0	0	0	0

Problem18˙label12˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1264	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1273	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1280	9	1.3e−74	1 − 6.1e−28	0	20	3
	pse	12	2	0.0	1 − 2.2e−19	0	22	5
	sse	22	9	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	9	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	16	8	0.0	6.1e−28	0	20	3

Problem11˙label58˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	960	1	6.1e−28	1 − 2.8e−9	-	-	-
	cqa-pse	1020	46	6.1e−28	1 − 2.9e−27	11	687	1108
	cqa-sse	1521	430	6.1e−28	1 − 2.9e−27	11	687	1108
	pse	94	72	6.1e−28	1 − 2.0e−27	13	897	1417
	sse	1295	996	6.1e−28	1 − 2.0e−27	13	897	1417
(2.8e−9)	cqa-pse-interval-1	60	45	5.6e−37	2.3e−27	11	687	1108
	cqa-sse-interval-1	561	429	5.6e−37	2.3e−27	11	687	1108

Problem13˙label32˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1312	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1326	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1337	11	1.1e−46	1 − 3.0e−27	0	33	22
	pse	21	3	0.0	1 − 1.1e−18	0	39	25
	sse	58	30	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	14	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	25	10	0.0	3.0e−27	0	33	22

Problem06˙label59˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1431	2	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	1464	10	6.1e−56	1 − 1.4e−26	0	67	76
	cqa-sse	1489	31	6.1e−56	1 − 1.4e−26	0	67	76
	pse	58	8	0.0	1 − 5.0e−18	0	91	91
	sse	128	66	0.0	1 − 5.0e−18	0	91	91
(2.8e−9)	cqa-pse-interval-1	33	8	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	58	29	0.0	1.4e−26	0	67	76

token˙ring.13˙false-unreach-call˙false-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	2650	1	2.2e−19	0	-	-	-
	cqa-pse	5397	2745	2.2e−19	0	0	0	0
	cqa-sse	5397	2745	2.2e−19	0	0	0	0
	pse	5392	1206	0.0	1 − 4.7e−9	0	5630	45080
	sse	5391	2935	0.0	1 − 4.7e−10	0	1105	390
(1 − 2.2e−19)	cqa-pse-interval-1	2747	2744	0.0	0	0	0	0
	cqa-sse-interval-1	2747	2744	0.0	0	0	0	0

Problem11˙label00˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1371	1	2.8e−37	1 − 2.8e−9	-	-	-
	cqa-pse	1471	65	2.8e−37	1 − 2.5e−35	1	581	1063
	cqa-sse	2055	536	2.8e−37	1 − 2.5e−35	1	581	1063
	pse	113	90	2.8e−37	1 − 1.4e−27	2	908	1417
	sse	1256	996	2.8e−37	1 − 1.4e−27	2	908	1417
(2.8e−9)	cqa-pse-interval-1	100	64	1.3e−46	2.5e−35	1	581	1063
	cqa-sse-interval-1	684	535	1.3e−46	2.5e−35	1	581	1063

Problem18˙label52˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1259	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1267	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1274	8	1.3e−74	1 − 6.1e−28	0	20	3
	pse	13	2	0.0	1 − 2.2e−19	0	22	5
	sse	25	10	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	8	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	15	7	0.0	6.1e−28	0	20	3

Problem10˙label48˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	834	1	2.4e−37	1 − 2.3e−9	-	-	-
	cqa-pse	847	8	2.4e−37	1 − 1.3e−35	1	120	189
	cqa-sse	927	74	2.4e−37	1 − 1.3e−35	1	120	189
	pse	26	17	2.4e−37	1 − 9.1e−28	2	212	212
	sse	214	180	2.4e−37	1 − 9.1e−28	2	212	212
(2.3e−9)	cqa-pse-interval-1	13	7	1.1e−46	1.3e−35	1	120	189
	cqa-sse-interval-1	93	73	1.1e−46	1.3e−35	1	120	189

Problem13˙label16˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1318	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1331	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1338	8	1.1e−46	1 − 3.0e−27	0	33	22
	pse	19	2	0.0	1 − 1.1e−18	0	39	25
	sse	55	28	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	13	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	20	7	0.0	3.0e−27	0	33	22

Problem06˙label05˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1370	1	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	1402	9	6.1e−56	1 − 1.4e−26	0	67	76
	cqa-sse	1432	30	6.1e−56	1 − 1.4e−26	0	67	76
	pse	44	6	0.0	1 − 5.0e−18	0	91	91
	sse	129	66	0.0	1 − 5.0e−18	0	91	91
(2.8e−9)	cqa-pse-interval-1	32	8	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	62	29	0.0	1.4e−26	0	67	76

email˙spec9˙product26˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	557	1	1.0e−84	1 − 4.7e−10	-	-	-
	cqa-pse	2100	1126	1.0e−28	1 − 1.0e−28	819	7209	0
	cqa-sse	5391	3401	1.0e−28	1 − 1.0e−28	792	4903	0
	pse	1650	1252	1.0e−28	1 − 1.0e−28	820	8212	0
	sse	4817	2813	1.0e−28	1 − 1.0e−28	820	8212	0
(4.7e−10)	cqa-pse-interval-1	1543	1125	1.0e−28	1.0e−28	819	7209	0
	cqa-sse-interval-1	4834	3400	1.0e−28	1.0e−28	792	4903	0

Problem18˙label34˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1448	1	1.3e−74	1 − 4.7e−10	-	-	-
	cqa-pse	2120	666	1.3e−74	1 − 2.2e−19	0	9	3
	cqa-sse	2713	1260	1.3e−74	1 − 4.7e−10	0	0	0
	pse	9	1	0.0	1 − 2.2e−19	0	22	5
	sse	26	10	0.0	1 − 2.2e−19	0	22	5
(4.7e−10)	cqa-pse-interval-1	672	665	0.0	2.2e−19	0	9	3
	cqa-sse-interval-1	1265	1259	0.0	0	0	0	0

Problem03˙label26˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1168	1	2.8e−37	1 − 2.8e−9	-	-	-
	cqa-pse	1680	458	2.8e−37	1 − 6.5e−18	0	5788	7176
	cqa-sse	5386	3160	2.8e−37	1 − 2.8e−37	0	4688	1
	pse	5392	4831	0.0	1 − 2.3e−9	0	54286	64679
	sse	5394	3237	2.8e−37	1 − 2.8e−37	1	5798	0
(2.8e−9)	cqa-pse-interval-1	512	457	0.0	6.5e−18	0	5788	7176
	cqa-sse-interval-1	4218	3159	0.0	3.0e−71	0	4688	1

cdaudio˙simpl1˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	925	1	4.7e−66	1 − 4.7e−10	-	-	-
	cqa-pse	1046	112	2.2e−47	1 − 2.2e−47	10	355	0
	cqa-sse	1198	252	2.2e−47	1 − 2.2e−47	10	355	0
	pse	463	439	2.2e−47	1 − 2.2e−47	12	1199	0
	sse	1346	1246	2.2e−47	1 − 2.2e−47	12	1199	0
(4.7e−10)	cqa-pse-interval-1	121	111	2.2e−47	2.2e−47	10	355	0
	cqa-sse-interval-1	273	251	2.2e−47	2.2e−47	10	355	0

Problem18˙label33˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1303	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1313	4	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1321	9	1.3e−74	1 − 6.1e−28	0	20	3
	pse	9	1	0.0	1 − 2.2e−19	0	22	5
	sse	26	11	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	10	3	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	18	8	0.0	6.1e−28	0	20	3

Problem03˙label28˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1175	1	1.3e−46	1 − 2.8e−9	-	-	-
	cqa-pse	5394	4003	1.3e−46	1 − 6.5e−18	0	23804	29294
	cqa-sse	5394	3298	1.3e−46	1 − 1.3e−46	0	4219	0
	pse	5392	4734	0.0	1 − 2.3e−9	0	67304	80025
	sse	5394	3192	1.3e−46	1 − 1.3e−46	1	5766	0
(2.8e−9)	cqa-pse-interval-1	4219	4002	0.0	6.5e−18	0	23804	29294
	cqa-sse-interval-1	4219	3297	0.0	3.1e−71	0	4219	0

Problem02˙label44˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	738	1	5.0e−28	1 − 2.3e−9	-	-	-
	cqa-pse	5390	4663	5.0e−28	1 − 5.4e−18	0	1	0
	cqa-sse	5391	4651	5.0e−28	1 − 5.4e−18	0	1	0
	pse	5389	4742	0.0	1 − 1.9e−9	0	15608	20176
	sse	5388	3597	5.0e−28	1 − 5.0e−28	22	6795	0
(2.3e−9)	cqa-pse-interval-1	4652	4662	0.0	5.4e−18	0	1	0
	cqa-sse-interval-1	4653	4650	0.0	5.4e−18	0	1	0

Problem18˙label01˙false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1296	1	6.2e−84	$1-4.7e{-}10$	-	-	-
	cqa-pse	1825	525	6.2e−84	$1-4.7e{-}10$	0	0	0
	cqa-sse	1782	480	6.2e−84	$1-4.7e{-}10$	0	0	0
	pse	10	1	0.0	$1-2.2e{-}19$	0	22	5
	sse	27	12	0.0	$1-2.2e{-}19$	0	22	5
(4.7e−10)	cqa-pse-interval-1	529	524	0.0	0	0	0	0
	cqa-sse-interval-1	486	479	0.0	0	0	0	0

Problem18˙label57˙false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1248	1	1.3e−74	$1-2.8e{-}9$	-	-	-
	cqa-pse	1256	3	1.3e−74	$1-6.1e{-}28$	0	20	3
	cqa-sse	1262	8	1.3e−74	$1-6.1e{-}28$	0	20	3
	pse	9	1	0.0	$1-2.2e{-}19$	0	22	5
	sse	24	9	0.0	$1-2.2e{-}19$	0	22	5
(2.8e−9)	cqa-pse-interval-1	8	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	14	7	0.0	6.1e−28	0	20	3

Problem06˙label56˙false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1382	1	6.1e−56	$1-2.8e{-}9$	-	-	-
	cqa-pse	1416	9	6.1e−56	$1-1.4e{-}26$	0	67	76
	cqa-sse	1441	30	6.1e−56	$1-1.4e{-}26$	0	67	76
	pse	52	6	0.0	$1-5.0e{-}18$	0	91	91
	sse	147	82	0.0	$1-5.0e{-}18$	0	91	91
(2.8e−9)	cqa-pse-interval-1	34	8	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	59	29	0.0	1.4e−26	0	67	76

email˙spec0˙product21˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	562	1	2.2e−19	0	-	-	-
	cqa-pse	2274	1230	2.2e−19	1 − 2.2e−19	371	8660	0
	cqa-sse	5394	3441	2.2e−19	1 − 2.2e−19	356	4833	0
	pse	2023	1575	2.2e−19	1 − 2.2e−19	372	8660	0
	sse	4597	2612	2.2e−19	1 − 2.2e−19	372	8660	0
(1 − 2.2e−19)	cqa-pse-interval-1	1712	1229	3.0e−28	1.0 − 3.0e−28	371	8660	0
	cqa-sse-interval-1	4832	3440	3.0e−28	1.0 − 3.0e−28	356	4833	0

Problem13˙label35˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1269	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1285	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1297	13	1.1e−46	1 − 3.0e−27	0	33	22
	pse	19	2	0.0	1 − 1.1e−18	0	39	25
	sse	54	27	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	16	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	28	12	0.0	3.0e−27	0	33	22

email˙spec8˙product22˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	576	1	1.0e−84	1 − 4.7e−10	-	-	-
	cqa-pse	1961	972	1.0e−28	1 − 1.0e−28	819	7209	0
	cqa-sse	5393	3327	1.0e−28	1 − 1.0e−28	812	5437	0
	pse	2003	1572	1.0e−28	1 − 1.0e−28	820	8212	0
	sse	5389	3454	1.0e−28	1 − 1.0e−28	820	7252	0
(4.7e−10)	cqa-pse-interval-1	1385	971	1.0e−28	1.0e−28	819	7209	0
	cqa-sse-interval-1	4817	3326	1.0e−28	1.0e−28	812	5437	0

Problem11˙label43˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1277	1	6.1e−28	1 − 2.8e−9	-	-	-
	cqa-pse	1329	40	6.1e−28	1 − 2.9e−27	0	698	1108
	cqa-sse	1660	275	6.1e−28	1 − 2.9e−27	0	698	1108
	pse	112	87	6.1e−28	1 − 2.0e−27	1	909	1417
	sse	1158	902	6.1e−28	1 − 2.0e−27	1	909	1417
(2.8e−9)	cqa-pse-interval-1	52	39	0.0	2.3e−27	0	698	1108
	cqa-sse-interval-1	383	274	0.0	2.3e−27	0	698	1108

Problem11˙label42˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	872	1	6.1e−28	1 − 2.8e−9	-	-	-
	cqa-pse	946	60	6.1e−28	1 − 2.9e−27	12	686	1108
	cqa-sse	1443	433	6.1e−28	1 − 2.9e−27	12	686	1108
	pse	110	86	6.1e−28	1 − 2.0e−27	19	892	1416
	sse	1266	991	6.1e−28	1 − 2.0e−27	19	892	1416
(2.8e−9)	cqa-pse-interval-1	74	59	2.3e−36	2.3e−27	12	686	1108
	cqa-sse-interval-1	571	432	2.3e−36	2.3e−27	12	686	1108

Problem13˙label45˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1261	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1277	4	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1284	10	1.1e−46	1 − 3.0e−27	0	33	22
	pse	23	4	0.0	1 − 1.1e−18	0	39	25
	sse	52	25	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	16	3	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	23	9	0.0	3.0e−27	0	33	22

Problem10_label15_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1751	1	2.4e−37	1 − 2.3e−9	-	-	-
	cqa-pse	1765	9	2.4e−37	1 − 1.3e−35	0	121	189
	cqa-sse	1854	87	2.4e−37	1 − 1.3e−35	0	121	189
	pse	24	15	2.4e−37	1 − 9.1e−28	1	213	212
	sse	201	169	2.4e−37	1 − 9.1e−28	1	213	212
(2.3e−9)	cqa-pse-interval-1	14	8	0.0	1.3e−35	0	121	189
	cqa-sse-interval-1	103	86	0.0	1.3e−35	0	121	189

Problem06_label11_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1372	1	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	1399	7	6.1e−56	1 − 1.4e−26	0	67	76
	cqa-sse	1419	22	6.1e−56	1 − 1.4e−26	0	67	76
	pse	58	8	0.0	1 − 5.0e−18	0	91	91
	sse	114	54	0.0	1 − 5.0e−18	0	91	91
(2.8e−9)	cqa-pse-interval-1	27	6	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	47	21	0.0	1.4e−26	0	67	76

Problem18_label19_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1355	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1363	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1369	8	1.3e−74	1 − 6.1e−28	0	20	3
	pse	9	1	0.0	1 − 2.2e−19	0	22	5
	sse	27	11	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	8	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	14	7	0.0	6.1e−28	0	20	3

Problem11_label51_false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1010	1	6.1e−28	$1-2.8e{-9}$	-	-	-
	cqa-pse	1084	59	6.1e−28	$1-2.9e{-27}$	27	671	1108
	cqa-sse	1590	447	6.1e−28	$1-2.9e{-27}$	27	671	1108
	pse	65	45	6.1e−28	$1-2.0e{-27}$	36	878	1415
	sse	1159	915	6.1e−28	$1-2.0e{-27}$	36	878	1413
(2.8e−9)	cqa-pse-interval-1	74	58	1.1e−36	2.3e−27	27	671	1108
	cqa-sse-interval-1	580	446	1.1e−36	2.3e−27	27	671	1108

Problem18_label35_false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1283	1	1.3e−74	$1-2.8e{-9}$	-	-	-
	cqa-pse	1293	4	1.3e−74	$1-6.1e{-28}$	0	20	3
	cqa-sse	1300	9	1.3e−74	$1-6.1e{-28}$	0	20	3
	pse	13	2	0.0	$1-2.2e{-19}$	0	22	5
	sse	26	11	0.0	$1-2.2e{-19}$	0	22	5
(2.8e−9)	cqa-pse-interval-1	10	3	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	17	8	0.0	6.1e−28	0	20	3

Problem13_label12_false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1321	1	1.1e−46	$1-2.3e{-9}$	-	-	-
	cqa-pse	1336	4	1.1e−46	$1-3.0e{-27}$	0	33	22
	cqa-sse	1348	13	1.1e−46	$1-3.0e{-27}$	0	33	22
	pse	21	3	0.0	$1-1.1e{-18}$	0	39	25
	sse	57	31	0.0	$1-1.1e{-18}$	0	39	25
(2.3e−9)	cqa-pse-interval-1	15	3	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	27	12	0.0	3.0e−27	0	33	22

Problem02˙label59˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	381	1	2.4e−37	1 − 2.3e−9	-	-	-
	cqa-pse	5392	1	2.4e−37	1 − 2.3e−9	0	0	0
	cqa-sse	5393	5011	2.4e−37	1 − 2.3e−9	0	0	0
	pse	5390	4764	0.0	1 − 1.9e−9	0	15344	19862
	sse	5389	3937	2.4e−37	1 − 2.4e−37	21	6122	0
(2.3e−9)	cqa-pse-interval-1	5011	0	0.0	0	0	0	0
	cqa-sse-interval-1	5012	5010	0.0	0	0	0	0

Problem02˙label50˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	504	1	2.4e−37	1 − 2.3e−9	-	-	-
	cqa-pse	5396	4890	2.4e−37	1 − 4.3e−18	0	18	0
	cqa-sse	5397	4251	2.4e−37	1 − 2.4e−37	15	4132	0
	pse	5389	4740	0.0	1 − 1.9e−9	0	16508	21402
	sse	5389	3792	2.4e−37	1 − 2.4e−37	26	6471	0
(2.3e−9)	cqa-pse-interval-1	4892	4889	0.0	4.3e−18	0	18	0
	cqa-sse-interval-1	4893	4250	4.4e−46	4.4e−46	15	4132	0

Problem01˙label37˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	403	2	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	5386	4332	6.1e−56	1 − 6.5e−18	0	15246	20682
	cqa-sse	5386	3978	6.1e−56	1 − 6.1e−56	0	5138	0
	pse	5388	4588	0.0	1 − 2.3e−9	0	26274	37584
	sse	5388	3685	6.1e−56	1 − 6.1e−56	1	7244	0
(2.8e−9)	cqa-pse-interval-1	4983	4330	0.0	6.5e−18	0	15246	20682
	cqa-sse-interval-1	4983	3976	0.0	7.2e−62	0	5138	0

Problem10˙label41˙false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1202	1	5.0e−28	$1-2.3e{-9}$	-	-	-
	cqa-pse	1220	12	5.0e−28	$1-1.8e{-27}$	0	171	188
	cqa-sse	1277	59	5.0e−28	$1-1.8e{-27}$	0	171	188
	pse	24	15	5.0e−28	$1-1.4e{-27}$	1	213	212
	sse	203	172	5.0e−28	$1-1.4e{-27}$	1	213	212
(2.3e−9)	cqa-pse-interval-1	18	11	0.0	1.3e−27	0	171	188
	cqa-sse-interval-1	75	58	0.0	1.3e−27	0	171	188

Problem06˙label58˙false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1512	1	6.1e−56	$1-2.8e{-9}$	-	-	-
	cqa-pse	5390	3851	6.1e−56	$1-3.9e{-18}$	0	59	64
	cqa-sse	5388	3851	6.1e−56	$1-1.4e{-26}$	0	29	13
	pse	48	5	0.0	$1-5.0e{-18}$	0	91	91
	sse	141	75	0.0	$1-5.0e{-18}$	0	91	91
(2.8e−9)	cqa-pse-interval-1	3878	3850	0.0	3.9e−18	0	59	64
	cqa-sse-interval-1	3876	3850	0.0	1.4e−26	0	29	13

Problem06˙label20˙false-unreach-call.c		time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
	cqa-#	1362	1	6.1e−56	$1-2.8e{-9}$	-	-	-
	cqa-pse	1395	9	6.1e−56	$1-1.4e{-26}$	0	67	76
	cqa-sse	1412	24	6.1e−56	$1-1.4e{-26}$	0	67	76
	pse	57	8	0.0	$1-5.0e{-18}$	0	91	91
	sse	121	55	0.0	$1-5.0e{-18}$	0	91	91
(2.8e−9)	cqa-pse-interval-1	33	8	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	50	23	0.0	1.4e−26	0	67	76

Problem18˙label06˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1289	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1299	4	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1308	10	1.3e−74	1 − 6.1e−28	0	20	3
	pse	12	2	0.0	1 − 2.2e−19	0	22	5
	sse	28	12	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	10	3	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	19	9	0.0	6.1e−28	0	20	3

email˙spec9˙product15˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	445	1	1.0e−84	1 − 4.7e−10	-	-	-
	cqa-pse	1840	986	4.7e−38	1 − 4.7e−38	397	7507	0
	cqa-sse	5398	3540	4.7e−38	1 − 4.7e−38	397	5301	0
	pse	1659	1279	4.7e−38	1 − 4.7e−38	398	8634	0
	sse	4784	2921	4.7e−38	1 − 4.7e−38	398	8634	0
(4.7e−10)	cqa-pse-interval-1	1395	985	4.7e−38	4.7e−38	397	7507	0
	cqa-sse-interval-1	4953	3539	4.7e−38	4.7e−38	397	5301	0

token˙ring.08˙false-unreach-call˙false-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	2006	1	2.2e−19	0	-	-	-
	cqa-pse	5022	3015	2.2e−19	0	0	0	0
	cqa-sse	5397	3390	2.2e−19	0	0	0	0
	pse	5388	3326	0.0	1 − 2.3e−9	0	16945	24054
	sse	5391	2940	2.2e−19	1 − 4.7e−10	43	439	793
(1 − 2.2e−19)	cqa-pse-interval-1	3016	3014	0.0	0	0	0	0
	cqa-sse-interval-1	3391	3389	0.0	0	0	0	0

email˙spec9˙product20˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	604	1	4.8e−94	1 − 4.7e−10	-	-	-
	cqa-pse	2325	1272	4.7e−38	1 − 4.7e−38	397	7507	0
	cqa-sse	5392	3391	4.7e−38	1 − 4.7e−38	397	4926	0
	pse	1646	1271	4.7e−38	1 − 4.7e−38	398	8634	0
	sse	5344	3375	4.7e−38	1 − 4.7e−38	398	8634	0
(4.7e−10)	cqa-pse-interval-1	1721	1271	4.7e−38	4.7e−38	397	7507	0
	cqa-sse-interval-1	4788	3390	4.7e−38	4.7e−38	397	4926	0

Problem01˙label57˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	388	1	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	1050	595	6.1e−56	1 − 6.5e−18	0	3143	4090
	cqa-sse	1034	564	6.1e−56	1 − 2.8e−53	0	1257	0
	pse	5388	4563	0.0	1 − 2.3e−9	0	27297	39136
	sse	5388	3702	6.1e−56	1 − 6.1e−56	1	7184	0
(2.8e−9)	cqa-pse-interval-1	662	594	0.0	6.5e−18	0	3143	4090
	cqa-sse-interval-1	646	563	0.0	2.8e−53	0	1257	0

Problem18˙label00˙false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1283	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1294	4	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1299	9	1.3e−74	1 − 6.1e−28	0	20	3
	pse	10	1	0.0	1 − 2.2e−19	0	22	5
	sse	28	13	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	11	3	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	16	8	0.0	6.1e−28	0	20	3

Problem02˙label13˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	812	1	5.0e−28	1 − 2.3e−9	-	-	-
	cqa-pse	3843	1	5.0e−28	1 − 2.3e−9	0	0	0
	cqa-sse	1099	285	5.0e−28	1 − 2.3e−9	0	0	0
	pse	5390	4736	0.0	1 − 1.9e−9	0	15820	20463
	sse	5389	3937	5.0e−28	1 − 5.0e−28	4	6110	0
(2.3e−9)	cqa-pse-interval-1	3031	0	0.0	0	0	0	0
	cqa-sse-interval-1	287	284	0.0	0	0	0	0

Problem01˙label50˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	644	1	1.3e−46	1 − 2.8e−9	-	-	-
	cqa-pse	1321	675	1.3e−46	1 − 2.8e−9	0	0	0
	cqa-sse	1305	660	1.3e−46	1 − 2.8e−9	0	0	0
	pse	5389	4520	0.0	1 − 2.3e−9	0	28634	40899
	sse	5389	3917	1.3e−46	1 − 1.3e−46	1	6924	0
(2.8e−9)	cqa-pse-interval-1	677	674	0.0	0	0	0	0
	cqa-sse-interval-1	661	659	0.0	0	0	0	0

email˙spec8˙product20˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	577	1	4.8e−94	1 − 4.7e−10	-	-	-
	cqa-pse	2209	1208	4.7e−38	1 − 4.7e−38	397	7507	0
	cqa-sse	5395	3348	4.7e−38	1 − 4.7e−38	397	5634	0
	pse	1927	1495	4.7e−38	1 − 4.7e−38	398	8634	0
	sse	5389	3479	4.7e−38	1 − 4.7e−38	398	7412	0
(4.7e−10)	cqa-pse-interval-1	1632	1207	4.7e−38	4.7e−38	397	7507	0
	cqa-sse-interval-1	4818	3347	4.7e−38	4.7e−38	397	5634	0

Problem01˙label33˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	375	1	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	5388	4622	6.1e−56	1 − 6.5e−18	0	15784	21466
	cqa-sse	5387	3896	6.1e−56	1 − 6.1e−56	0	5963	0
	pse	5388	4496	0.0	1 − 2.3e−9	0	28986	41301
	sse	5389	3663	6.1e−56	1 − 6.1e−56	1	7520	0
(2.8e−9)	cqa-pse-interval-1	5013	4621	0.0	6.5e−18	0	15784	21466
	cqa-sse-interval-1	5012	3895	0.0	6.3e−62	0	5963	0

Problem18˙label45˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1277	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1286	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1293	8	1.3e−74	1 − 6.1e−28	0	20	3
	pse	10	2	0.0	1 − 2.2e−19	0	22	5
	sse	26	11	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	9	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	16	7	0.0	6.1e−28	0	20	3

Problem06˙label27˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1380	1	2.8e−65	0	-	-	-
	cqa-pse	4428	3030	2.8e−65	1 − 2.3e−9	0	29	18
	cqa-sse	2344	945	2.8e−65	0	0	0	0
	pse	59	7	0.0	1 − 5.0e−18	0	91	91
	sse	116	57	0.0	1 − 5.0e−18	0	91	91
(1 − 2.8e−65)	cqa-pse-interval-1	3048	3029	0.0	1.0 − 2.3e−9	0	29	18
	cqa-sse-interval-1	964	944	0.0	0	0	0	0

Problem11_label34_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1303	1	2.8e−37	1 − 2.8e−9	-	-	-
	cqa-pse	1373	55	2.8e−37	1 − 2.5e−35	1	581	1063
	cqa-sse	1852	422	2.8e−37	1 − 2.5e−35	1	581	1063
	pse	63	44	2.8e−37	1 − 1.4e−27	2	908	1417
	sse	1178	919	2.8e−37	1 − 1.4e−27	2	908	1417
(2.8e−9)	cqa-pse-interval-1	70	54	1.3e−46	2.5e−35	1	581	1063
	cqa-sse-interval-1	549	421	1.3e−46	2.5e−35	1	581	1063

Problem13_label48_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1309	1	1.1e−46	0	-	-	-
	cqa-pse	1329	9	1.1e−46	1 − 1.3e−18	0	35	22
	cqa-sse	1372	51	1.1e−46	1 − 1.3e−18	0	35	22
	pse	22	3	0.0	1 − 1.1e−18	0	39	25
	sse	49	24	0.0	1 − 1.1e−18	0	39	25
(1 − 1.1e−46)	cqa-pse-interval-1	20	8	0.0	1.0 − 1.3e−18	0	35	22
	cqa-sse-interval-1	63	50	0.0	1.0 − 1.3e−18	0	35	22

Problem10_label24_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	946	1	2.4e−37	1 − 2.3e−9	-	-	-
	cqa-pse	958	8	2.4e−37	1 − 1.3e−35	0	122	188
	cqa-sse	1010	52	2.4e−37	1 − 1.3e−35	0	122	188
	pse	19	11	2.4e−37	1 − 9.1e−28	1	213	212
	sse	274	241	2.4e−37	1 − 9.1e−28	1	213	212
(2.3e−9)	cqa-pse-interval-1	12	7	0.0	1.3e−35	0	122	188
	cqa-sse-interval-1	64	51	0.0	1.3e−35	0	122	188

Problem13˙label21˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1330	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1345	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1353	9	1.1e−46	1 − 3.0e−27	0	33	22
	pse	21	3	0.0	1 − 1.1e−18	0	39	25
	sse	47	22	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	15	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	23	8	0.0	3.0e−27	0	33	22

Problem06˙label48˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1412	1	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	1447	9	6.1e−56	1 − 1.4e−26	0	67	76
	cqa-sse	1474	32	6.1e−56	1 − 1.4e−26	0	67	76
	pse	58	9	0.0	1 − 5.0e−18	0	91	91
	sse	110	50	0.0	1 − 5.0e−18	0	91	91
(2.8e−9)	cqa-pse-interval-1	35	8	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	62	31	0.0	1.4e−26	0	67	76

Problem06˙label24˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1394	1	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	1428	9	6.1e−56	1 − 1.4e−26	0	67	76
	cqa-sse	1449	27	6.1e−56	1 − 1.4e−26	0	67	76
	pse	45	6	0.0	1 − 5.0e−18	0	91	91
	sse	130	68	0.0	1 − 5.0e−18	0	91	91
(2.8e−9)	cqa-pse-interval-1	34	8	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	55	26	0.0	1.4e−26	0	67	76

Problem10˙label47˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	688	1	5.0e−28	1 − 2.3e−9	-	-	-
	cqa-pse	702	10	5.0e−28	1 − 1.9e−27	0	170	189
	cqa-sse	755	54	5.0e−28	1 − 1.9e−27	0	170	189
	pse	17	9	5.0e−28	1 − 1.4e−27	1	213	212
	sse	249	217	5.0e−28	1 − 1.4e−27	1	213	212
(2.3e−9)	cqa-pse-interval-1	14	9	0.0	1.4e−27	0	170	189
	cqa-sse-interval-1	67	53	0.0	1.4e−27	0	170	189

Problem13˙label24˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1276	1	5.1e−56	0	-	-	-
	cqa-pse	1306	19	5.1e−56	1 − 1.3e−18	0	34	22
	cqa-sse	1426	138	5.1e−56	1 − 1.3e−18	0	34	22
	pse	21	3	0.0	1 − 1.1e−18	0	39	25
	sse	56	29	0.0	1 − 1.1e−18	0	39	25
(1 − 5.1e−56)	cqa-pse-interval-1	30	18	0.0	1.0 − 1.3e−18	0	34	22
	cqa-sse-interval-1	150	137	0.0	1.0 − 1.3e−18	0	34	22

email˙spec9˙product30˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	609	1	4.8e−94	1 − 4.7e−10	-	-	-
	cqa-pse	2037	1010	4.7e−38	1 − 4.7e−38	397	7507	0
	cqa-sse	5397	3375	4.7e−38	1 − 4.7e−38	397	5313	0
	pse	1305	959	4.7e−38	1 − 4.7e−38	398	8634	0
	sse	4809	2855	4.7e−38	1 − 4.7e−38	398	8634	0
(4.7e−10)	cqa-pse-interval-1	1428	1009	4.7e−38	4.7e−38	397	7507	0
	cqa-sse-interval-1	4788	3374	4.7e−38	4.7e−38	397	5313	0

Problem13_label28_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1251	1	1.1e−46	$1 - 2.3e{-9}$	-	-	-
	cqa-pse	1264	3	1.1e−46	$1 - 3.0e{-27}$	0	33	22
	cqa-sse	1270	8	1.1e−46	$1 - 3.0e{-27}$	0	33	22
	pse	23	3	0.0	$1 - 1.1e{-18}$	0	39	25
	sse	50	24	0.0	$1 - 1.1e{-18}$	0	39	25
(2.3e−9)	cqa-pse-interval-1	13	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	19	7	0.0	3.0e−27	0	33	22

Problem01_label56_false-unreach-call_false-termination.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	414	2	6.1e−56	$1 - 2.8e{-9}$	-	-	-
	cqa-pse	5397	4346	6.1e−56	$1 - 6.5e{-18}$	0	14443	19497
	cqa-sse	5396	3978	6.1e−56	$1 - 6.1e{-56}$	0	5403	0
	pse	5393	4528	0.0	$1 - 2.3e{-9}$	0	28138	40263
	sse	5388	3681	6.1e−56	$1 - 6.1e{-56}$	1	7407	0
(2.8e−9)	cqa-pse-interval-1	4983	4344	0.0	6.5e−18	0	14443	19497
	cqa-sse-interval-1	4982	3976	0.0	6.8e−62	0	5403	0

Problem13_label07_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1309	1	1.1e−46	$1 - 2.3e{-9}$	-	-	-
	cqa-pse	1323	3	1.1e−46	$1 - 3.0e{-27}$	0	33	22
	cqa-sse	1331	10	1.1e−46	$1 - 3.0e{-27}$	0	33	22
	pse	23	3	0.0	$1 - 1.1e{-18}$	0	39	25
	sse	44	19	0.0	$1 - 1.1e{-18}$	0	39	25
(2.3e−9)	cqa-pse-interval-1	14	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	22	9	0.0	3.0e−27	0	33	22

token˙ring.10˙false-unreach-call˙false-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	2158	1	2.2e−19	0	-	-	-
	cqa-pse	5385	3225	2.2e−19	0	0	0	0
	cqa-sse	5385	3225	2.2e−19	0	0	0	0
	pse	5395	2421	0.0	1 − 3.3e−9	0	12875	33878
	sse	5392	2783	0.0	1 − 4.7e−10	0	842	598
(1 − 2.2e−19)	cqa-pse-interval-1	3227	3224	0.0	0	0	0	0
	cqa-sse-interval-1	3227	3224	0.0	0	0	0	0

Problem11˙label29˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1470	1	6.1e−28	1 − 2.8e−9	-	-	-
	cqa-pse	1554	54	6.1e−28	1 − 2.9e−27	1	697	1108
	cqa-sse	2033	425	6.1e−28	1 − 2.9e−27	1	697	1108
	pse	114	90	6.1e−28	1 − 2.0e−27	2	908	1417
	sse	1162	901	6.1e−28	1 − 2.0e−27	2	908	1417
(2.8e−9)	cqa-pse-interval-1	84	53	2.8e−37	2.3e−27	1	697	1108
	cqa-sse-interval-1	563	424	2.8e−37	2.3e−27	1	697	1108

Problem13˙label25˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1326	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1340	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1349	10	1.1e−46	1 − 3.0e−27	0	33	22
	pse	19	2	0.0	1 − 1.1e−18	0	39	25
	sse	54	28	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	14	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	23	9	0.0	3.0e−27	0	33	22

Problem03_label31_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1158	1	1.3e−46	1 − 2.8e−9	-	-	-
	cqa-pse	5391	4023	1.3e−46	1 − 6.5e−18	0	23817	29324
	cqa-sse	5391	3009	1.3e−46	1 − 1.3e−46	0	4620	0
	pse	5392	4766	0.0	1 − 2.3e−9	0	60941	72619
	sse	5395	3474	1.3e−46	1 − 1.3e−46	1	5322	0
(2.8e−9)	cqa-pse-interval-1	4233	4022	0.0	6.5e−18	0	23817	29324
	cqa-sse-interval-1	4233	3008	0.0	3.0e−71	0	4620	0

Problem18_label20_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1340	1	6.2e−84	1 − 4.7e−10	-	-	-
	cqa-pse	1706	360	6.2e−84	1 − 4.7e−10	0	0	0
	cqa-sse	1902	555	6.2e−84	1 − 4.7e−10	0	0	0
	pse	9	2	0.0	1 − 2.2e−19	0	22	5
	sse	27	11	0.0	1 − 2.2e−19	0	22	5
(4.7e−10)	cqa-pse-interval-1	366	359	0.0	0	0	0	0
	cqa-sse-interval-1	562	554	0.0	0	0	0	0

Problem18_label55_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1279	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1291	4	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1296	9	1.3e−74	1 − 6.1e−28	0	20	3
	pse	12	2	0.0	1 − 2.2e−19	0	22	5
	sse	23	8	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	12	3	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	17	8	0.0	6.1e−28	0	20	3

Problem01˙label15˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	434	2	2.6e−46	1 − 2.8e−9	-	-	-
	cqa-pse	5386	4953	2.6e−46	1 − 7.8e−18	0	1	0
	cqa-sse	5386	4952	2.6e−46	1 − 7.8e−18	0	1	0
	pse	5389	4506	0.0	1 − 2.3e−9	0	29562	42098
	sse	5389	3918	1.3e−46	1 − 1.3e−46	1	6611	0
(2.8e−9)	cqa-pse-interval-1	4952	4951	0.0	7.8e−18	0	1	0
	cqa-sse-interval-1	4952	4950	0.0	7.8e−18	0	1	0

Problem18˙label38˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1257	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1265	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1271	8	1.3e−74	1 − 6.1e−28	0	20	3
	pse	9	1	0.0	1 − 2.2e−19	0	22	5
	sse	26	11	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	8	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	14	7	0.0	6.1e−28	0	20	3

Problem02˙label16˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	606	1	5.0e−28	1 − 2.3e−9	-	-	-
	cqa-pse	5393	4786	5.0e−28	1 − 5.4e−18	0	1	0
	cqa-sse	5393	4786	5.0e−28	1 − 5.4e−18	0	1	0
	pse	5389	4754	0.0	1 − 1.9e−9	0	15449	19985
	sse	5389	3889	5.0e−28	1 − 5.0e−28	18	5944	0
(2.3e−9)	cqa-pse-interval-1	4787	4785	0.0	5.4e−18	0	1	0
	cqa-sse-interval-1	4787	4785	0.0	5.4e−18	0	1	0

Problem01˙label44˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	596	1	2.8e−37	1 − 2.8e−9	-	-	-
	cqa-pse	5398	4800	2.8e−37	1 − 6.5e−18	0	10	0
	cqa-sse	5398	4800	2.8e−37	1 − 9.1e−27	0	13	0
	pse	5389	4497	0.0	1 − 2.3e−9	3	29364	41869
	sse	5389	3965	2.8e−37	1 − 2.8e−37	56	6634	0
(2.8e−9)	cqa-pse-interval-1	4802	4799	0.0	6.5e−18	0	10	0
	cqa-sse-interval-1	4802	4799	0.0	9.1e−27	0	13	0

Problem03˙label50˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1147	1	1.3e−46	1 − 2.8e−9	-	-	-
	cqa-pse	5395	4049	1.3e−46	1 − 6.5e−18	0	22353	27434
	cqa-sse	5395	3856	1.3e−46	1 − 1.3e−46	0	2641	0
	pse	5393	4693	0.0	1 − 2.3e−9	0	66484	79034
	sse	5395	3056	1.3e−46	1 − 1.3e−46	1	6330	0
(2.8e−9)	cqa-pse-interval-1	4248	4048	0.0	6.5e−18	0	22353	27434
	cqa-sse-interval-1	4248	3855	0.0	3.3e−63	0	2641	0

Problem02˙label45˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	614	1	2.4e−37	1 − 2.3e−9	-	-	-
	cqa-pse	5386	4770	2.4e−37	1 − 4.3e−18	0	19	0
	cqa-sse	5386	3939	2.4e−37	1 − 2.4e−37	20	4787	0
	pse	5390	4775	0.0	1 − 1.9e−9	0	14254	18399
	sse	5389	3620	2.4e−37	1 − 2.4e−37	27	6706	0
(2.3e−9)	cqa-pse-interval-1	4772	4769	0.0	4.3e−18	0	19	0
	cqa-sse-interval-1	4772	3938	4.4e−46	4.4e−46	20	4787	0

Problem13_label04_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1280	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1293	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1302	8	1.1e−46	1 − 3.0e−27	0	33	22
	pse	20	2	0.0	1 − 1.1e−18	0	39	25
	sse	55	27	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	13	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	22	7	0.0	3.0e−27	0	33	22

Problem10_label28_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1492	1	4.7e−37	1 − 2.3e−9	-	-	-
	cqa-pse	1508	9	4.7e−37	1 − 1.4e−35	0	117	181
	cqa-sse	1565	57	4.7e−37	1 − 1.3e−35	0	118	181
	pse	17	10	2.4e−37	1 − 9.1e−28	1	213	212
	sse	254	221	2.4e−37	1 − 9.1e−28	1	213	212
(2.3e−9)	cqa-pse-interval-1	16	8	0.0	1.3e−35	0	117	181
	cqa-sse-interval-1	73	56	0.0	1.2e−35	0	118	181

Problem18_label08_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1267	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1276	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1281	8	1.3e−74	1 − 6.1e−28	0	20	3
	pse	9	1	0.0	1 − 2.2e−19	0	22	5
	sse	26	11	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	9	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	14	7	0.0	6.1e−28	0	20	3

token˙ring.04˙false-unreach-call˙false-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	570	1	2.2e−19	1 − 4.7e−10	-	-	-
	cqa-pse	5387	4815	5.4e−19	1 − 4.7e−10	2	0	0
	cqa-sse	5387	4815	2.2e−19	1 − 4.7e−10	0	0	0
	pse	5391	4459	0.0	1 − 9.3e−10	0	20930	8400
	sse	5391	4038	8.7e−19	1 − 4.7e−10	85	43	667
(4.7e−10)	cqa-pse-interval-1	4817	4814	3.3e−19	0	2	0	0
	cqa-sse-interval-1	4817	4814	0.0	0	0	0	0

Problem01˙label32˙false-unreach-call˙false-termination.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	376	1	2.8e−65	1 − 2.8e−9	-	-	-
	cqa-pse	5389	4622	2.8e−65	1 − 6.5e−18	0	16220	21445
	cqa-sse	5387	3807	2.8e−65	1 − 5.2e−62	0	4947	0
	pse	5388	4514	0.0	1 − 2.3e−9	0	28985	41301
	sse	5388	3905	2.8e−65	1 − 6.1e−62	1	6988	0
(2.8e−9)	cqa-pse-interval-1	5013	4621	0.0	6.5e−18	0	16220	21445
	cqa-sse-interval-1	5011	3806	0.0	5.2e−62	0	4947	0

Problem11˙label15˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1259	1	6.1e−28	1 − 2.8e−9	-	-	-
	cqa-pse	1312	40	6.1e−28	1 − 2.9e−27	6	692	1108
	cqa-sse	1646	269	6.1e−28	1 − 2.9e−27	6	692	1108
	pse	65	46	6.1e−28	1 − 2.0e−27	7	903	1417
	sse	1230	966	6.1e−28	1 − 2.0e−27	7	903	1417
(2.8e−9)	cqa-pse-interval-1	53	39	1.1e−36	2.3e−27	6	692	1108
	cqa-sse-interval-1	387	268	1.1e−36	2.3e−27	6	692	1108

Problem03_label09_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1179	2	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	5398	3490	6.1e−56	1 − 6.5e−18	0	18156	22820
	cqa-sse	5396	3219	6.1e−56	1 − 6.1e−56	0	4244	1
	pse	5393	4850	0.0	1 − 2.3e−9	0	54539	64925
	sse	5393	3086	6.1e−56	1 − 6.1e−56	1	6010	0
(2.8e−9)	cqa-pse-interval-1	4219	3488	0.0	6.5e−18	0	18156	22820
	cqa-sse-interval-1	4217	3217	0.0	3.1e−71	0	4244	1

Problem18_label03_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1299	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1308	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1313	7	1.3e−74	1 − 6.1e−28	0	20	3
	pse	13	2	0.0	1 − 2.2e−19	0	22	5
	sse	27	11	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	9	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	14	6	0.0	6.1e−28	0	20	3

Problem13_label29_false-unreach-call.c		time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
	cqa-#	1297	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1310	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1316	8	1.1e−46	1 − 3.0e−27	0	33	22
	pse	24	3	0.0	1 − 1.1e−18	0	39	25
	sse	44	20	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	13	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	19	7	0.0	3.0e−27	0	33	22

Problem18˙label25˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1320	1	1.3e−74	1 − 2.8e−9	-	-	-
	cqa-pse	1329	3	1.3e−74	1 − 6.1e−28	0	20	3
	cqa-sse	1336	8	1.3e−74	1 − 6.1e−28	0	20	3
	pse	13	2	0.0	1 − 2.2e−19	0	22	5
	sse	25	10	0.0	1 − 2.2e−19	0	22	5
(2.8e−9)	cqa-pse-interval-1	9	2	0.0	6.1e−28	0	20	3
	cqa-sse-interval-1	16	7	0.0	6.1e−28	0	20	3

email˙spec8˙product26˙false-unreach-call˙true-termination.cil.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	558	1	1.0e−84	1 − 4.7e−10	-	-	-
	cqa-pse	1943	950	1.0e−28	1 − 1.0e−28	819	7209	0
	cqa-sse	5391	3302	1.0e−28	1 − 1.0e−28	811	5443	0
	pse	2016	1567	1.0e−28	1 − 1.0e−28	820	8212	0
	sse	5389	3460	1.0e−28	1 − 1.0e−28	820	7152	0
(4.7e−10)	cqa-pse-interval-1	1385	949	1.0e−28	1.0e−28	819	7209	0
	cqa-sse-interval-1	4833	3301	1.0e−28	1.0e−28	811	5443	0

Problem13˙label44˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1297	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1310	3	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1317	8	1.1e−46	1 − 3.0e−27	0	33	22
	pse	18	2	0.0	1 − 1.1e−18	0	39	25
	sse	54	28	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	13	2	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	20	7	0.0	3.0e−27	0	33	22

Problem03_label43_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1075	1	1.3e−46	1 − 2.8e−9	-	-	-
	cqa-pse	5398	4081	1.3e−46	1 − 6.5e−18	44	24783	30589
	cqa-sse	5398	3131	1.3e−46	1 − 1.3e−46	23	4975	2
	pse	5391	4844	6.2e−84	1 − 2.3e−9	67	54174	64616
	sse	5394	3081	1.3e−46	1 − 1.3e−46	31	6038	0
(2.8e−9)	cqa-pse-interval-1	4323	4080	1.3e−74	6.5e−18	44	24783	30589
	cqa-sse-interval-1	4323	3130	1.8e−55	1.8e−55	23	4975	2

Problem13_label40_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1250	1	1.1e−46	1 − 2.3e−9	-	-	-
	cqa-pse	1263	4	1.1e−46	1 − 3.0e−27	0	33	22
	cqa-sse	1277	12	1.1e−46	1 − 3.0e−27	0	33	22
	pse	22	3	0.0	1 − 1.1e−18	0	39	25
	sse	51	25	0.0	1 − 1.1e−18	0	39	25
(2.3e−9)	cqa-pse-interval-1	13	3	0.0	3.0e−27	0	33	22
	cqa-sse-interval-1	27	11	0.0	3.0e−27	0	33	22

Problem11_label31_false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1457	2	2.8e−37	1 − 2.8e−9	-	-	-
	cqa-pse	5390	3933	2.8e−37	1 − 7.8e−18	0	1	0
	cqa-sse	5391	3930	2.8e−37	1 − 7.8e−18	0	1	0
	pse	71	51	2.8e−37	1 − 1.4e−27	2	908	1417
	sse	1345	1076	2.8e−37	1 − 1.4e−27	2	908	1417
(2.8e−9)	cqa-pse-interval-1	3933	3931	0.0	7.8e−18	0	1	0
	cqa-sse-interval-1	3934	3928	0.0	7.8e−18	0	1	0

Problem06˙label21˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1360	1	6.1e−56	1 − 2.8e−9	-	-	-
	cqa-pse	1387	7	6.1e−56	1 − 1.4e−26	0	67	76
	cqa-sse	1416	30	6.1e−56	1 − 1.4e−26	0	67	76
	pse	50	6	0.0	1 − 5.0e−18	0	91	91
	sse	143	77	0.0	1 − 5.0e−18	0	91	91
(2.8e−9)	cqa-pse-interval-1	27	6	0.0	1.4e−26	0	67	76
	cqa-sse-interval-1	56	29	0.0	1.4e−26	0	67	76

Problem11˙label20˙false-unreach-call.c		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	1368	1	2.8e−37	1 − 2.8e−9	-	-	-
	cqa-pse	1419	37	2.8e−37	1 − 2.5e−35	0	583	1062
	cqa-sse	1734	267	2.8e−37	1 − 2.5e−35	0	583	1062
	pse	111	87	2.8e−37	1 − 1.4e−27	2	908	1417
	sse	1292	1012	2.8e−37	1 − 1.4e−27	2	908	1417
(2.8e−9)	cqa-pse-interval-1	51	36	0.0	2.5e−35	0	583	1062
	cqa-sse-interval-1	366	266	0.0	2.5e−35	0	583	1062

nec11˙false-unreach-call˙false-termination.i		time	#-time	#(ψ)	#(¬ψ)	p(ψ)	p(¬ψ)	p(?)
	cqa-#	14	1	4.7e−10	1 − 4.7e−10	-	-	-
	cqa-pse	14	1	4.7e−10	1 − 4.7e−10	-	-	-
	cqa-sse	14	1	4.7e−10	1 − 4.7e−10	-	-	-
	pse	0	0	0.0	0	0	0	0
	sse	0	0	0.0	0	0	0	0

Ackermann02˙false-unreach-call˙true-no-overflow˙true-termination.c	time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
cqa-#	73	1	2.2e−19	1 − 2.2e−19	-	-	-
cqa-pse	73	1	2.2e−19	1 − 2.2e−19	-	-	-
cqa-sse	73	1	2.2e−19	1 − 2.2e−19	-	-	-
pse	7	3	2.2e−19	1 − 5.4e−18	1	50	24
sse	11	7	2.2e−19	1 − 5.4e−18	1	50	24

Fibonacci04˙false-unreach-call˙true-no-overflow˙true-termination.c	time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
cqa-#	22	1	4.7e−10	1 − 4.7e−10	-	-	-
cqa-pse	22	1	4.7e−10	1 − 4.7e−10	-	-	-
cqa-sse	22	1	4.7e−10	1 − 4.7e−10	-	-	-
pse	130	3	4.7e−10	1 − 5.0e−1	1	20	0
sse	186	6	0.0	0	0	0	0

kbfiltr˙simpl2˙false-unreach-call˙true-valid-memsafety˙true-termination.cil.c	time	#-time	#(ψ)	#($\neg\psi$)	p(ψ)	p($\neg\psi$)	p(?)
cqa-#	507	1	2.2e−19	1 − 2.2e−19	-	-	-
cqa-pse	507	1	2.2e−19	1 − 2.2e−19	-	-	-
cqa-sse	507	1	2.2e−19	1 − 2.2e−19	-	-	-
pse	133	123	2.2e−19	1 − 2.2e−19	4	296	0
sse	224	211	2.2e−19	1 − 2.2e−19	4	296	0

Fibonacci05˙false-unreach-call˙true-no-overflow˙true-termination.c	time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
cqa-#	43	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
cqa-pse	43	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
cqa-sse	43	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
pse	98	2	4.7e−10	$1 - 5.0e{-}1$	1	20	0
sse	132	5	0.0	0	0	0	0

id˙b3˙o2˙false-unreach-call˙true-termination˙true-no-overflow.c	time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
cqa-#	26	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
cqa-pse	26	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
cqa-sse	26	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
pse	103	67	4.7e−10	$1 - 1.0$	1	494	4
sse	1904	1857	4.7e−10	$1 - 1.0$	1	494	4

McCarthy91˙false-unreach-call˙true-no-overflow˙true-termination.c	time	#-time	$\#(\psi)$	$\#(\neg\psi)$	$p(\psi)$	$p(\neg\psi)$	p(?)
cqa-#	44	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
cqa-pse	44	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
cqa-sse	44	1	4.7e−10	$1 - 4.7e{-}10$	-	-	-
pse	5379	344	4.7e−10	$1 - 5.0e{-}1$	1	4618	0
sse	5385	4426	0.0	$1 - 5.0e{-}1$	0	38	1