Incorrectness Logic

PETER W. O'HEARN, Facebook and University College London, UK

10

Program correctness and incorrectness are two sides of the same coin. As a programmer, even if you would like to have correctness, you might find yourself spending most of your time reasoning about incorrectness. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools. This paper describes a simple logic for program incorrectness which is, in a sense, the other side of the coin to Hoare's logic of correctness.

CCS Concepts: • **Theory of computation** → **Programming logic**.

Additional Key Words and Phrases: Proofs, Bugs, Static Analysis

ACM Reference Format:
Peter W. O'Hearn. 2020. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (January 2020), 32 pages. https://doi.org/10.1145/3371078

1 INTRODUCTION

When reasoning informally about a program, people make abstract inferences about what might go wrong, as well as about what must go right. A programmer might ask "will the program crash if we give it a large string?", without saying *which* large string. In this paper we investigate the hypothesis that reasoning about the presence of bugs can be underpinned by sound techniques in a principled logical system, just as reasoning about correctness (absence of bugs) has been demonstrated to have sound logical principles in an extensive research literature. We also consider the relationship of the principles to automated reasoning tools for finding bugs in software.

We explore our hypothesis by defining incorrectness logic, a formalism that is similar to Hoare's logic of program correctness [Hoare 1969], except that it is oriented to proving incorrectness rather than correctness. Hoare's theory is based on specifications of the form

$$\{\textit{pre-condition}\}\textit{code}\{\textit{post-condition}\}$$

which say that the post-condition *over-approximates* (describes a superset of) the states reachable upon termination when the code is executed starting from states satisfying the pre-condition (the so-called strongest post). Conversely, we use a specification form

$$[\textit{presumption}]\textit{code}[\textit{result}]$$

which says that the post-assertion *result* be an under-approximation (subset) of the final states that can be reached starting from states satisfying the *presumption*.

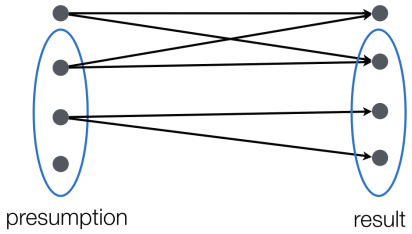
The under-approximate triples were studied (with a different but equivalent definition) previously by de Vries and Koutavas [2011] in their reverse Hoare logic, which they used to specify randomized algorithms. Incorrectness logic adds post-assertions for errors as well as for normal termination, and these assertions describe erroneous states that can be reached by actual program executions. Dijkstra [1976] famously remarked that "testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence," and he made this remark while arguing for the

Author's address: Peter W. O'Hearn, Facebook and University College London, UK.

2475-1421/2020/1-ART10
https://doi.org/10.1145/3371078

use of mathematical proof to show that bugs don't occur. Here we obtain a logic where instead one can prove the presence of bugs but not their absence: Because of under-approximation, reasoning is arranged to avoid false positives (bug suggestions that are not true). We will attempt to show that the resulting logic has principles relevant to pragmatic issues in the design of reasoning tools.

The simple shift from over to under has a significant impact on what specifications mean. The post-condition in correctness logic constrains what happens, ruling out anything that doesn't satisfy it, where the result assertion in incorrectness logic indicates the positive fact that certain states, at least those satisfying it, can occur. But the result assertion does not rule out other final states, or even that a pre-state might have no successers (i.e., divergence). As a picture:



If the code might be applied to any state satisfying the *presumption*, then we will be able to reach *any* state in the *result* assertion. Or more briefly, from the presumption we achieve the result.

In colloquial terms we might say that Hoare triples speak the whole truth, where the under-approximate triples speak nothing but the truth.

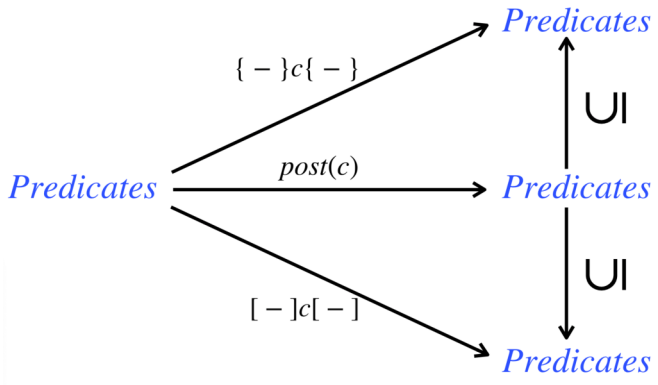
Incorrectness logic is so basic that it could have been defined and studied immediately after or alongside the fundamental works of Floyd [1967] and Hoare [1969] on correctness in the 1960s. It is the intervening developments in the science and, especially, the engineering of reasoning tools that motivated us to do so. The work in this paper arose as part of an attempt to re-imagine how static reasoning about programs might be done, taking the perspective of bug catching instead of (only) proving absence of bugs. It was motivated, in particular, by work at Facebook on static analysis [Distefano et al. 2019]. Informal, abstract reasoning about the presence of bugs is a fundamental part of communication between people during the code review process, and in their deployment at Facebook mechanized reasoning tools act as bots participating in these conversations: in these deployments they often use symbolic reasoning to identify regressions rather than to prove than none can occur, just as human reviewers frequently do.

The paper was also inspired by the use of symbolic execution for testing [Cadar and Sen 2013]. More generally, many practical tools use symbolic reasoning for bug catching, but few of them are accompanied by soundness arguments, and they have sometimes been considered as unprincipled. We suggest that static bug catchers could in future be cast in terms of finding logical proofs of under-approximation in a formal system, analogously to how correctness-oriented tools can be thought of as finding proofs.

While we appeal to tool-based considerations as we go along, we attempt to keep the presentation self contained. We give suggestive examples and comments on relevance to automatic program analysis problems, but don't delve into any specific analyses or tools.

2 UNIFIED PICTURE

Although our technical focus in this article is on incorrectness, it will be useful first to describe correctness and incorrectness reasoning in a way that emphasizes their relationship to one another. Readers who prefer seeing examples first can safely skip forward to the next section.



$$\wedge\vee \textit{ Symmetry:}\quad [p]c[q_1] \wedge [p]c[q_2] \iff [p]c[q_1 \vee q_2]$$
$$\{p\}c\{q_1\} \wedge \{p\}c\{q_2\} \iff \{p\}c\{q_1 \wedge q_2\}$$

$$\Uparrow\Downarrow \textit{ Symmetry:}\quad p' \Leftarrow p \wedge [p]c[q] \wedge q \Leftarrow q' \implies [p']c[q']$$
$$p' \Rightarrow p \wedge \{p\}c\{q\} \wedge q \Rightarrow q' \implies \{p'\}c\{q'\}$$

$$\textit{Principle of Agreement:}\quad [u]c[u'] \wedge u \Rightarrow o \wedge \{o\}c\{o'\} \implies u' \Rightarrow o'$$

$$\textit{Principle of Denial:}\quad [u]c[u'] \wedge u \Rightarrow o \wedge \neg(u' \Rightarrow o') \implies \neg(\{o\}c\{o'\})$$

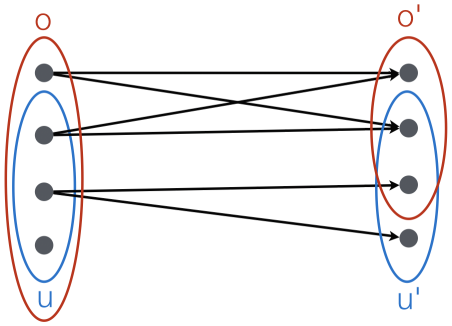
Fig. 1. Correctness and Incorrectness Principles

Figure 1 depicts the two triples. *Predicates* in the diagram represents the set of subsets of a collection of program states, and the arrows are binary relations. The relation $[-]c[-]$ relates any pair (p, q) of predicates when $[p]c[q]$ holds, and similarly for $\{-\}c\{-\}$. $post(c)$ is actually a function (a single-valued relation), which maps each input predicate to the set of those states reachable upon termination; the others are non-functional relations. The picture is a commuting diagram in the category of sets and binary relations: i.e., $[-]c[-] = post(c); \supseteq$ and $\{-\}c\{-\} = post(c); \subseteq$, where ';' is sequential composition of binary relations. The diagram can be taken to define $[-]c[-]$ and $\{-\}c\{-\}$ in terms of $post(c)$. A rigorous semantics of these notions will be given later in Section 5.

$post(c)p$ is often called the "strongest" post-condition, because it is the smallest set of states amongst those q satisfying $\{p\}c\{q\}$. This nomenclature reflects the historical accident of focus on the top triangle, and not the equally (we shall attempt to show) natural bottom. $post(c)p$ is also the largest q satisfying $[p]c[q]$, what we might call the "weakest under-approximate post".

Below the diagram we list several principles. The under-approximate triple part of $\wedge\vee$ symmetry is central: it supports sound reasoning covering fewer than all the paths in a program. The correctness version of the symmetry is true in our framework but the $\Rightarrow$ implication has sometimes been denied in correctness logics (e.g., [Gotsman et al. 2011]). Both parts of the $\Uparrow\Downarrow$ symmetry are important: these are "rules of consequence" which allow specifications to be adapted to broader contexts.

To understand the Principles of Agreement and Denial is it helpful fo appeal to testing terminology. In this picture



the regions labelled u and u' represent assertions in an under-approximate triple $[u]c[u']$ and the horizontal lines show the transition relation of the program. We think of the post-condition o' as the "test oracle" in a *putative* Hoare triple $\{o\}c\{o'\}$. If the question "is this a member of o'?" fails for a final state obtained by executing the program from a start state satisfying o, then the correctness triple is false. Denial says, as in this picture, that if we have an under-approximate triple taking a subset u of o to u', and part of u' lies outside of o', then our test oracle will fail on it, denying the putative triple; i.e., $\neg(\{o\}c\{o'\})$. Agreement gives conditions under which the oracle will be happy: the picture would adjust so that u' was contained in o'. Note that Agreement and Denial are logically equivalent. Traditional program testing corresponds to when u and u' are both singleton sets describing a test run. Incorrectness logic uses predicates to describe bigger sets, while remaining under-approximate.

In this paper we won't study correctness and incorrectness triples together, and will use a program statement `error()` to give the capability of test/verification oracles. This is to keep the paper contained; ultimately, it does make sense for them to be used together.

There has been important work which uses logical reasoning to help generate test cases [Cadar and Sen 2013]. Property-based testing uses assertions as test oracles [Claessen and Hughes 2000], but as in the discussion above of Agreement and Denial, this is essentially as in Hoare logic: the assertions over-approximate the expected. But, logical principles for showing the presence of bugs have not been as extensively studied as those for showing their absence. Using logic for incorrectness potentially has many of the same benefits as it does for correctness.

The essential point is that symbolic, logical reasoning can cover many states at once, even many program paths. While this does give the alluring in-principle ability to cover *all* paths, to prove the absence of errors, even when fewer than all are covered major advantages can accrue; in particular, it can be the case that symbolically approximating many states can be done more compactly, and efficiently, than enumerating the individual states, and this is the case whether or not we are in the, in practice, comparatively rare situation that every possible state and path is covered.

Incorrectness logic is also relevant to specifying interfaces. In the ideal case, assertions describing correctness conditions at exit points in a program (post-assertions) should be over-approximate, while for incorrectness they should be under-approximate.

It is important to note that the exact reasoning of the middle line of the diagram is definable mathematically but not computable (unless highly incomputable formulae are used to describe the post). Approximating in either direction provides a way to escape undecidability. Over-approximate reasoning can prove the absence of errors [Cousot and Cousot 1977]: if an over-approximation claims there are no errors, then (modulo assumptions) there will be none when a program is executed. Tools based on over-approximation suffer from false positives, reports of bugs that cannot happen; a state satisfying a post-condition might not be reachable, and apparent errors that follow from it can be false positives. On the other hand, under-approximation is relevant to

guaranteeing the presence of errors: if a state leads to an error, and satisfies an assertion describing an under-approximation of the reachable program states, then it can occur in a real execution. Under-approximation removes false positives, but opens the way to false negatives (missed bugs).

Finally, the reader might be wondering, why do we need incorrectness reasoning? Can't we just negate correctness reasoning? That might be the possible in an ideal world where full formal specifications were written for all software beforehand, and where there were no decidability or efficiency issues to contend with. But, the inability to prove an over-approximate spec (whether found by a tool or specified by a human) does not imply an error in a program, and neither does not having found a bug imply that there are none: thus, the need for dedicated techniques for each.

3 BEGINNING EXAMPLES

Incorrectness logic provides a perspective on reasoning about programs that, based on our experience describing it to people, appears to take some getting used to. So, we start with an informal warm up session before moving on to the formal development.

3.1 Under-approximating Triples

In examples we will use "presumes" to identify an incorrectness logic pre-assertion and "achieves" to identify a post-assertion. This is by analogy with the use of "requires" and "ensures" in some tools for correctness reasoning (e.g., https://en.wikipedia.org/wiki/Java_Modeling_Language). Note that we are not making a concrete recommendation that "presumes" and "achieves" be used in a specification language: these terms are used in code mainly to help us explain examples as we explore the theory.

Consider this program, together with the presumption and result.

```
1  /* presumes: [z==11] */
2    if (x is even) {
3       if (y is odd) {
4         z=42;
5    }  }
6  /* achieves: [z==42] */
```

Question: is the assertion z==42, the result, an under-approximation of the states reachable by executing this code starting from the presumption?

We have found that people very often answer yes, initially. Then we point out, it is not. The reason is that the assertion z==42 is satisfied by states where x is odd or y is even. E.g., the state [z:42,x:1,y:2] satisfies z==42, but is not reachable starting from the pre. So, we would say that this is a false under-approximate triple, but that

```
1  /* presumes: [z==11] */
2    if (x is even) {
3      if (y is odd) {
4         z=42;
5    }  }
6  /* achieves: [z==42 && (x is even) && (y is odd) ] */
```

is a true under-approximate triple. (Similar issues come up for post-conditions associated with errors cause by `error()` statements, by division by 0, by use-after-free, etc. The question arises: what is the post-state when the error occurs?) You might also wonder: why did we include z==11 as the pre in the first triple? What if we replaced it with `true`. Would z==42 on its own then be a true under-approximation of the post? The answer is yes.

At this point folks have exclaimed: What is going on? People are sometimes led to an initial "yes" answer to the question above because they are unwittingly applying the rule of weakening the

post-condition. This lets us make an inference like

$$\frac{\{p\}C\{q \wedge r\}}{\{p\}C\{q\}}$$

which drops conjuncts. You just can't do that soundly when reasoning about under-approximation.

In fact, there is a fundamental logic for reasoning about under-approximation. Above, when concluding a correct under-approximate post-assertion for our program, you might have noticed that the post-assertion corresponds to only one path through the program: it doesn't cover them all. We (silently) applied the incorrectness logic part of the principle of $\wedge\vee$ Symmetry from before. Not only does incorrectness logic deny weakening the post-condition, it supports other rules like

$$\frac{[p]C[q_1 \vee q_2]}{[p]C[q_1]}$$

which allow us to focus on fewer than all the paths, a feature which is a hallmark of under-approximation. What is more, soundly dropping disjuncts is a useful capability which can be called upon in order to help a reasoning tool scale.

3.2 Specifying Incorrectness

We can reason about errors by distinguishing result-assertion forms, for normal and erroneous or abnormal termination. For example, consider the following program, which is a variant of one used to illustrate buffer overflows by One [1996].

```
1  void foo(char* str)
2  /*  presumes: [*str[]==s]
3      achieves: [er: *str[]==s && length(s) > 16 ] */
4    { char buf[16];
5      strcpy(buf,str);
6    }
7
8  int main(int argc, char *argv[])
9    { foo(argv[1]); }
```

The specification for `foo()` says that if the length of the input string is greater than 16 then we can get an error (in this case a buffer overflow). (There, we are writing `*str[]==s` to indicate that sequence s held as a null-terminated string starting at address str.)

Remark: it is not necessary that a segmentation fault must occur in C when a buffer overflows (worse can happen). Incorrectness is an abstraction that a programmer or tool engineer decides upon to help in engineering concerns for program construction. Logic provides a means to specify these assumptions, and then to perform sound reasoning based on them, but it does not set the assumptions.

3.3 Under-approximate Success

Even if we were mainly interested in incorrectness, under-approximate result assertions describing successful computations can help us soundly discover bugs that come after a procedure is called. In particular, if we were to have over-approximate assertions only for successful computations, then our reasoning could go wrong, as the following example illustrates.

```
1  void mkeven()
2  /*  presumes: [true],  wrong achieves: [ok: x==2 || x==4]     */
3    { x=2; }
4
5  void usemkeven()
6    {  mkeven();  if (x==4) {error();} }
```

We use `ok:` before an assertion to indicate that it describes a result for normal, not exceptional, termination of a program. The achieves assertion `mkeven()` describes an over-approximation of what the procedure produces, including a possibility (`x==4`) than cannot occur. If we were to use this wrong achieves assertion in `usemkeven()` to conclude that an error is possible then this would be a false positive warning.

For this reason, our formalism will include under-approximate achieves-assertions for both successful and erroneous termination. `mkeven()` achieves "`ok: x==2`", not "`ok: x==2 || x==4`".

The possibility of divergence gives rise to many more such examples. If we consider a version of `mkeven()` with body **`while`**`(true){};x=2`, then `x==4` is an over-approximate post-condition, and it lines up perfectly with the condition to trip the error, but this would again be a false positive. Reasoning about termination is needed to be under-approximate; but, as we shall see, what is needed is not the same as showing termination on all inputs.

4 PROOF SYSTEM

We will use a language for simple imperative programs in which a statement `error()` halts execution and raises a particular error signal, *er*. `error()` can be used to define an `assert(B)` statement that allows correctness (and hence, incorrectness) conditions to be written, in the same way that `assert()` is defined in terms of `abort()` in the C programming language.

The `error()` statement leads to abnormal termination, where control never returns to the statement following it. E.g., in

```
x=1; error(); x=42
```

we know that `x=42` will not be executed, and so x will be 1 in the final "abnormal" state. Abnormal control flows impact reasoning about sequential composition, and a standard approach to these flows is to associate assertions with different exit conditions [Clint and Hoare 1972].

Let ϵ range over a collection of 'exit conditions' (or exceptions), to include at least *ok* (for normal termination) and *er* (caused by `error()`). We will consider a specification form

$[p]C[\epsilon\colon q]$: q under-approximates the states when C exits via ϵ starting from states in p.

In this paper we are going to concentrate on a single kind of error, *er*, arising from `error()`. This is to concentrate on the essential technical issues. Obviously we could consider and distinguish different kinds of error (say, for division by zero, null pointer exceptions, etc).

We treat the assertions, the p, q in $[p]C[\epsilon\colon q]$, semantically in this paper. That is, an assertion is assumed to be a subset of Σ, and we don't fix a language for describing these subsets. The concept of a free variable referred to in proof rules is treated semantically by saying that x is free in p if p is invariant under changing x: i.e., $\sigma \in p \Leftrightarrow \forall v.\, (\sigma|x \mapsto v) \in p$, where $(\sigma \mid x \mapsto v)$ for the function like σ except that x maps to v. We sometimes equivalently say that p is independent of x. Substitution, $p[e/x]$, is also understood in the evident way as $\{(\sigma \mid x \mapsto [\![e]\!]\sigma) \mid \sigma \in p\}$ where $[\![e]\!]\sigma$ is the value of expression e in state σ. We use standard logical notation to refer to the complete Boolean algebra structure of the powerset $P(\Sigma)$ – $\wedge$ is intersection, $\vee$ is union, $\Rightarrow$ is subset inclusion, $\exists$ is existential quantification, etc. By treating assertions semantically we are essentially appealing to mathematics (or set theory) as an oracle in our proof theory when we use $\Rightarrow$ in proof rules.

We sometime write a quadruple

$[p]C[ok\colon q][er\colon r]$ as shorthand for $[p]C[ok\colon q]$ and $[p]C[er\colon r]$

We often write result assertions for normal termination in green and abnormal in red. In cases where the result could be either, as in $[p_i]C[\epsilon\colon q_i]$ we keep the more neutral blue.

It's important to note that the existence of a (consistent) error assertion does not indicate a definite bug in a program. Consider a procedure `f(x) = {assert(x!=0);...}`, with error spec

Empty under-approximates

$$[p]C[\epsilon\colon \mathit{false}]$$

Consequence

$$\frac{p' \Leftarrow p \quad [p]C[\epsilon\colon q] \quad q \Leftarrow q'}{[p']C[\epsilon\colon q']}$$

Disjunction

$$\frac{[p_1]C[\epsilon\colon q_1] \quad [p_2]C[\epsilon\colon q_2]}{[p_1 \vee p_2]C[\epsilon\colon q_1 \vee q_2]}$$

Unit

$$[p]\texttt{skip}[ok\colon p][er\colon \mathit{false}]$$

Sequencing (short-circuit)

$$\frac{[p]C_1[er\colon r]}{[p]C_1; C_2[er\colon r]}$$

Sequencing (normal)

$$\frac{[p]C_1[ok\colon q] \quad [q]C_2[\epsilon\colon r]}{[p]C_1; C_2[\epsilon\colon r]}$$

Iterate zero

$$[p]C^\star[ok\colon p]$$

Iterate non-zero

$$\frac{[p]C^\star; C[\epsilon\colon q]}{[p]C^\star[\epsilon\colon q]}$$

Backwards Variant (where n fresh)

$$\frac{[p(n) \wedge nat(n)]C[ok\colon p(n+1) \wedge nat(n)]}{[p(0)]C^\star[ok\colon \exists n.p(n) \wedge nat(n)]}$$

Choice (where $i = 1$ or 2)

$$\frac{[p]C_i[\epsilon\colon q]}{[p]C_1 + C_2[\epsilon\colon q]}$$

Error

$$[p]\texttt{error}()[ok\colon \mathit{false}][er\colon p]$$

Assume

$$[p]\texttt{assume } B[ok\colon p \wedge B][er\colon \mathit{false}]$$

$$\texttt{while } B \texttt{ do } C \quad =_{def} \quad (\texttt{assume}(B);\ C)^\star; \texttt{assume}(\neg B)$$

$$\texttt{if } B \texttt{ then } C \texttt{ else } C' \quad =_{def} \quad (\texttt{assume}(B);\ C) + (\texttt{assume}(\neg B);\ C')$$

$$\texttt{assert}(B) \quad =_{def} \quad \texttt{assume}(B) + (\texttt{assume}(\neg B);\ \texttt{error}())$$

Fig. 2. Generic Proof Rules of Incorrectness Logic

Assignment

$$[p]x = e[ok\colon \exists x'.p[x'/x] \wedge x = e[x'/x]][er\colon \mathit{false}]$$

Nondet Assignment

$$[p]x = \texttt{nondet}()[ok\colon \exists x'p][er\colon \mathit{false}]$$

Constancy

$$\frac{[p]C[\epsilon\colon q]}{[p \wedge f]C[\epsilon\colon q \wedge f]} \ \mathit{Mod}(C) \cap \mathit{Free}(f) = \emptyset$$

Local Variable

$$\frac{[p]C(y/x)[\epsilon\colon q]}{[p]\texttt{local } x.C[\epsilon\colon \exists y.q]} \ y \notin \mathit{Free}(p, C)$$

Substitution I

$$\frac{[p]C[\epsilon\colon q]}{([p]C[\epsilon\colon q])(e/x)} \ \big(\mathit{Free}(e) \cup \{x\}\big) \cap \mathit{Free}(C) = \emptyset$$

Substitution II

$$\frac{[p]C[\epsilon\colon q]}{([p]C[\epsilon\colon q])(y/x)} \ y \notin \mathit{Free}(p, C, q)$$

Fig. 3. Rules for Variables and Mutation

$[x==0]\texttt{f(x)}[er\colon x==0]$. A tool would often choose not to flag an error in `f()`, instead warning at call sites that supply 0. We don't expect error specs necessarily to be ephemeral, to be removed as bugs are fixed: a spec like this could be kept in cache, even when part of a complete program that is error free, to be reused to help protect against future regressions when the program is changed.

We give two sets of proof rules. Those in Figure 2 are generic ones that we expect to be valid in many models, allowing for different models of states or commands themselves. Two such models

(based on traces and on separation) are mentioned in the last section of the paper. Figure 3 contains rules which assume that the states are functions of type *Variables* → *Values*.

Instead of considering while loops and if statements, we have Kleene iteration $C^\star$, assume statements, and nondeterministic choice (+); these let us encode while and if. Kleene iteration does zero or more iterations of C before exiting, behaving as the recursive equation $C^\star \equiv \texttt{skip} + C; C^\star$. In the `assume(B)` statement B is a Boolean expression, which we allow can be built up using the primitives in an otherwise-unspecified first-order logic signature. The commands overall are those built up from primitive commands using choice, iteration and sequential composition.

We use the notation $x = e$ for assignment and $e == e'$ for equality. $Mod(C)$ is the set of variables modified by assignment statements in C, and $Free(r)$ is the set of free variables in an assertion r. Normal assignment $x = e$ and $x = \texttt{nondet}()$ are distinct syntactic forms. In the former, e is an expression built up from a first-order logic signature, can appear within assertions, and is side-effect free. `nondet()` does not appear in assertions. Incorrectness logic uses Floyd's forward-running assignment axiom (see Figure 3) rather than Hoare's backwards-running one. We actually cannot use Hoare's, which substitutes into the pre as $[p[e/x]]x = e[ok\colon p]$, because it is unsound here; for example, in $[42 == y]x = 42[ok\colon x == y]$ the post-assertion does not under-approximate the states reachable from the pre: a non-reachable state satisfying $x == y$ is one where both x and y denote 3.

The axioms for `assume` and `skip` give the expected assertions for normal termination, but specify *false* (the empty set of states) for abnormal. *false* is a valid result assertion for either normal or abnormal termination, for any program, and we have included an axiom to that effect.

Incorrectness logic's rule of consequence lets us enlarge (weaken) the pre and shrink (strengthen) the post-assertion. As we indicated earlier, this lets us drop disjuncts from the post, using the implication $q \Rightarrow q \vee q'$. Conversely, we can drop conjuncts in the pre:

$$\frac{[p_1 \wedge p_2]C[\epsilon\colon q]}{[p_1]C[\epsilon\colon q]}\ .$$

Enlarging the pre was used in the Abductor tool ([Calcagno et al. 2011], which led to Facebook Infer), when guessing pre-conditions in programs with loops. This operation was unsound in the over-approximating logic used there, and was one of the reasons for a re-execution step which filtered out unsound pre-conditions. This step would be sound in incorrectness logic.

The first rule for sequencing is for short-circuiting, when `error()` happens within the first statement, and the second covers sequencing when the first command terminates normally. The *Iterate zero* rule corresponds to immediate exit from the loop. It shows that any assertion is a valid under-approximate invariant for Kleene iteration; loop invariants don't play a central role in under-approximate reasoning like they do for over-approximate. The *Iterate non-zero* rule uses $C^\star; C$ rather than $C; C^\star$ to help reasoning about cases where an error is thrown inside an iteration: In that case some number of successful iterations will occur, and the error will be thrown on the last iteration. To reason about such cases you can first find a result assertion for normal termination, the "frontier" before then error-throwing iteration, and then reason about errors that flow from the frontier in the next step. An example showing this capability will be used in Section 6.1, and this reasoning pattern will be used in the completeness proof in Section 5.1.

We can derive useful rules for choice and iteration using the disjunction rule.

$$\frac{[p]C^i[\epsilon\colon q_i],\ \text{all } i \le \mathit{bound}}{[p]C^\star[\epsilon\colon \bigvee_{i \le \mathit{bound}} q_i]} \qquad \frac{[p]C_1[\epsilon\colon q_1] \qquad [p]C_2[\epsilon\colon q_2]}{[p]C_1 + C_2[\epsilon\colon q_1 \vee q_2]}$$

Derived Unrolling Rule *Derived Rule of Choice*

In the Unrolling Rule C^i is the i-fold sequential composition, where $C^0 = \texttt{skip}$ and $C^{n+1} = C; C^n$. The rule says that one of the things that iteration can do is execute its body i times. The Unrolling

Rule gives us a capability similar to symbolic bounded model checking [Clarke et al. 2004]. It is a simple and surprisingly effective way to discover some post-conditions for a loop, but in general stronger reasoning is needed (as provided by the Backwards Variant rule).

An important point is that $[presumption]c[\epsilon: result]$ expresses a reachability property that involves termination.

Every state in the result is reachable from some state in the presumption.

This reachability property does not imply that a loop must terminate on all executions. Instead, it establishes that enough paths terminate to cover all the states in the result assertion, while allowing that other paths might lead to divergence. The Backwards Variant proof rule is designed with this in mind. $p(\cdot)$ in the rule is a parametrized predicate (a function from expressions to predicates). The requirement that n be fresh in the rule means that it is a variable not free in C or $p(\cdot)$. The rule is similar to proof rules for proving program termination which appeal to a "variant" that decreases on each loop iteration [Apt 1981; Harel 1980], except that it reflects the backwards nature of this property: the variant, the parameter to p, goes down when executing backwards rather than forwards. A forwards variant of the rule is possible to formulate, but is not very useful in incorrectness logic, since $[\exists n. p(n) \wedge nat(n)]C^{\star}[ok: p(0)]$ is automatically true without any premises: from a state satisfying the post we can get back to the pre by choosing zero iterations and using 0 to witness $\exists n$.

Section 6.1 shows reasoning about a loop which contains both terminating and non-terminating executions using the Backwards Variant rule.

The reachability property is intuitively related to "liveness", which says that "something (good) will eventually happen" [Lamport 1977]. Entertainingly, in our case the "good" might be a bug. We can cast the property as stated in terms of something happening by reading a spec backwards: for every state in the result, it is possible to eventually reach a state in the pre by executing backwards. Reading forwards is trickier, but possible: if we explore executions from all the start states described by a pre, then eventually any given state in the result will be encountered. This exploration could be relative to an oracle to enumerate the pre-states, and it would dovetail and backtrack executions on the different states so divergence or nondeterminism from one state does not block other explorations. Note that the "eventually" here does not concern all paths, and appears related technically to what is known as an "existential liveness property" [Manolios and Trefler 2001]. In contrast, the over-approximate triple $\{pre\}C\{post\}$ describes a safety property, that "nothing bad will happen", where "bad" is delivering a final state outside the post, and this does not require termination of any paths.

The *Iterate zero* rule can be derived from the *Backwards Variant* rule together with *Consequence* and *Empty under-approximates* by choosing using $p(\cdot)$ of the form $\lambda n.\ if\ n > 0\ then\ false\ else\ p$. We have not attempted to formulate a minimal set of proof rules, but in Section 5.1 we shall prove a completeness theorem indicating that no proof rules are missing.

A version of the Backwards Variant rule is also present in reverse Hoare logic [de Vries and Koutavas 2011], as are most of our other rules for *ok* termination. They use a language with an infinitary syntax, and include an infinitary version of the disjunction rule. While we have not included an infinitary rule, the expressive power of infinite disjunctions is essentially used in selecting a variant assertion in the proof of completeness (Theorem 6).

We do not include the dual of the disjunction rule, the rule with $\wedge$ in place of $\vee$, because it is unsound for under-approximation. For the program

$$C \quad = \quad (\texttt{assume}\ x == 1\ ;\ x = 88)\ +\ (\texttt{assume}\ x == 2\ ;\ x = 88)$$

we have $[x == 1]C[ok\colon x == 88]$ and $[x == 2]C[ok\colon x == 88]$, but the conjunction $x == 1 \wedge x == 2$ is *false*, and we don't have $[false]C[ok\colon x == 88]$ because *false* is the post of C applied to *false*, and $x == 88$ is not a subset of *false*.

The rules of *Substitution* and *Constancy*, as well as *Consequence*, are important for adapting specifications for use in different contexts, as will be seen in examples in Section 6.4. The Local Variable rule is similar to one from Hoare logic [Apt 1981], except that we have $\exists y$ in the post-assertion where the Hoare rule omits the $\exists y$ while requiring $y \notin Free(q)$. In over-approximate reasoning one can move from q to $\exists y.q$ using the rule of Consequence, but that step is not available in under-approximate reasoning. If we did not include $\exists y$ then the rule would be too weak in many examples. The simple

Derived Local Variable Rule

$$\frac{[p]C[\epsilon\colon q]}{[\exists x.p]\texttt{local}\ x.C[\epsilon\colon \exists x.q]}$$

comes in handy. It can be derived using the Local Variable rule, Consequence, and Substitution II.

We finish this section with an exercise: deriving an axiomatic treatment of `assert` statements based on the encoding in Figure 2. The rule for + gives us these two quadruples:

$$[p \wedge B]\texttt{assert}(B)[ok\colon (p \wedge B)][er\colon false]$$
$$[p \wedge \neg B]\texttt{assert}(B)[ok\colon false][er\colon p \wedge \neg B]$$

For example,

$$[x \text{ is even}]\texttt{assert}(x \text{ is even})[ok\colon x \text{ is even}][er\colon false]$$
$$[x \text{ is odd}]\texttt{assert}(x \text{ is even})[ok\colon false][er\colon x \text{ is odd}]$$

In a symbolic execution tool, these sorts of axiom might be used in case we wish to call a theorem prover at branch points of if statements, in order to prune paths in the search for bugs.

A different derived axiom might be useful in some circumstances, in case we wish to postpone the theorem prover call. Namely, suppose we have a presumes assertion before an `assert` statement

$$[p]\texttt{assert}(B)[??]$$

and we don't know if p implies B or $\neg B$. We can push information to the result assertions, as follows:

$$[p]\texttt{assert}(B)[ok\colon (p \wedge B)][er\colon p \wedge \neg B].$$

For example,

$$[true]\texttt{assert}(x \text{ is even})[ok\colon x \text{ is even}][er\colon x \text{ is odd}].$$

5 SEMANTIC FOUNDATION

We provide a semantic foundation for the above. Readers who are more interested in seeing the proof rules in action can skip forward to the examples in the next section.

5.1 Relations

We presume we are given a set Σ of states, and predicates p, q, etc denote subsets of Σ. Much of our development works for arbitrary Σ, but to interpret the assignment statement we consider states to be of the form $\Sigma = [Variables \to Values]$ where a state maps variables to values. To reason about termination we further assume that *Values* contains the set of natural numbers.

Generic Semantics for arbitrary state sets Σ

$$[\![C]\!]\epsilon \subseteq \Sigma \times \Sigma$$
$$[\![B]\!] : \Sigma \to \mathit{Bool}$$

$$\begin{array}{rcl@{\qquad}rcl}
[\![\texttt{skip}]\!]ok & = & \{(\sigma,\sigma) \mid \sigma \in \Sigma\} & [\![\texttt{skip}]\!]er & = & \emptyset \\
[\![\texttt{error()}]\!]ok & = & \emptyset & [\![\texttt{error()}]\!]er & = & \{(\sigma,\sigma) \mid \sigma \in \Sigma\} \\
[\![\texttt{assume } B]\!]ok & = & \{(\sigma,\sigma) \mid [\![B]\!]\sigma = \texttt{true}\} & [\![\texttt{assume } B]\!]er & = & \emptyset \\
[\![C^\star]\!]\epsilon & = & \bigcup_{i \in Nat} [\![C^{\,i}]\!]\epsilon & [\![C_1 + C_2]\!]\epsilon & = & [\![C_1]\!]\epsilon \cup [\![C_2]\!]\epsilon \\
[\![C_1;C_2]\!]\epsilon & = & \{(\sigma_1,\sigma_3) \mid \exists \sigma_2.(\sigma_1,\sigma_2) \in [\![C_1]\!]ok \text{ and } (\sigma_2,\sigma_3) \in [\![C_2]\!]\epsilon\} & & & \\
 & & \cup \left(\mathit{if}\ (\epsilon = ok)\ \mathit{then}\ \emptyset\ \mathit{else}\ \{(\sigma_1,\sigma_2) \mid (\sigma_1,\sigma_2) \in [\![C_1]\!]er\}\right) & & &
\end{array}$$

Semantics of mutation and local variables

$$\Sigma = \mathit{Variables} \to \mathit{Values}$$
$$[\![e]\!] : \Sigma \to \mathit{Values}$$

$$\begin{array}{rcl@{\qquad}rcl}
[\![x = e]\!]ok & = & \{(\sigma, (\sigma \mid x \mapsto [\![e]\!]\sigma)) \mid \sigma \in \Sigma\} & [\![x = e]\!]er & = & \emptyset \\
[\![x = \texttt{nondet()}]\!]ok & = & \{(\sigma, (\sigma \mid x \mapsto v)) \mid \sigma \in \Sigma, v \in \mathit{Values}\} & [\![x = \texttt{nondet()}]\!]er & = & \emptyset \\
[\![\texttt{local } x.C]\!]\epsilon & = & \{((\sigma \mid x \mapsto v), (\sigma' \mid x \mapsto v)) \mid (\sigma,\sigma') \in [\![C]\!]\epsilon, v \in \mathit{Values}\} & & &
\end{array}$$

Fig. 4. Relational Denotational Semantics

Definition 1 (Post and Semantic Triples). *For any relation $r \subseteq \Sigma \times \Sigma$ and predicate $p \subseteq \Sigma$ define*

- *The post-image of r, $post(r) \in P(\Sigma) \to P(\Sigma)$:*

$$post(r)p = \{\sigma' \mid \exists \sigma \in p.\ (\sigma,\sigma') \in r\}$$

- *The under-approximate triple:*

$$[p]r[q] \text{ is true} \quad \mathit{iff} \quad post(r)p \supseteq q$$

- *The over-approximate triple (Hoare triple):*

$$\{p\}r\{q\} \text{ is true} \quad \mathit{iff} \quad post(r)p \subseteq q$$

First, we remark that this semantics validates the discussion in Section 2.

Theorem 2. *All of the listed properties in Figure 1 are true of this semantics.*

There is an alternate characterization of under-approximate triples in terms of the previously-stated reachability property, which is in fact an equivalence that is useful when doing semantic calculations.

Lemma 3 (Characterization). *The following are equivalent:*

(1) $[p]r[q]$ *is true*
(2) *Every state in the result is reachable from some state in the pre:* $\forall \sigma_q \in q.\ \exists \sigma_p \in p.\ (\sigma_p,\sigma_q) \in r$.

de Vries and Koutavas [2011] take (2) in this lemma as the semantics of triples, rather than the equivalent from Definition 1.

Relations give a particularly simple denotational (i.e., compositional) semantics of programs, as in Figure 4. In this semantics each program C is associated with two relations, $[\![C]\!]ok$ and $[\![C]\!]er$,

describing state transitions for successful and erroneous computations. The semantics of $C^\star$ is not literally compositional as presented, but $[\![C^i]\!]$ is defined in terms of $[\![C]\!]$ so it expands to a compositional definition. $[\![C^\star]\!]$ can also be presented as the least fixed-point of a function of the form $\lambda r.\texttt{skip} + C; r$ using lattices of relations ordered by graph inclusion.

This relational semantics treats divergence in an implicit way, in that there are no explicit $\bot$'s. An always-divergent program such as **`while`**`(true)`**`do`** `skip` denotes the empty relation, and the semantics satisfies the identity $C = C + \mathit{diverge}$. For readers familiar with subtleties that arise in the semantics of nondeterminism [Apt and Plotkin 1986; Cousot 2002], particularly when reasoning about the liveness property of total correctness (over-approximation plus termination), it might be surprising that we can use such a simple semantics. The relational semantics is known to work well for partial correctness, and it turns out that it is also suitable for studying the reachability property of under-approximate triples.

We can connect the relational semantics to the proof system of incorrectness logic by interpreting the proof-theoretic specifications in terms of the semantic triples.

Definition 4 (Interpretation of Specifications). *$[p]C[\epsilon\colon q]$ is true iff the semantic triple $[p]([\![C]\!]\epsilon)[q]$ holds.*

The proofs for the *ok* cases in the following soundness and completeness results can be as in reverse Hoare logic [de Vries and Koutavas 2011], and most can also be obtained from arguments in dynamic logic [Harel 1979]. We will briefly sketch some of these cases along with the *er* ones to keep the paper contained.

Theorem 5 (Soundness). *The relational semantics in Figure 4 validates all the rules in Figures 2 and 3: each axiom is true and each inference rule preserves truth.*

Proof: The soundness of the first two iteration rules follow from the true equivalence $C^\star \equiv \texttt{skip} + C^\star; C$. For the Backwards Variant rule we will use the Characterization Lemma. Assuming we have

σ satisfying $\exists n.p(n)$

we wish to show that

there is σ' satisfying $p(0)$ which $(C)^\star$ transforms to σ

Starting from σ, let j be a value of n which witnesses the existential. If $j = 0$ we are done, we can choose zero iterations of $(C)^\star$. Otherwise, the premise of the rule says that applying C once backwards gives us at least one σ_0 satisfying $p(j-1)$. If $j-1$ is 0 then we are done. Else, we repeat this step and iterate again backwards from σ_0 to obtain some σ_1 where $p(j-2)$ holds. Continuing this process will eventually in a finite number of steps deliver $p(0)$.

Amongst the variable rules we give the proof for the Local Variable rule; our rule, which follows Apt [1981], is slightly different from the one in reverse Hoare logic, which relies on alpha-renaming bound variables. Suppose $[p]C(y/x)[\epsilon\colon q]$ and $(\sigma_q \mid x \mapsto v, y \mapsto v_y) \in \exists y.q$. Then $(\sigma_q \mid x \mapsto v, y \mapsto v_2) \in q$ for some v_2. We execute $C(y/x)$ backwards and get $(\sigma_p \mid x \mapsto v, y \mapsto v_1) \in p$ by the Characterization Lemma. Since p is independent of y we can set y back to v_y and it will still be in p: $(\sigma_p \mid x \mapsto v, y \mapsto v_y) \in p$. This sequence of steps can be mimicked in the semantics of $\texttt{local}\, x.C$, stepping from $(\sigma_q \mid x \mapsto v, y \mapsto v_y)$ via backwards finalization to $(\sigma_q \mid x \mapsto v_2, y \mapsto v_y)$, then backwards via C to $(\sigma_p \mid x \mapsto v_1, y \mapsto v_y)$, and then via backwards initialization to $(\sigma_p \mid x \mapsto v, y \mapsto v_y)$. We already argued above that $(\sigma_p \mid x \mapsto v, y \mapsto v_y) \in p$, so this gives us the truth of $[p]\texttt{local}\, x.C[\epsilon\colon \exists y.q]$ by the Characterization Lemma.

The argument for the other variable rules similarly appeals to the Characterization Lemma. The arguments for + and ; are immediate from the semantic definitions. For assignment soundness

follows from the fact that Floyd's axiom expresses the strongest post-condition, which is therefore also an under-approximation. Similar is true for `skip`, `error()`, `x=nondet()` and `assume(B)`. ■

Next we need a condition on predicates. A subset $p \subseteq \Sigma (= \mathit{Variables} \rightarrow \mathit{Values})$ might depend on all variables, but we need to be able to select a fresh variable y not in a given assertion in order to apply the Local Variable rule in the proof of completeness. So, suppose that *Variables* is countably infinite, and define a predicate p to be *finitely-supported* if it is independent of all but a finite number of variables. Typically in applications, assertions will be written syntactically as finite expressions involving finitely many variables, so being finitely-supported is not a drastic restriction.

Theorem 6 (Completeness). *Every true triple involving finitely-supported predicates is provable.*

This theorem relies on us using semantic (subset of Σ) rather than syntactic predicates, with an oracle to decide implications. It does not contradict undecidability results, but rather confirms a sense in which no proof rules for programs are missing. [To cognoscenti: semantic predicates let us assume (or dodge the issue of) "expressiveness" [Cook 1978] used in completeness for Hoare logics, and requiring that *Values* contains the natural numbers builds in the "arithmetical" interpretation idea used in arguments for liveness proof rules [Harel 1979]. Our use of semantic predicates has precedent, e.g., in the metatheory of separation logic [Calcagno et al. 2007; Yang 2001] and embeddings of program logics in proof assistants [Nipkow 2002].]

Proof: We need to show that $[p]C[\epsilon\colon q]$ is provable on the assumption that it is true, and we do this by induction on the structure of C. The base cases for assignment, error and assume follow at once using the rule of consequence and the fact that the axioms express strongest post-conditions.

For choice, if $[p]C_1 + C_2[\epsilon\colon q]$ is true then we know that q is a subset of $\mathit{post}([\![C_1 + C_2]\!]\epsilon)p$, which is equal to $\mathit{post}([\![C_1]\!]\epsilon)p \vee \mathit{post}([\![C_2]\!]\epsilon)p$. By induction we know that each $[p]C_i[\epsilon\colon \mathit{post}([\![C_2]\!]\epsilon)p]$ is provable, and so the provability of $[p]C_1 + C_2[\epsilon\colon q]$ follows by the disjunction rule and consequence.

For sequencing, in case $\epsilon = ok$ the result follows by induction using the first sequencing rule, consequence, and that $\mathit{post}([\![C_1; C_2]\!]ok)p = \mathit{post}([\![C_2]\!]ok)(\mathit{post}([\![C_1]\!]ok)p)$. In case $\epsilon = er$ we use induction, consequence and the rule of disjunction together with the equality $\mathit{post}([\![C_1; C_2]\!]er)p = \Big(\mathit{post}([\![C_2]\!]er)(\mathit{post}([\![C_1]\!]ok)p)\Big) \vee \mathit{post}([\![C_1]\!]er)p$.

For iteration first we do the proof for $\epsilon = ok$. Supposing $[p](C)^\star[ok\colon q]$ is true, we define

$$p(n) = \{\sigma \mid \text{you can get back from } \sigma \text{ to some state in } p \text{ by executing } C \text{ backwards } n \text{ times}\}.$$

Note that $p(0) = p$ by this definition. From the definition of $p(n)$ it is evident that

$$[p(n) \wedge \mathit{nat}(n)]C[ok\colon p(n+1) \wedge \mathit{nat}(n)]$$

is true, and hence it is provable by induction hypothesis. We apply the Backwards Invariant rule and then Consequence using $q \Rightarrow \exists n.p(n)$, which is a true implication because of the Characterization lemma. This shows that $[p](C)^\star[ok\colon q]$ is provable. (We use n to describe the number of iterations in a similar way to Harel [1979], except that he appeals to Gödel encoding, and to de Vries and Koutavas [2011], who use an infinitary disjunction.)

Now, for $\epsilon = er$ we use the idea is that if an error is thrown then some number of successful iterations happens first, followed by error happening on thenext (last) iteraiton. We use the rule Iterate non-zero to deal with this case. So, suppose $[p](C)^\star[er\colon q]$ is true and define *frontier* to be the reachable states for normal termination; i.e., $\mathit{frontier} = \mathit{post}([\![C]\!]ok)p$. By the just-proven completeness case for iteration and normal termination, we know that $[p](C)^\star[ok\colon \mathit{frontier}]$ is provable. Now, $[\mathit{frontier}]C[er\colon q]$ must be true (note the absence of $\star$), or else the beginning assumption that $[p](C)^\star[er\colon q]$ could not be. By induction hypothesis we know $[\mathit{frontier}]C[er\colon q]$ is provable, and we can use Sequencing (normal) and Iterate non-zero to conclude that $[p](C)^\star[er\colon q]$ is provable.

Our final case is local variables. Suppose $[p]\texttt{local}\, x.C[\epsilon\colon q]$ is valid. We pick y independent of p and not free in C (p being finitely supported guarantees y exists), and claim that

(*) $q \subseteq \exists y.post(\llbracket C(y/x) \rrbracket \epsilon)p$.

Postponing proof of (*), $[p]C(y/x)[\epsilon\colon post(\llbracket C(y/x) \rrbracket \epsilon)p]$ is provable by induction hypothesis, and then we can use the Local Variable Rule together with (*) and the rule of Consequence to conclude that $[p]\texttt{local}\, x.C[\epsilon\colon q]$ is provable. [Note: induction hypothesis does not literally apply because $C(y/x)$ is not a subterm of $\texttt{local}\, x.C$; to be fully rigorous we would include an additional injective renaming component – if $[p]C[\epsilon\colon q](\vec{y}/\vec{x})$ is true then it is provable – to give us a stronger induction hypothesis; this is standard when doing proofs when substitution is involved (cf., [Apt 1981; Stoughton 1988]).]

We are left to prove (*). If q is empty we are done. If q is nonempty then the validity of $[p]\texttt{local}\, x.C[\epsilon\colon q]$ means that there exists $((\sigma_p \mid x \mapsto v_0, y \mapsto v), (\sigma_q \mid x \mapsto v_0, y \mapsto v)) \in \llbracket \texttt{local}\, x.C) \rrbracket$ with $(\sigma_p \mid x \mapsto v_0, y \mapsto v) \in p$ and $(\sigma_q \mid x \mapsto v_0, y \mapsto v) \in q$. This transition is witnessed (for some v_1, v_2) by the steps in the semantics of $\texttt{local}\, x.C$ from

> $(\sigma_p \mid x \mapsto v_0, y \mapsto v)$ via initialization to $(\sigma_p \mid x \mapsto v_1, y \mapsto v)$, via C to $(\sigma_1 \mid x \mapsto v_2, y \mapsto v)$, and via finalization to $(\sigma_q \mid x \mapsto v_0, y \mapsto v)$.

Since $(\sigma_p \mid x \mapsto v_0, y \mapsto v) \in p$ and p is independent of y we also have $(\sigma_p \mid x \mapsto v_0, y \mapsto v_1) \in p$, mimicking the initialization step above. We can mimick the remaining steps by moving from

> $(\sigma_p \mid x \mapsto v_0, y \mapsto v_1)$ via $C(y/x)$ to $(\sigma_q \mid x \mapsto v_0, y \mapsto v_2) \in post(\llbracket C(y/x) \rrbracket \epsilon)p$.

This shows $(\sigma_q \mid x \mapsto v_0, y \mapsto v) \in \exists y.post(\llbracket C(y/x) \rrbracket \epsilon)p$, and (*) follows. ■

The rules of Substitution and Constancy from Figure 3 were not needed for the completeness proof. They are used in Section 6.4 to adapt specifications to calling contexts. The substitution rules are suitable for use with parameterless procedures, but additional rules are needed to deal with parameters. This has been a notoriously subtle problem-area in Hoare logics [Apt 1981; Cook 1978], and a thorough treatment of incorrectness logic with procedures would be worthwhile.

5.2 Predicate Transformers

Predicate transformers, functions that map predicates to predicates, are a fundamental semantic tool in program logic and analysis. In this section we find that:

(1) forwards transformers, which form the basis for program verification and (especially) program analysis tools, are well behaved in incorrectness logic;
(2) backwards transformers, which have been used extensively in tools and in program refinement [Back and von Wright 1998; Dijkstra 1976], do not always exist for incorrectness logic (at least in standard forms).

Forwards Transformers. The post-condition operator $post(r)p$ can be given an alternate characterization as the strongest predicate satisfying the Hoare triple $\{p\}r\{q\}$. This is usually called the strongest post-condition, but we will call it strongest-over-approximate post to distinguish from the opposite notion, the weakest under-approximate post.

Definition 7. *For $r \subseteq \Sigma \times \Sigma$:*

- $StrongestOverPost(r)p \;=\; \bigwedge\{q \mid \{p\}r\{q\}\ holds\}$
- $WeakestUnderPost(r)p \;=\; \bigvee\{q \mid [p]r[q]\ holds\}$

Proposition 8. $StrongestOverPost(r) \;=\; WeakestUnderPost(r) \;=\; post(r)$

This proposition gives a starting point for defining under-approximating program analyses. We explain this in the key cases of iteration and choice, which encapsulate the issues of undecidability and path explosion that program analyses need to deal with.

The strongest post for iteration can be defined by appeal to loop invariants.

$$post([\![C^\star]\!]ok)p \quad = \quad \bigwedge\{I \mid p \Rightarrow I \wedge \{I\}C\{I\} \text{ is true}\}$$

This equation can be proven, in the left-to-right direction, from soundness of the Hoare proof rule for invariants, and from right-to-left by selecting I to be the set of all states reachable in some number of executions of C from p. A different definition is possible with under-approximate triples.

$$post([\![C^\star]\!]\epsilon)p \quad = \quad \bigvee_{i \in Nat}\{q \mid [p]C^i[\epsilon\colon q] \text{ is true}\}$$

That this equation is true can be proven by selecting $q = \{s\}$ for each of the reachable states s, and observing that when s is reachable we must get there by some number i of iterations.

These two (equivalent) definitions of the iteration predicate transformer present different compromise possibilities that reasoning tools can take in the face of undecidability. For over-approximation, if we discover a loop invariant then the first equation tells us that we will over-approximate the post [Cousot and Cousot 1977]. This is nontrivial, and tends to require the invention of remarkable abstract domains and widening operators which are generally not yet available in every situation. For under-approximation, the second equation tells us that the simple option of limiting the loop unrollings can be used

$$\underline{post}([\![C^\star]\!]\epsilon)p \quad = \quad \bigvee_{i \leq bound}\{q \mid [p]C^i[\epsilon\colon q] \text{ is true}\}.$$

This is an alternate predicate transformer semantics where the calculated post is a subset of the exact one. While blunt, this approach does not require human invention for every new analysis problem, and presents less of a bottleneck as regards the availability of possibly scarce human expertise.

The idea that you can under-approximate via bounded loop unrollings is well known. We have made these points only to exemplify how incorrectness logic's principles fit well with (under-approximate) reasoning intuition, in a complementary way to how Hoare logic fits for correctness. Of course, bounded unrollings are not sufficient in all situations, an issue we will explore in the Section 6.1.

Turning now to nondeterministic choice, we have

$$post([\![C_1 + C_2]\!]\epsilon)p \quad = \quad post([\![C_1]\!]\epsilon)p \ \vee \ post([\![C_2]\!]\epsilon)p.$$

Even if loops are bounded to remove infinite executions, we can still obtain an explosion of paths when + is used to model conditionals.

A standard technique in program analysis is to avoid disjunctions by keeping a single or few abstract states at each program point. Theoretically, this can be seen as selecting a "join operator" $\overline{\vee}$ that over-approximates disjunction: $p \vee q \Rightarrow p \,\overline{\vee}\, q$. For under-approximate reasoning we need to do the reverse: consider $\underline{\vee}$ where $p \,\underline{\vee}\, q \Rightarrow p \vee q$. Then, we can define

$$\underline{post}([\![C_1 + C_2]\!]\epsilon)p \quad = \quad \underline{post}([\![C_1]\!]\epsilon)p \ \underline{\vee} \ \underline{post}([\![C_2]\!]\epsilon)p$$

The general point is that a predicate transformer semantics $\underline{post}$ with just these clauses in place of the standard ones produces correct under-approximate post-states.

Often in program analysis a set of abstract states is used to represent a disjunction, and a representation of the disjunction of two of the sets is their union. In this case, a valid implementation of $\underline{\vee}$ is simply to trim the resulting union so that it stays smaller than a given threshold. That is the approach taken in Infer.Pulse, a program analyzer for object lifetimes in production at Facebook.

There is much more to producing a program analysis than changing the predicate transformer semantics in this way. In particular, compositional reasoning, where pre- as well as post-assertions are inferred, is important for scalability and incrementality[Calcagno et al. 2011; Distefano et al. 2019; O'Hearn 2018]. We will not define a compositional analysis method in this paper, but will

comment on the connection between under-approximate triples and the concept of procedure summary during our discussion in the next section.

Backwards Transformers. Where the forwards transformers for under-approximation are similarly well behaved as their over-approximate cousins, the same is not true for backwards.

Fact 9. *Valid presumptions need not exist: Given a relation r and assertion q, there need not exist any p such that $[p]r[q]$.*

For example, there is no p making $[p]x = 41[ok: true]$ hold. One might try to remedy this problem by adding a special predicate (say, $\top$) to assign as the pre when no appropriate state set exists. However, this simple tack is not enough: even when valid presumptions exist, there need not be a smallest (strongest) one. The example $C = (\texttt{assume}\ x==1\ ;\ x = 88) + (\texttt{assume}\ x==2\ ;\ x = 88)$ from earlier has $x==1$ as $x==2$ are valid pre's for result $\texttt{ok}:\texttt{x==88}$, but their conjunction is not.

While strongest under-approximate presumptions do not exist in general, this does not mean that backwards reasoning is impossible to use. We can use standard backwards transformers, which were designed to discover over-approximate triples, to generate pre-assertions, and then reason forwards to generate sound under-approximate post-assertions. We illustrate with an example in Section 6.3.

5.3 Other Characterizations

We described the under-approximate triple directly using predicate transformers, and gave an equivalent characterization which we called the reachability property. de Vries and Koutavas go the other way around and take the reachability property as the definition [de Vries and Koutavas 2011] . Other equivalent characterizations are possible using a number of logical theories that have been developed over the years.

Relational and Kleene Algebra. We can describe the under-approximate triple using the theory of binary relations, and the dual of an encoding of over-approximate triples suggested by Hoare in work on Concurrent Kleene Algebra [Hoare et al. 2011]: he suggested to read $\{p\}r\{q\}$ as $p; r \sqsubseteq q$, and we will reverse $\sqsubseteq$. Instead of working at the algebraic we will stick with the particular model of relations.

To formulate the connection we use the following mapping of predicates to binary relations.

Definition 10. *If $p \subseteq \Sigma$ then let $make(p) = \{(\sigma, \sigma') \mid \sigma' \in p\}$.*

Fact 11. *Let ";" denote composition of binary relations in diagrammatic order. Suppose $r \subseteq \Sigma \times \Sigma$ is a relation and $p, q \subseteq \Sigma$ are predicates.*

- *$\{p\}r\{q\}$ holds iff $make(p); r \subseteq make(q)$.*
- *$[p]r[q]$ holds iff $make(p); r \supseteq make(q)$.*

Based on this characterization we can make a further connection to the theory of Kleene Algebra with Tests (KAT, [Kozen 2000]). A model of KAT is an algebraic structure (an idempotent semiring with additional axioms) including operations for sequential composition and nondeterministic choice, and equipped with an embedding of a Boolean algebra that sends disjunction to + and conjunction to sequential composition. A standard model of KAT takes binary relations on a state space Σ as the carrier of the algebra, and embeds the Boolean algebra $P(\Sigma)$ into it via a mapping $assume(\cdot)$ that sends a predicate p to the subset of the identity relation with domain p. Then $make(p)$ corresponds to the sequential composition $\top; assume(p)$, where $\top$ is the everywhere-true binary relation. The under-approximate triple can then be represented in KAT as $\top; assume(p); r \sqsupseteq \top; assume(q)$, where $a \sqsupseteq b$ is defined as $a + b = a$.

Dynamic Logic. Dynamic logic is a modal logic with an operator $\langle C\rangle p$ which says that, from the current state, it is possible to transition via program C to a state satisfying p [Harel et al. 2000]. The dual, $[C]p$, says that all ways of transiting via C satisfy p. In terms of our relational semantics:

Definition 12. *Suppose $r \subseteq \Sigma \times \Sigma$ is a relation and $p \subseteq \Sigma$ is a predicate.*

- *$\langle r\rangle p$ is true in state σ iff $\exists \sigma' \in p.\ (\sigma, \sigma') \in r$.*
- *$[r]p = \neg\langle r\rangle\neg p$.*
- *r^{-1} denotes the reversal of relation r.*

r^{-1} gives us an alternate way to describe the *post*() predicate transformer, and this provides another way to describe the over-approximate and under-approximate triples.

Fact 13.
- $\langle r^{-1}\rangle p \ = \ post(r)p$
- $\{p\}r\{q\}$ *holds iff* $q \Leftarrow \langle r^{-1}\rangle p$ *holds (iff* $p \Rightarrow [r]q$ *holds)*
- $[p]r[q]$ *holds iff* $q \Rightarrow \langle r^{-1}\rangle p$ *holds*

Dynamic logic is a general logic that includes negation, conjunction, nestings of modal operators, etc. The principles of Figure 1 can be seen as true formulae of dynamic logic.

There has been work on reasoning about incorrectness using dynamic logic [Rümmer and Shah 2007], but it is quite different to that here. They focus on disproving total correctness statements for deterministic programs, which is to say proving formulae of the form $\neg(pre \Rightarrow \langle r\rangle post)$, rather than on under-approximation and on the connection to problems of program analysis.

It seems likely that most if not all of our proof rules for *ok* termination, as well as soundness and completeness results (and incidentally those of de Vries and Koutavas [2011]), could be derived by embedding into dynamic logic. In particular, dynamic logic gives a pleasant explanation for the backwardness in the proof rule for variants. Harel [1979] formulates existential reasoning with formulae like $p(n+1) \wedge nat(n) \Rightarrow \langle r\rangle p(n) \wedge nat(n)$. In terms of the encoding of total correctness for deterministic programs as *pre-cond* $\Rightarrow \langle r\rangle$*post-cond*, this describes how a variant decreases on each forwards loop iteration. But if we replace the r by r^{-1}, and regard the left-hand side of the implication as the result assertion and the right as the presumption, then the formula for decreasing n is equivalent to the premise of the Backwards Variant rule. Harel [1979] does not include r^{-1}, but it is studied elsewhere where it is called the "converse" operator [Harel et al. 2000].

From this perspective, the main technical contribution of the present article would seem to be the treatment of error assertions. Of course, the larger contribution is conceptual, involving the relevance to proving the presence of bugs as well as connections to principles in static analysis tools, including the work in the next section.

6 REASONING WITH THE LOGIC

In this section we investigate reasoning patterns that can be expressed with incorrectness logic. We consider examples motivated by existing tools such as DART, KLEE, CBMC and Infer, as well as ones that might lead in future to increased expressiveness or efficiency in reasoning. We stress, though, that we are not claiming at this time that incorrectness logic leads to better practical results than these mature tools; this is an exploration to give insight into the theory, and a basic test of a potential foundational formalism is how it expresses a variety of patterns that have arisen naturally.

We do not include a formal treatment of procedures in this paper, but we will use the proof system in a way that lets us reason from hypotheses for parameterless procedures, as in

$$[p]\texttt{foo}()[ok\colon q][er\colon s] \vdash [p']C[ok\colon q'][er\colon s']$$

where calls to `foo()` can occur within C. Where we reason about the body of `foo()` to establish a spec, we don't need to revisit its body at call sites. This *principle of reuse* is fundamental in program

logics, and also in summary-based program analysis [Reps et al. 1995]. (To people with a program analysis background, when you see a presumes/achieves spec you might think: "summary"!)

6.1 Reasoning about Loops

Figure 5 contains several loops, and client programs that use them. The first loop, `loop0()`, is a variant on an example `testme_inf` of Cadar and Sen [2013], which was used there to demonstrate a loop with an indeterminate number of iterations.

When we omit mention of a `presumes` assertion, as throughout this figure, the understanding is that it defaults to `true`.

We have given an `achieves` result for `loop0()`. Postponing for a moment discussion of how we can actually prove this result assertion, we can use it during reasoning about `client0()`. Immediately after the call to `loop0()` on line 14 we use incorrectness logic's rule of consequence with an implication backward from the **`if`** condition

$$\frac{[\mathit{true}]\texttt{loop0()}[\mathit{ok}\colon x >= 0] \qquad x >= 0 \iff x == 2,000,000}{[\mathit{true}]\texttt{loop0()}[\mathit{ok}\colon x == 2,000,000]}$$

and we use the sequencing rule to conclude that an error is reachable, which is recorded in the `achieves` of `client0()` at line 13.

But how can we obtain the spec of `loop0()`? It is very easy: we unwind the loop once. Here is the unrolling, with assertions at intermediate points to hint at how to construct a proof.

```
1   [x==0]
2        if (n>0) {
3           [x==0 && n>0]      x=x+n;  n=nondet();  [x>0]
4         } else
5         { [x==0 && n<=0]    skip;
6         }
7        [x>0  ||  (x==0 && n<=0)]
8         assume (n<=0);
9        [(x>0 && n<=0) || x==0 && n<=0)]
10  [ok: x>=0 && n<=0]
```

The disjunction at line 9 of this proof outline contains one assertion for each branch of the **if**, and the assertions at lines 9 and 10 are logically equivalent; they follow the condition for loop exit at line 8. If we put this together with the first two statements in `loop0()`, then existentially quantify n using the local variable rule, we obtain the triple $[\mathit{true}]\texttt{loop0()}[\mathit{ok}\colon x >= 0]$ using the Unrolling Rule plus plus the Rule of Consequence to strip $\exists n$.

Now, incorrectness logic tells us that `[ok: x>=0]` is a correct under-approximate result, but how do we know how many paths our reasoning covers? After all, we only considered one loop unrolling before reaching our conclusion. In this case, we happen to be in the lucky situation where the post-assertion is also true for over-approximation. That is, we have

$$[\mathit{true}]\texttt{loop0()}[\mathit{ok}\colon x >= 0] \quad \text{and} \quad \{\mathit{true}\}\texttt{loop0()}\{\mathit{ok}\colon x >= 0\}$$

and it would not be difficult for a tool to discover an invariant $x >= 0$ sufficient to prove the over-approximate triple. So, the under-approximate result assertion covers all terminating paths.

Cadar and Sen [2013] use this example to illustrate the challenge of needing to cover potentially infinitely many paths, and they rightly say "In practice, one needs to put a limit on the search, e.g., a timeout, or a limit on the number of paths, loop iterations, or exploration depth." Interestingly, for this example we have shown a precise sense in which we can get a post that is as general as can be by reasoning in incorrectness logic with one loop unrolling (though we do appeal to over-approximate reasoning to confirm that this general assertion does indeed cover all paths).

```
1  int x;
2
3  void loop0()
4  /* (default presumes is "true" when not specified)
5     achieves: [ok: x>=0 ]    */
6    { int n = nondet();  x = 0;
7       while (n > 0) {
8         x = x+n;
9         n = nondet();
10   }   }
11
12   void client0()
13   /*  achieves: [err: x==2,000,000 ]   */
14     { loop0();
15       if (x==2,000,000) {error();}
16     }
17
18 void loop1()
19 /*  achieves1: [ok: x==0 || x==1 || x==2 || x==3 ]
20     achieves2: [ok: x>=0]   */
21   { x = 0;
22     Kleene-star{
23       x = x+1;
24   }  }
25
26 void client1()
27 /*  achieves: [er: x==2,000,000]  */
28    { loop1();
29      if (x==2,000,000) {error();}
30    }
31
32   void loop2()
33 /* achieves: [er: x==2,000,000]   */
34   { x = 0;
35     Kleene-star{
36       if (x==2,000,000) {error();}
37       x = x+1;
38    } }
```

Fig. 5. Iteration Examples

We can't always be so lucky as to get such a general result after one iteration. Consider `loop1()` in the figure, which starts with `x=0` and then increments `x` an arbitrary number of times before exiting. We use `Kleene-star(..)` instead of the $(.)^\star$ notation when showing code. There are infinitely many paths through `loop1()`, and the loop is not guaranteed to terminate. Using the Unrolling rule we can obtain post-conditions for any of the finite unrollings of the program. `achieves1` shows the obtained result for three unrollings. While it does not cover all of the behaviours of the program, it is indeed a sound under-approximation.

We can think of the pair `true/achieves1` as a procedure summary for `loop1()`. This way of reasoning about individual paths to obtain an under-approximate assertion is related to the method of Godefroid [2007] to produce procedure summaries for the purpose of scaling symbolic execution.

Next let's consider the function `client1()` that calls `loop1()`. The assertion `achieves1` is not enough to prove that an error can occur. We could boundedly unroll the loop $2,000,000$ times to prove the bug in `client1()`, but then this unrolling would not cover a different client program where the bug happens at $4,000,000$. If, however we are lucky enough to have the more

general `achieves2: [ok: x>=0]` for `loop1()`, then we can strengthen it using the implication $x == 2,000,000 \implies x >= 0$, as we did with `client0()` above, to prove the error.

We can obtain the better under-approximate assertion `x>=0` for `loop1()` using the following instance of the Backwards Variant rule where $p(n)$ is $x == n$.

$$\frac{[x == n \wedge nat(n)]\mathsf{x = x + 1}[ok\colon x == n + 1 \wedge nat(n)]}{[x == 0](\mathsf{x = x + 1})^\star[ok\colon \exists n.x == n \wedge nat(n)]}\ n \text{ fresh}$$

The premise can be derived using the assignment axiom and the rule of consequence,

The use of the result assertion `achieves2` to prove `client1()` is related to the use of under-approximation to obtain faster counterexamples in CBMC [Kroening et al. 2015].

The Backwards Variant rule does not describe cases where an error occurs during execution of a loop body. But, as describe before, we can use it to describe the "frontier", the states obtained by some number of iterations before a last, erroneous, one, and then use the *Iterate non-zero*.

For example, suppose we move the assert statement inside the loop, as in the function `loop2()` in Figure 5. We can prove this in two stages. First, we show

$$[x == 0](\mathsf{Body})^\star[ok\colon 0 <= x <= 2,000,000].$$

This can be proven using the Backwards Variant rule taking $p(n)$ to be $0 <= x <= 2,000,000 \;\&\&\; x == n$. We then use the rule of consequence to shrink the post-assertion, giving

$$[x == 0](\mathsf{Body})^\star[ok\colon x == 2,000,000].$$

Next, the assert rule plus the short-circuiting rule for Sequencing give us

$$[x == 2,000,000]\mathsf{Body}[er\colon x == 2,000,000].$$

These specs line up for sequential composition, to give us

$$[x == 0](\mathsf{Body})^\star; \mathsf{Body}[er\colon x == 2,000,000]$$

and then *Iterate non-zero* yields

$$[x == 0](\mathsf{Body})^\star[er\colon x == 2,000,000].$$

This proves `loop2()` from Figure 5.

Reasoning with loop variants sometimes allows more compact and general under-approximate assertions to be derived than would be obtained by taking posts for each of many individual paths.

6.2 Conditionals, Expressiveness and Pruning

Conditional statements cause challenges for reasoning tools. The use of Boolean conditions that are difficult for current theorem provers to deal with causes expressiveness issues, and the number of paths that can occur through the code, even without loops or recursion, can lead to inefficiency.

Conditions that are Difficult for Symbolic Reasoning. Consider the code for `client()` in Figure 6. The function `difficult()` is assumed to be difficult for a reasoning tool to treat precisely. Cadar and Sen [Cadar and Sen 2013] present an example using multiplication, which goes beyond the decidable subsets of arithmetic encoded in automatic theorem provers. Other examples might be hash functions, unknown or binary code, or code that is obfuscated in a way that purposely makes automated analysis difficult. How can we obtain a correct under-approximate post?

Dynamic symbolic execution, as represented by DART, EXE, and KLEE [Cadar and Sen 2013], takes a particularly pragmatic approach based on the principle: when symbolic reasoning is difficult, replace a symbolic variable with a concrete value. In the example above, let's consider replacing x by the number 7. By this tack, we can obtain `achieves1: [y==49 && x==1]`, and we do this by shrinking the post-assertion. That is, the assume statement corresponding to the true branch

```
39  int x,y;
40
41  int difficult(int y)
42   {  return (y*y); /* or, return hash(y) ... or, unknown code */
43   }
44
45   void client()
46   /*achieves1: [ok: y==49 && x==1]
47     achieves2: [ok: exists z. (y==difficult(z)&& x==1)||(y!=difficult(z)&& x==2)]
48     achieves3: [ok: x==1 || x==2]   */
49     { int z = nondet();
50       if (y == difficult(z))
51         {x=1;}
52       else
53         {x=2;}
54     }
55
56  void test1()
57  /*achieves2: [ok: exists z. (y==difficult(z)&& x==1)||(y!=difficult(z)&& x==2)]*/
58    { client();
59       if (x==1 || x==2) { error(); }
60    }
61
62  void test2()
63     { client();
64       if (x==2) {error();}
65     }
```

Fig. 6. Conditional Examples

of the if statement is `assume(y==difficult(z))` with post-assertion `[y==z*z]`, and we have an implication $y == z * z \iff y == z * z \wedge z == 7$ so using the incorrectness logic consequence rule we obtain a post with $y == 49$. Thus, the logical principle of *shrinking the post* gives us sound triples corresponding to the pragmatic analyzer principle of *concretizing symbolic values*.

Another sound approach to difficult-to-reason-about Booleans is to record information lazily, in the post-assertion, in the hopes that the code might be used in contexts where the difficulty is immaterial. `achieves2` on line 47 does this, and can be used in `test1()` to prove an error.

A final approach is to record disjuncts for both branches while discarding the difficult bits, as in `achieves3`. This is unsound: we need to keep information correlating y with x as, e.g., the final state $(x \mapsto 1, y \mapsto 3)$ is not reachable but satisfies `x==1 || x==2`. Further, if we use `achieves3` when reasoning about `test2`, then we would wrongly conclude that there could be an error.

Though unsound, this approach has been used for pragmatic reasons in SMART [Godefroid 2007], descendant of DART and a precursor of SAGE [Godefroid et al. 2008] at Microsoft, and in Infer.RacerD, a data race detector in production at Facebook [Blackshear et al. 2018]. SMART trips a flag indicating this unsoundness when it occurs, and then uses procedure summaries that no longer under-approximate when generating further test cases. Overall soundness is rescued by appeal to concrete execution. RacerD on the other hand adjusts its soundness claim [Gorogiannis et al. 2019]: it is an under-approximation of an over-approximation, where the over-approximation arises by replacing Booleans it doesn't understand with nondeterministic choice. So the reasoning is sound when **`if`** is replaced by `+`. It seems as if the same under-of-over approach could be taken theoretically to prove an additional result about SMART's summaries.

So, an unsound choice, `achieves3`, is not unreasonable. Localized unsound decisions might be made by a tool, which represent assumptions that can be used as input to further sound steps which

are justified by logic. From this perspective, the role of logic is not to produce iron-clad unconditional guarantees, but is to clarify assumptions and their role when making sound inferences.

Dropping disjuncts. We have described earlier how removing a disjunct is a sound step in incorrectness logic. Infer.Pulse takes the blunt approach of having two limits, the number of loop unrollings and the number of disjunctions; it simply drops disjuncts when they exceed the limit. Jules Villard communicated a recent example of running Pulse on a C++ codebase inside Facebook with 100s of thousands of lines, comparing a 20 disjunct limit against 50 disjuncts; in both cases with the unrolling limit was set to 5. The result was that the 20 disjuncts case was ~2.75x wall clock time faster, ~3.1x user time faster, and it found 97% of the issues that the 50 disjuncts case found.

The point is that engineering and not only philosophical considerations are relevant to the question of when it is worth the energy cost, computer time or human time to address the other 3%. The choice is not a a binary one between fast and slow; e.g., we we might deploy them at different times, the fast as part of code review and the slow later in the software development process (and run less frequently). The specific numbers such as ~2.75x are not important, and we caution the reader not to read too much into them. But, the method of dropping disjuncts should not be dismissed as unprincipled; it gives engineers a knob to turn when considering how and when to deploy a tool.

While setting a limit can be useful, sometimes one can manage to always keep exactly one disjunct, obtained in a smarter way than by pruning. Our looping examples earlier did this for specific programs, by manually-provided assertions that cover infinitely many paths. Automatic reasoning in over-approximate tools often aim for a join operator that produces a single state over-approximates a disjunction. The Infer.RacerD data race detector [Blackshear et al. 2018] has an example for under-approximate joining.

RacerD assumes that the code it runs on will mostly use locking and unlocking in a bracketed way. It exploits this by keeping track of the number of times a (re-entrant) lock has been locked, and then taking the maximum of lock counts at join points (e.g., at the end of conditionals). This decision leads to false negatives, missing bugs when code of the form `maybe_lock; write(x)` can race with `lock; read-or-write(x)`, but it does not lead to false positives.

An example like this is in Figure 7. There, the achieves of `maxing()` is such that reasoning with it will not reveal the error in `false_negative()`, but it is sufficient to find the error in `true_positive()`. We can obtain the `achieves` assertion instead of `[locked==0 || locked==1]` using the typical incorrectness logic reasoning that shrinks the result assertion.

Aside: While the ability to drop disjuncts allows one of the practical difficulties of over-approximate reasoning – the need to cover all paths – to be avoided, one might wonder whether the converse inability to drop conjuncts when reasoning along a path might lead to practical difficulties. This is a legitimate question, but there are a number of remediations. First, localization via the Constancy and Local Variable rules lets us limit the information in the post. This is (implicitly) used in SMART, where under-approximate summaries are represented concisely by tracking modified values without explicitly writing the full post. Second, one can consider doing under-approximation after an over-approximation step, as in RacerD and Infer.Pulse. Over lets you ignore values, and *then* reason under, leading to underapproximate-modulo-assumptions. The theoretical restriction on dropping information along a path is not an ultimate blocker for under-approximate reasoning.

6.3 Symbolic Reasoning and Flakey Tests

A "flaky test" is one that, due to nondeterminism, can give different answers on different test runs. Conversely, we call a test "sturdy" if gives the same answer on all runs of a program. Flakiness

```
int locked;

 void maxing()
 /* achieves:  [ok: locked==1] */
   { int z=nondet();
     if (z)
       {locked=0;}
     else
       {locked=1;}
   }

void false_negative()
/* achieves:  [er: false]  and  [ok: locked==1] */
  { maxing();
    if (locked==0) { error(); }
  }

 void true_positive()
 /* achieves:  [er: locked==0] */
  { maxing();
    if (locked>0) {locked=locked-1;};
    if (locked==0) { error(); }
  }
```

Fig. 7. Maxing Out

```
void foo()
/*  sturdy presumes [x is even] , sturdy achieves [er: x is even],[ok: false]
    flaky presumes  [x is odd] ,  flaky achieves [er: x is odd],[ok: x is odd]  */
 { if (x is even) {error();}
   else { if (nondet()) { skip;} else {error();}
     }
 } }

 void flaky_client()
 /*  flaky achieves: [er: x==3 || x==5]    */
 { x = 3;
   foo();
   x = x+2;
   assert(x==4);
  }
```

Fig. 8. Sturdy and Flaky

is increasingly considered to be a problem in deployments of testing in industry [Harman and O'Hearn 2018]. We develop an example illustrating a logical account of sturdy and flaky tests.

We are going to use classic notions from backwards reasoning. If π is a path in a program, then

- $wp(\pi)q$ describes states for which execution of π is guaranteed to terminate and satisfy q;
- $wpp(\pi)q$ describes states for which execution of π is possible to terminate and satisfy q.

wp is Dijkstra's weakest pre-conditon [Dijkstra 1976]. wpp is the weakest *possible* pre-conditon, a predicate transformer described by Hoare [1978] for what he referred to as "possible correctness". (Note: wp cannot be defined using the relational model of this paper.) We will use these ideas to obtain pre-assertions, followed by forwards reasoning to obtain under-approximate post-assertions.

Procedure `foo()` in Figure 8 contains three paths, and $wp($`assume x is even`$)$*true* for the path prefix of the first of them gives us an input condition which forces execution to reach `error()` on line

4. We install this as the (sturdy) presumes. Forward reasoning obtains this as the under-approximate error post as well. The sturdy presumes assertion catches one bug but misses two others: the second `error()` statement at line 5 in `foo()`, and the assert at line 14 in `flaky_client()`.

The other two paths in in Figure 8 end in `skip` and `error()`, and each has a prefix π equivalent to `assume(x is odd); b = nondet(); assume b` for a variable `b` considered local to the trace. $wp(\pi)true$ is *false*, indicating that there is no input state which can force execution down either path. So, we appeal to the weakest *possible* pre-condition, which describes the pre-states that can possibly lead to the post, and we find that $wpp(\pi)true$ = `x is odd`. We install this as what we label the flaky pre, and forwards reasoning then gives us the two achieves assertions. The flaky error assertion reveals the potential bug of the second `error()` in `foo()`, and the ok assertion is used in `flaky_client()` to reveal that the assert statement can fail.

Note that even though we started reasoning backwards from *true*, we did not keep *true* as the achieves assertion (the post). The reason is that *true* is not a sound under-approximate post. The effect of this could be seen if we replaced the statement `assert(x==4)` by `assert(x==5)` and simultaneously the flaky achieves `[ok: x is odd]` with `[ok:true]`. Then, we would wrongly conclude that `assert(x==5)` in the altered `flaky_client()` could fail. When we use backwards reasoning to generate a pre starting from a given post, we need also to be careful to update that post if we are to re-use the reasoning in other contexts.

Test runs starting from inputs satisfying the flaky presumes are not reproducible, unless our testing engine were to record all nondeterministic choices; an expensive matter for a testing tool. From a logical point of view there is an interesting possibility: even if we have nondeterminism at runtime, a *proof* of incorrectness can be checked again and again in a deterministic fashion. We wonder whether symbolic proofs of bugs might help with the practical problem of flaky tests.

6.4 Composition and Procedural Abstraction

This section discusses reasoning that chains several procedure calls together.

If we have a path without procedure calls – say a sequential composition of assignment, assume and assert statements – then we can perform strongest post-condition reasoning, which is also therefore under-approximate. Collecting together such pre/post pairs for a number of paths can allow us to form an under-approximate summary for a procedure. But then reasoning with a path containing a procedure call should use these summaries, and there is subtlety in doing so soundly.

We illustrate with the specs in Figure 9. `x>0` is the strongest post of `x=x+1` given pre `x>=0`, and so is therefore under-approximate too. Let's try to reason about the sequential composition in `client()` using `presumes1/achieves1` from line 2. In Hoare logic we could do this with the rule of Consequence and $x > 0 \Rightarrow x >= 0$, giving $\{x >= 0\}$`inc();inc()`$\{x > 0\}$. The post here is a correct over-approximation, but it is not the strongest post of the sequential composition, and it is not a correct under-approximation. If we were to use this `wrong achieves 1` at line 9 while reasoning about `test()` then we would wrongly conclude that an error is possible.

What is happening here is that there is a discrepancy between the strongest post-condition of the code for `inc();inc()` and what can be inferred from the specs, even though `achieves1` at line 2 is the strongest post of a single call. That is, `wrong achieves1` *is* the strongest post for the composition $r; r$ for greatest relation r satisfying $\{i >= 0\}r\{x > 0\}$, but it is not for `x=x+1`: the greatest relation, what we know from the spec, does not under-approximate the code for `inc()`.

The moral of this story is that when proving that bugs can occur we need to be careful about what strongest post-condition reasoning gives us in the presence of procedures. Incorrectness logic provides this care. When faced with the above scenario, it tells us not to make the problematic inference, that we should insist instead on an implication in the reverse direction. That is, we ask for $x > 0 \Leftarrow x >= 0$ and when we see it is false, we block reasoning about the sequential composition.

```
1   void inc()
2   /*  presumes1:    [x>=0]          ,  achieves1: [ok: x>0]
3       presumes2:    [x==m && m>=0] ,  achieves2: [ok: x==m+1 && m>=0]  */
4    { assert(x>=0);
5      x=x+1;
6    }
7
8    void client()
9    /* presumes1: [x>=0]          ,  wrong achieves1:[x>0]
10      presumes2: [x==m && m>=0] ,  achieves2: [ok: x==m+2 && m>=0]    */
11   { inc();
12     inc();
13   }
14
15   void test()
16   /* wrong achieves1: [er: x==1]
17      achieves2: [er: false]     */
18   { x = 0;
19     client();
20     assert(x>=2);
21   }
```

Fig. 9. Specs for Composition

So, incorrectness logic prevents the unsound (for bug catching) inference. Furthermore, it shows that we need a different spec to draw useful conclusions about the composition.

A different spec of `inc()`, given by `presumes2/achieves2` on line 3, lets us reason about the composition `inc();inc()` in `client()` more positively, to obtain `presumes2/achieves2` as stated for `client()` on line 10. The reasoning leading to these specs uses the rules of Substitution and Constancy from Figure 3. Note that to apply these rules, a procedure spec or summary should carry information about free variables and modified – here, that x is both free and modified, and that m is not free in the procedure body (m functions as a "logical variable" or a "ghost variable".) An instance of Substitution I, substituting $m+1$ for m, is

$$\frac{[i == m \;\&\&\; m >= 0]\texttt{inc()}[ok\colon i == m+1 \;\&\&\; m >= 0]}{[i == m+1 \;\&\&\; m+1 >= 0]\texttt{inc()}[ok\colon i == m+1+1 \;\&\&\; m+1 >= 0]}$$

Next, we apply Constancy using $m >= 0$ as the invariant, to give us

$$[i == m+1 \;\&\&\; m+1 >= 0 \;\&\&\; m >= 0]\texttt{inc()}[ok\colon i == m+1+1 \;\&\&\; m+1 >= 0 \;\&\&\; m >= 0]$$

and this is equivalent to

$$[i == m+1 \;\&\&\; m >= 0]\texttt{inc()}[ok\colon i == m+2 \;\&\&\; m >= 0]$$

Now we have a triple for `inc()` whose pre exactly matches the given `achieves2` for `inc()` on line 3 in Figure 9, and we can use the sequencing rule to infer $i == m+2 \;\&\&\; m >= 0$ as the result assertion for `inc();inc()` and the `achieves2` for `client()` at line 10. When reasoning about `test()` using this spec we will not wrongly trip the assertion at line 20 because the Boolean condition `x>=2` does not imply the post-assertion `false` for the call to `client()` at line 19.

There is a general pattern to this reasoning. Let's say that a *symbolic state* is of the form

$$x_1 == m_1 \;\&\&\; \cdots \;\&\&\; x_k == m_k \wedge F$$

where the m_i are "logical variables", variables that do not occur in any program, and F is expressed in terms of logical variables only. We can make the pre-assertion of one procedure imply the

post-assertion of another by using the Substitution I together with conjoining the F parts using Constancy. Patterns of this form could be taken advantage of in automated tools.

Finally, the reader might have recognized a lack of generality of the specs given in examples previously, e.g., for the loops in Figure 5. There, we used `true` for the (default) presumes assertions, and this prevents reasoning about programs that chain several calls together. If we try to reason about `client1(); client1()` then because the pre of the second call does not imply the post of the first we have a problem. The reader might enjoy adjusting the specs to allow such reasoning .

7 CONTEXT

The ideas leading to this paper have been stewing for some time. Since the Infer tool was deployed at Facebook in 2014 it has seen over 100,000 reported issues fixed by developers before reaching production [Distefano et al. 2019]. This impact is based on two factors. First, Infer uses a composiitonal algorithm that is crucial to its deployment on a rapidly changing codebase with 10s of millions of lines. Second, it is used to find regressions on code modifications as part of the code review process. The theory Infer was based on originally [Calcagno et al. 2011] gave rise to the compositional algorithm, but was formulated as a correctness theory that does not match its use to find bugs rather than to prove their absence.

This mismatch led to a focus on under-approximation in the design of the RacerD program analyzer in work with Sam Blackshear, Nikos Gorogiannis and Ilya Sergey [Blackshear et al. 2018], an idealized version of which enjoyed a "no false positives theorem" [Gorogiannis et al. 2019]. By this point the author was becoming convinced that, perhaps contrary to prevailing opinion, design decisions favouring reduction of false positives over false negatives could be taken in a principled way, not based only on heuristics such as alarm filtering.

These considerations led to the decision to base a new analyzer – Pulse, developed principally by Jules Villard – on the idea: "what if we go back to the original design of Infer, but take bug-catching instead of proving absence as the aim?". Discussions between Jules and the author led to design decisions that were incompatible with the over-approximate theory of the original Infer:

- bound loops to escape the quagmire of over-approximating loops (i.e., depending on "magical" abstract domains that do not yet and may never exist, e.g., for memory properties);
- be disjunctive to enable precision, but prune disjuncts to maintain scalability.

Yes, it would be wonderful to have the magical abstract domains capable of automatically proving absence of all bugs in large codebases (e.g., linux, the 100s of millions of LOC at Facebook [Distefano et al. 2019], the billions of LOC at Google [Potvin and Levenberg 2016]), while also keeping up with the rapid pace of change of such codebases. No such domains have been demonstrated as of yet, so bug catching remains important.

Next, in a discussion in Lisbon at POPL'19, Derek Dreyer and Ralf Jung suggested that the aims of Pulse might be understood in terms of a logic tor proving presence of faults. At that time Derek coined the term "incorrectness logic". They proposed a model of triples with finding faults in mind, but it turned out not to fit with Pulse (e.g., it did not allow dropping disjuncts). Then, based on my extended ruminations on under-approximation, I discovered the under-approximate triples, and was surprised to find a smooth proof theory for them. It was clear that the ideas were much more general than the specific analysis problems that prompted them and I decided to try to develop the simplest version of the theory I could think. I toyed with the name "under-approximation logic", but Derek's term "incorrectness logic" has a nice ring so I used it (with his permission).

After completing the work in the paper, but fortunately before publication, I learned of the paper of de Vries and Koutavas [2011] on reverse Hoare logic (thanks to Emanuele D'Osualdo for pointing this out). They include a proof system and completeness theorem for normal termination for what

I call under-approximate triples, and I was amazed to see that they had almost the same rules for triples with *ok* conclusions. This coincidence suggests that the rules are natural or perhaps even inevitable. They do not include error specs or make the connection to proving the presence of bugs, the main topic of the present paper. But, I acknowledge the priority of de Vries and Koutavas in their discovery of the under-approximate triple.

I also acknowledge that many if not all the rules for triples with *ok* post-assertions can be obtained via compilation into dynamic logic; see Section 5.3.

There has been important related research on test-case generation by symbolic execution, as in the pioneering work of King [1976]. This whole area (see Cadar and Sen [2013] for a survey) provided inspiration for the current article. We were particularly motivated by compositional methods for symbolic execution [Godefroid 2007], which shows cases where under-approximate summaries can be computed in a way that helps the reasoning to scale to larger programs.

The concept of *necessary preconditon* [Cousot et al. 2013] is related. A necessary precondition for a program is a predicate which, whenever falsified, leads to divergence or an error, but never to successful termination. If p is a necessary pre-condition, then $\neg p$ need not be a presumption which leads to an error in incorrectness logic, because the possibility of divergence from a necessary pre-condition leads to false positives. Additionally, when $[\mathit{presumption}]C[\mathit{er}\colon \mathit{result}]$ holds it can be that C delivers successful states as well as erroneous ones when starting from the $\mathit{presumption}$. Finally, there are programs for which no non-trivial necessary pre-condition exists (e.g., `skip + error()`), but where perfectly fine presumptions exist for incorrectness logic.

Another related work is on the Thresher tool [Blackshear et al. 2013], which includes a proof system for refuting spurious counterexamples. They have a reversed rule of consequence but their post-assertions do not under-approximate, allowing for unreachable states. Where our triple is defined by under-approximating the strongest postcondition, theirs can be thought of as over-approximating the weakest possible precondition $wpp(C)q$, which describes the states that can possibly lead to q (i.e., $\langle C\rangle q$) in dynamic logic) . Thresher makes use of the capability of enlarging the pre-assertion, by dropping conjuncts in a bid to over-approximate a loop going backwards, and it may be that such a facility could prove useful for incorrectness logic as well.

Another backwards transformer, the weakest liberal preconditon $wlp(C)q$ ('liberal' allowing for divergence), can be used to characterize Hoare's triple: $p \subseteq wlp(C)q$ and $post(C)p \subseteq q$ give equivalent characterizations of $\{p\}C\{q\}$. Curiously, flipping the subset relation gives two inequivalent notions, $p \supseteq wlp(C)q$ and $post(C)p \supseteq q$. Replacing wlp with the weakest *possible* preconditon $wpp(C)q$ leads to a relationship $p \supseteq wpp(C)q$ which is as in Thresher and in necessary preconditions, the difference being that in Thresher q describes error states where Cousot et al. [2013] use q to describe success states. I am grateful to Benno Stein for discussions on Thresher, necessary preconditions, and wpp.

8 CONCLUSION AND OPEN PROBLEMS

Techniques for reasoning about program correctness have been extensively developed. Turing [1949], used logical assertions to reason about a particular program. and in the 1960s Floyd [1967] and Hoare [1969] created systematic methods for reasoning about classes of programs. In all these cases the assertions were arranged to over-approximate the reachable program states. Further developments in verification, including temporal logic [Pnueli 1981] and separation logic [O'Hearn 2019; Reynolds 2002], have expanded the techniques available for proving absence of errors.

In this paper we have suggested that reasoning about program incorrectness (or, the presence of bugs) can be placed on a logical footing, related to but different from the well developed foundations for showing correctness (or, the absence of bugs). Each form of reasoning is as fundamental as the

other, they just have different principles, as illustrated by the following fundamental duality which has been highlighted in the paper.

> For correctness reasoning, you *get to forget information* as you go along a path, but you *must remember* all the paths.
> For incorrectness reasoning, you *must remember information* as you go along a path, but you *get to forget* some of the paths.

In the paper we: (1) described how the under-approximate triple is relevant to proving the presence of bugs, and why assertions covering successful termination are needed even if our concern is with errors; (2) designed a specific logic, which we called incorrectness logic, along with a semantics and proof theory; (3) explored reasoning idioms, including making connections to concerns in automatic program analysis. We have tried to argue and show via the examples that the principles exposed in (1) and (2) are germane to the problem of proving the presence of bugs.

There are many problems for further work, including the following.

Other Models. There are models worth exploring other than where states denote functions from variable to values and where commands denote relations. Particularly important are models taking into account executions and not only initial and final states. The model in this paper describes *existence* of executions leading to errors, but not the traces themselves.

An interesting proposal in this direction has been made by Hoare (see, e.g., [Hoare et al. 2011]). He describes an approach to over-approximating triples, where $\{p\}c\{q\}$ is interpreted as $q \sqsupseteq p; c$, and it makes sense as well to interpret the under-approximate triple $[p]c[q]$ as $q \sqsubseteq p; c$. In Section 5.3 we considered such an interpretation when p, c and q denote relations, but Hoare envisages a wider range of models, including ones where they denote sets of traces. The Hoare triple can be read as saying that q over-approximates the composition of program c with an assertion describing traces up to the point before the program starts. Likewise, it would be appealing to consider a traces model of incorrectness logic and show that it forms a useful basis for (especially, automatic) reasoning. The conceptual appeal of a traces model is possibly even stronger for under-approximating than over-approximating logic: the traces would show actual executions, leading to actual bugs.

In order to reason about pointer manipulation, one might consider an under-approximate version of separation logic. This extension is not at all trivial, as the standard heap model of separation logic does not mesh well with under-approximation (separation logic's frame rule can become unsound), but in work with colleagues at MPI-SWS and Facebook we have shown that local reasoning based on the frame rule can be preserved in an alternate model; investigations continue.

Concurrency. Correctness logics for concurrency attempt to go beyond the method of enumerating interleavings to obtain more efficient means of program proof [Brookes and O'Hearn 2016; Hayes and Jones 2017]. Techniques that they use (e.g., resource invariants) were designed with over-approximation in mind, and do not immediately carry over to under-approximation. RacerD [Blackshear et al. 2018; Gorogiannis et al. 2019] gives one pragmatic example of what a concurrency logic for under-approximation might seek to cover or explain.

Compositional and Incremental Static Bug Catching. The theory of summary-based and compositional program analysis has assumed over-approximation [Calcagno et al. 2011; Cousot and Cousot 2001; Reps et al. 1995]. But, often, tools whose architecture resembles that prescribed by theory are deployed as bug catchers [Blackshear et al. 2018; Bush et al. 2000; Distefano et al. 2019; McPeak et al. 2013]. Some tools borrow ideas from interprocedural static analysis, but fall back on concrete execution for their soundness [Godefroid 2007]. It would be worthwhile to adapt the analysis that led to Facebook Infer [Calcagno et al. 2011], and the recent bounded version [Santos et al. 2019], to refer to under-approximation. Pulse is one attempt in this direction.

In addition to providing a foundation for further development of static and mixed static/dynamic bug catchers, it is hoped that theory can help expand the effectiveness of such tools.

Termination Proving. The under-approximate triple $[p]c[q]$ does not guarantee termination on all inputs, but nevertheless guarantees existence of some terminating paths. A variety of techniques have been developed for automatically inferring loop variants and other forms of termination argument [Cook et al. 2011]. We speculate that such techniques might be brought to bear on automatic reasoning about incorrectness, both to accelerate reasoning techniques within one procedure and to infer more general summaries for under-approximate inter-procedural analysis.

Abstract Interpretation. Abstract interpretation is a general theory of semantics which can in principle be used to describe under-approximation as well as over [Cousot and Cousot 1977]. There have been papers on under-approximation (e.g., [Ranzato 2013; Rival 2005; Schmidt 2007]), but the vast majority of work has concentrated on over-approximation. We hope that incorrectness logic can be neatly characterized in abstract interpretation terms, perhaps adapting the account for over-approximate logics of Cousot [2002]. More generally, we expect that abstract interpretation can eventually play a guiding and explanatory role for a wide range of static and dynamic under-approximate tools for bug catching, similar to what it already does for over-approximate analyses.

Testing. We described the relationship with testing at a high level in the discussion of the Principle of Denial in Section 2. We wonder whether logic might be used to make testing faster, perhaps in a similar way to how symbolic model checking achieved striking gains over explicit state [Burch et al. 1992]. Another avenue to explore is to attack the problem of flaky tests using logic. Generally, it seems that there is much more that can be done to exploit logic in program testing.

In summary, there is a rich variety of problems for both experimental and theoretical work to bring the foundations of reasoning about program incorrectness onto a par with the extensively developed foundations for correctness.

ACKNOWLEDGMENTS

I'm sure that I would never have developed incorrectness logic were it not for the experience of moving from academia to Facebook engineering [Constine 2013]: trying to deploy reasoning to production forced me to grapple with problems I otherwise wouldn't have encountered. I'm indebted both to my teammates on the Infer team and to the engineers in the product teams we serve for teaching me so much about applying reasoning tools in real world. Special thanks to Mark Harman and Tony Hoare for discussions on the specific content and the broader picture for this work. Finally, thanks to all the colleagues mentioned in Section 8 for discussions that influenced my thinking.

REFERENCES

K. R. Apt. 1981. Ten Years of Hoare's Logic: A Survey - Part 1. *ACM Trans. Program. Lang. Syst.* 3, 4 (1981), 431–483.
K. R. Apt and G. D. Plotkin. 1986. Countable nondeterminism and random assignment. *J. ACM* 33, 4 (1986), 724–767.
R.-J. Back and J. von Wright. 1998. *Refinement Calculus - A Systematic Introduction.* Springer.
S. Blackshear, B.-Y. Evan Chang, and M. Sridharan. 2013. Thresher: precise refutations for heap reachability. In *PLDI.*
S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey. 2018. RacerD: Compositional static race detection. *PACMPL* 2, OOPSLA (2018), 144:1–144:28.
S. Brookes and P. W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65.
H. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. 1992. Symbolic Model Checking: 10^20 States and Beyond. *Inf. Comput.* 98, 2 (1992), 142–170.

W. R. Bush, J. D. Pincus, and D. J. Sielaff. 2000. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.* 30, 7 (2000), 775–802.
C. Cadar and K. Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26. Preliminary version in POPL'09.
C. Calcagno, P. W. O'Hearn, and H.Yang. 2007. Local Action and Abstract Separation Logic. In *LICS*. 366–378.
K. Claessen and J. Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*.
E. Clarke, D. Kroening, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS*. 168–176.
M. Clint and C. A. R. Hoare. 1972. Program Proving: Jumps and Functions. *Acta Inf.* 1 (1972), 214–224.
J. Constine. 2013. Facebook acquires assets of UK mobile bug-checking software developer Monoidics. (2013). *Techcrunch*.
B. Cook, A. Podelski, and A. Rybalchenko. 2011. Proving program termination. *Commun. ACM* 54, 5 (2011), 88–98.
S. A. Cook. 1978. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. Comput.* 7, 1 (1978).
P. Cousot. 2002. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.* 277, 1-2 (2002), 47–103.
P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. 238–252.
P. Cousot and R. Cousot. 2001. Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation. In *Proceedings of SSGRR*.
P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *VMCAI*.
E. de Vries and V. Koutavas. 2011. Reverse Hoare Logic. In *SEFM*. 155–171.
E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.
D. Distefano, M Fahndrich, F Logozzo, and P.W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (2019).
R. W. Floyd. 1967. Assigning meanings to programs. In *Proc. of the Symposium on Applied Mathematics*. 19–32.
P. Godefroid. 2007. Compositional dynamic test generation. In *POPL*. 47–54.
P. Godefroid, M. Y. Levin, and D. A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*.
N. Gorogiannis, P. W. O'Hearn, and I. Sergey. 2019. A true positives theorem for a static race detector. *POPL* (2019).
A. Gotsman, J. Berdine, and B. Cook. 2011. Precision and the Conjunction Rule in Concurrent Separation Logic. *Electr. Notes Theor. Comput. Sci.* 276 (2011), 171–190.
D. Harel. 1979. *First-Order Dynamic Logic*. Lecture Notes in Computer Science, Vol. 68. Springer.
D. Harel. 1980. Proving the Correctness of Regular Deterministic Programs: A Unifying Survey Using Dynamic Logic. *Theor. Comput. Sci.* 12 (1980), 61–81.
D. Harel, J. Tiuryn, and D. Kozen. 2000. *Dynamic Logic*. MIT Press, Cambridge, MA, USA.
M. Harman and P. W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Source Code Analysis and Manipulation*. 1–23.
I. J. Hayes and C. B. Jones. 2017. A Guide to Rely/Guarantee Thinking. In *SETSS*. 1–38.
C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.
C. A. R. Hoare. 1978. Some Properties of Predicate Transformers. *J. ACM* 25, 3 (1978), 461–480.
T. Hoare, B. Möller, G. Struth, and I. Wehrman. 2011. Concurrent Kleene Algebra and its Foundations. *J. Log. Algebr. Program.* 80, 6 (2011), 266–296.
J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.
D. Kozen. 2000. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.* 1, 1 (2000), 60–76.
D. Kroening, M. Lewis, and G. Weissenbacher. 2015. Under-approximating loops in C programs for fast counterexample detection. *Formal Methods in System Design* 47, 1 (2015), 75–92.
L. Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.* 3, 2 (1977), 125–143.
P. Manolios and R. J. Trefler. 2001. Safety and Liveness in Branching Time. In *LICS*. 366–374.
S. McPeak, C.-H. Gros, and M. K. Ramanathan. 2013. Scalable and incremental software bug detection. In *ESEC/FSE*.
T. Nipkow. 2002. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, H. Schwichtenberg and R. Steinbrüggen (Eds.). Kluwer, 341–367.
P. W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *LICS*. 13–25.
P. W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.
Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7, 49 (November 1996).
A. Pnueli. 1981. The temporal semantics of concurrent programs. (1981). *Theoretical Computer Science*, 13(1), 45–60.
R. Potvin and J. Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59 (2016).
F. Ranzato. 2013. Complete Abstractions Everywhere. In *VMCAI*. 15–26.
T. W. Reps, S. Horwitz, and S. Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*.

J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. 55–74.
X. Rival. 2005. Understanding the Origin of Alarms in Astrée. In *SAS*. 303–319.
P. Rümmer and M. A. Shah. 2007. Proving Programs Incorrect Using a Sequent Calculus for Java Dynamic Logic. In *TAP*.
J. F. Santos, P. Maksimovic, G. Sampaio, and P. Gardner. 2019. JaVerT 2.0: Compositional symbolic execution for JavaScript. *PACMPL* 3, POPL (2019), 66:1–66:31.
D. A. Schmidt. 2007. A calculus of logical relations for over- and underapproximating static analyses. *Sci. Comput. Program.* 64, 1 (2007), 29–53.
A. Stoughton. 1988. Substitution Revisited. *Theor. Comput. Sci.* 59 (1988), 317–325.
A. M. Turing. 1949. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines, Univ. Math. Lab., Cambridge*. 67–69.
H. Yang. 2001. *Local Reasoning for Stateful Programs.* Ph.D. Dissertation. University of Illinois.