Continuous Reasoning: Scaling the impact of formalmethods

Peter W. O'Hearn
Facebook and University College London

Abstract

This paper describes work in *continuous reasoning*, where formal reasoning about a (changing) codebase is done in a fashion which mirrors the iterative, continuous model of software development that is increasingly practiced in indus- try. We suggest that advances in continuous reasoning will allow formal reasoning to scale to more programs, and more programmers. The paper describes the rationale for contin- uous reasoning, outlines some success cases from within industry, and proposes directions for work by the scientific community.

Keywords Reasoning, Continuous Integration

Introduction

An increasing trend in software engineering is the practice of continuous development, where a number of programmers work together sending, sometimes concurrent, updates to a shared codebase [35, 36]. The code is not viewed as a fixed ar- tifact implementing a finished product, but continues evolv- ing. In some organizations a codebase in the millions of lines can undergo rapid modification by thousands of program- mers. Their work is often backed by a continuous integration system (CI system) which ensures that the code continues to be buildable, and that certain tests pass, as the code evolves. CI-backed development extends and automates aspects of prior informal practice [50]. People naturally develop pro- grams in an iterative style, where coding and testing feed back on design, and so on. The iterative model of software development can be contrasted with the waterfall model – where one proceeds successively from requirements to de- sign, implementation, testing and deployment. Of course, the way that humans construct proofs has iterative aspects as well; e.g., trying to develop a proof of a mathematical the- orem can cause one to update the statement of the theorem, try to prove and use a different lemma, etc. The purpose of this paper is to suggest that significant benefits could accrue if formal reasoning about code can be done in an automatic continuous fashion which mirrors the iterative, continuous model of software development. Then, formal reasoning could more easily scale to many pro- grams and many programmers. Continuous reasoning views a codebase as a changing artifact that evolves through modifi- cations submitted by programmers. Continuous reasoning is done quickly with the code changes, and feeds back informa- tion to programmers in tune with their workflow rather than outside of it. To be effective a reasoning technique ideally should scale to a large codebase (sometimes in the millions of lines of code), but run quickly on code modifications (on the order of low tens of minutes). Then, it can participate as a bot in the code review system that often accompanies CI. This paper describes context for work on continuous rea- soning, but does not set out to be a comprehensive survey. We have seen continuous reasoning deployed industrially in ways that have significantly boosted impact; we mention several of the prominent cases in the next section, and then attempt a synthesis. This paper draws on experience work- ing with the Infer program analysis platform at Facebook, and we describe that experience and generalize from it. The purpose of the paper, however, is not to mainly recount what we have done at Facebook; it is rather to connect a number of related developments, as well as to tell you some of the things we don't know. We believe that there is much more that can be done with continuous formal reasoning, that can benefit the research community (in the form of challeng- ing and relevant problems) as well as software development practice (through wider impact of reasoning tools).

2 Continuous Reasoning

Rather than attempt a general definition of continuous rea- soning, we start with examples. As this paper draws on our experience at Facebook, it is natural to start there. Then we mention relevant work from the scientific literature, as well as experiences from other companies. Finally, we summarize key aspects of continuous reasoning along with its rationale.

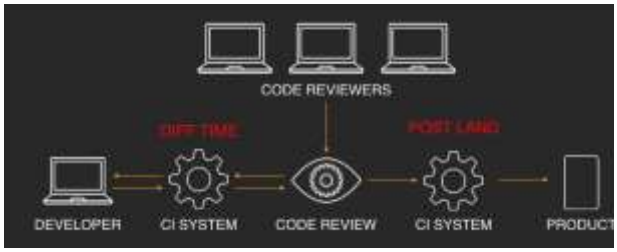


Figure 1. Continuous Development

2.1 Facebook Infer

Deployment Infer is a static analysis tool applied to Java, Objective C and C++ code at Facebook [7, 8]. It uses inter- procedural analysis, yet scales to large code.

Facebook practices *continuous software development* where a codebase is altered by thousands of programmers submit- ting 'diffs', or code modifications. A programmer prepares a diff and submits it to code review. Infer participates in this workflow as a bot, writing comments for the programmer and other human reviewers to consider. Figure 1 shows a simplified picture of this process. The developers share ac- cess to a single codebase and they land, or commit, a diff to the codebase after passing code review. Infer is run at diff time, before land, while longer-running perf and other tests are run post land.

When a diff is submitted an instance of Infer is run in Facebook's internal CI system (Sandcastle). Infer does not need to process the entire code base in order to analyze adiff, and so is fast. Typically, Infer will take 10-15mins torun on a diff, and this includes time to check out the source repository, to build the diff, and to run on base and (possibly) parent commits. Infer then writes comments to the code review system[1]. The fast reporting is essential to keep in tune with developers' workflows. In contrast, when Infer is run in whole-program mode it can take more than an hour (depending on the app being analyzed). This would be tooslow for diff-time reporting.

Foundation: Automatic Compositonality The technical feature which enables Infer's diff-time deployment is com- positionality. The idea of compositionality comes from lan- guage semantics: a semantics is compositional if the meaning of a complex phrase is defined in terms of the meanings of its parts and a means of combining them. The model theory of formal logic is typically compositional, and this idea was emphasized in computer science prominently in Scott and Strachey's denotational semantics. It can be characterized with equations of the form

$$[[C[M_1, .., M_n]]] = f([[M_1]], ..., [[M_n]])$$

and is thus similar to the concept of homomorphism, except going beyond universal algebra are involved. In contrast, operational semantics is typically non-compositional.

We can transport the idea of compositionality to program analysis and verification by substituting 'analysis' or 'verifi-cation' for 'meaning'. E.g.,

Compositional Analysis: an automatic program analysis is compositional if the analysis resultof a composite program is defined in terms of the analysis results of its parts and a means of combining them.

A crucial point is that 'the analysis result of a part' is defined (and is a meaningful concept) without having the context it appears in: compositional analyses, by their nature, are not reliant on a whole program.

A compositional reasoning example from the Abductor tool [10], the academic precursor of Facebook Infer, is given in Figure 2. Suppose we are given a pre/post pair for the procedure `foo()` which says that `foo` takes two linked listsoccupying separate memory and returns a list (`foo()` mightbe, say, list merge or append) . Abductor then discovers the precondition at line 2 and the postcondition at line 13. It does this using reasoning techniques about separation logic [58, 63] that involve *abduction* and *frame inference* [10][2]. Apart from mechanism, the important point is that a pre/post spec is inferred, without knowing any call sites of `q()`.

Infer uses compositionality to provide analysis algorithms

that fit well with diff-time CI deployment. When a proce- dure changes it can be analyzed on its own, without re- analyzing the whole program. Infer started as a specialized analysis based on separation logic targeting memory issues, but evolved into an analysis framework supporting a variety of sub-analyses, including ones for data races [5], for security (taint) properties, and for other specialized properties. These sub-analyses are implemented as instances of a framework Infer.AI for building *compositional abstract interpreters*, all

The ROFL Episode. The significance of the diff-time rea- soning of Infer is best understood by contrast with a failure. The first deployment was batch rather than continuous. In this mode Infer would be run once per night, and it would generate a list of issues. We manually looked at the issue list, and assigned them to the developers we thought best able toresolve the potential errors.

The response was stunning: we were greeted by near si- lence. We assigned 20-30 issues to developers, and almost none of them were acted on. We had worked hard to try to get the *false positive rate* down, and yet the *fix rate* – the proportion of reported issues that developers resolved – was near zero. Batch deployment can be effective in some situ- ations, but the lesson was telling. Mark Harman has even since coined a term to describe the hole we fell into [40]:

The ROFL (Report Only Failure List) Assumption: All an analysis needs to do is report only a fail- ure list, with low false positives, in order to be effective.

Let's refer to this failed deployment for the remainder of thepaper as the 'ROFL episode'.

Related observations on the challenges of batch mode deployment have been made at Coverity (see slide 5 of [18]), at Google (who exclaim *'Bug dashboards are not the answer.'* [64]), and elsewhere.

The Human Factors Lesson. Soon after the ROFL episode we switched Infer on at diff time. The response of engineers was just as stunning: the fix rate rocketed to over 70%. The same program analysis, with same false positive rate, had much greater impact when deployed at diff time.

One problem that diff-time deployment addresses is the *mental effort of context switch*. If a developer is working on one problem, and they are confronted with a report on a separate problem, then they must swap out the mental con- text of the first problem and swap in the second. ROFL is simply silent on this fundamental problem. But by partici- pating as a bot in code review, the context switch problem is largely solved by diff-time deployment: programmers cometo the review tool to discuss their code with human review- ers, with mental context already swapped in. This illustrates as well how important timeliness is: if a bot were to run for an hour or more on a diff, instead of 10-15 minutes, it would sometimes be too late to participate well in code review.

A second problem that diff-time deployment addresses is *relevance*. When an issue is discovered in the codebase, it can be non-trivial to assign it to the right person. In the extreme, the issue might have been caused by somebody who has left the company. Furthermore, even if you think you have found someone familiar with the codebase, the issue might not be relevant to any of their past or current work. But, ifwe comment on a diff that introduces an issue then there isa pretty good (but not perfect) chance that it is relevant.

Impact. Over the past four years, tens of thousands of issues flagged by Infer have been fixed by Facebook developers before reaching production.

Our earlier labelling of the 'ROFL episode' was admittedly tongue-in-cheek. The continuous deployment of Infer was always planned, and was not done in reaction to the episode(though perhaps it was accelerated). We have since had some successes with batch mode deployment; these will reported on in future. But, the vast majority of Infer's impact to this point is attributable to continuous reasoning at diff time, and the point is that this deployment *eases* a number of difficulties, but not that that it is the *only* way to do so.

2.2 Scientific Context

Numerous ideas have been advanced in the research com- munity which are relevant to continuous reasoning.

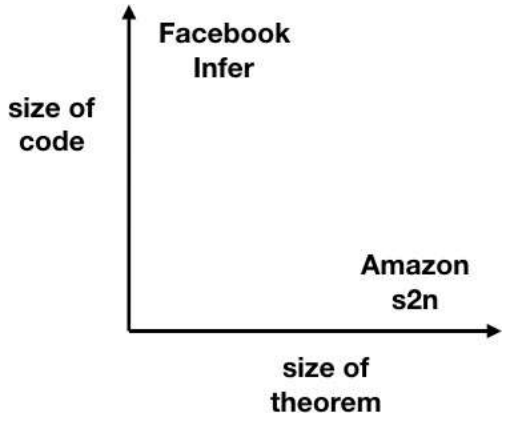
A fundamental principle of Hoare logic is its incremen- tality: one reasons about calls to a procedure using its spec, and not its code, and therefore a change to the code of a procedure would not always necessitate re-proving an entire program [42]. This principle drove the initial foundation of Infer: the pre/post specs are computed automatically, and then incrementality flows as in Hoare logic. In fact, a com- plete program is not needed to verify a procedure wrt its pre/post specs in Hoare logic style; Hoare logic is composi- tional in this sense as well as incremental.

Pioneering work of King showed how a program could beannotated with assertions at key points and then verification conditions would be generated which could be discharged by a theorem prover [47]. Many tools have been developed over the years extending the ideas of King, and relying on Hoare logic's incrementality. Representative recent tools, which have seen impressive successes in verifying challeng- ing programs, include Dafny [52] and VST-Floyd [11]. Dafny uses an automatic prover to discharge the verification condi- tions derived from a procedure body and it's pre/post spec (plus, perhaps, loop invariants): it is an example of what issometimes referred to as a *mostly automatic* verifier: 'mostly' because the human does the work of supplying pre/post specs and other annotations, and then the automatic prover takes over. VST-Floyd uses a similar overall approach, except that an interactive theorem prover (in this case, the Coq proof assistant) is used to discharge the verification condi-tions. One might say that in Dafny and VST-Floyd share a manual approach to compositionality, differing in their level of automation for reasoning after specs have been placed, while Infer is automatic both in its compositonality and in its reasoning. Typically, one can prove stronger properties as automation decreases.

In the early 2000s, Hoare proposed a grand challenge for computing research – the verifying compiler [43] – which evolved into the verified software initiative [45]. Hoare re- alized that getting feedback to the programmer at compile time could

make a difference in effectiveness of signal, and conceived of mostly automatic verifiers being used by programmers. The Infer experience above is in a similar spirit, except that signal happens at diff rather than compile time, and that the human is not required to supply annotations to drive the tool. With diff-time deployment for code review enough time has elapsed for the tool to do significant work, but not so much time that signal comes too late.

Turning to automatic program analysis, while the idea of a compositional analysis is well known (e.g. [24]), it seems fair to say that the majority of research papers in the subject focus on a whole-program setup. Typically, an analyzer targets small intricate programs or a collection of more general test programs, but in either case its evaluation is given by manually examining a list of alarms (or the information that a proof succeeded); this evokes the ROFL assumption mentioned earlier. While this kind of work, which has often fixated on precision for fixed (often small) programs, has led to many important ideas, perhaps some rebalancing is in order: more research effort might be directed towards the problems of large, changing codebases.



Finally, ideas of analyzing the difference between two pieces of code, and of reporting results as soon as possible, have also been the subject of scientific papers; e.g., [32, 49].

2.3 Industry Context

Deployments similar to Infer's implementation of continuous reasoning have occurred in several other contexts. We begin with one that is nearly the opposite on several dimensions.

Amazon s2n is an implementation of the Transport Level Security protocol which is in widespread use in Amazon's data centers. Amazon have, in a collaboration with Galois, proven strong security properties of key components of s2n [19]. For example, they establish that the s2n implementa- tion of HMAC implements a pseudorandom function. Some salient features of their effort are as follows.

The specifications themselves are the subject of signif- icant design and proof work, connecting several levels of specs by reasoning in the Coq proof assistant and the Cryptol tool from Galois, some done in [19] and some building on separate work [4].

The code is small: just over 550 lines of C.

The connection between the lowest level of Cryptol specification and s2n's C code is verified automatically using the SAW tool from Galois [33]. The pro of is re- played in CI on every submitted code change.

Pre/post specs are supplied manually, in common with the mostly automatic verifiers, but loop invariants are not needed because the loops are bounded. The incre- mentality of Hoare logic means that the effects of code changes on a proof effort can be localized.

The specification is expected to change much less fre- quently than the code; e.g., the top-level spec of HMAC has changed only once since 2002.

Figure 3. Facebook versus Amazon

Over a period of just over a year from November 2016 the proof was re-played in CI 956 times, and had to be (manually) altered only 3 times.

Theirs is one of many proofs of small security-critical pro- grams that have been carried out; the distinguishing feature is how the proof is re-played in CI. The motivation they give is that *'verification must continue to work with low effort as developers change the code' [19]*. That the proof had to only be altered 3 times is perhaps the most important take-home from the work. A fundamental challenge for full-blown verifi- cation is the question of ongoing drag (see [40] for definition and broader discussion), and in the worst case one could imagine needing a verification expert sitting beside each pro-grammer to drive the re-proof: i.e., human CI. Although the data from the s2n work is too limited to draw broad conclu- sions for general code, it seems clear that in some scenarios continuous verification will require considerably less than one verification expert per programmer.

The ways in which the Facebook and Amazon efforts differ are striking (Figure 3). Infer emphasizes covering large code in the millions of lines, it targets simple to state properties (e.g, run-time errors; 'small theorems about big code'), and it uses fully automatic compositional reasoning. The Amazon s2n work is for small code in the hundreds of lines, it targets a highly nontrivial and lengthy specification ('big theorems about small code'), and it uses manual specification of inter-faces to achieve incrementality. Both efforts, as it happens, involve unchanging (or rarely changing) specifications.

Microsoft Numerous reasoning tools have been deployed over the years at Microsoft. Two of the more prominent are *Prefix* and *Prefast* [51]. Prefix is a global inter-procedural analysis that scales to large code bases, but was too slow to deploy at the analogue of diff time or code review time: it was

used in overnight runs. Prefast was a reaction to challenges of Prefix. Prefast analysis is limited to a single function. Therefore, the analysis is fast and can be run at the same places as build; this could be the analogue of diff time in CI, or on local machines. Prefast needs inputs and outputs to be annotated manually by the programmer (e.g., with range or nullness information) to deliver effective signal .

Google A recent paper explains how Google has experi- mented with numerous static analyses [64]. Earlier, in the context of ROFL, we mentioned how they concluded that batch deployment producing bug lists is not effective. They also say that *'Code review is a sweet spot at Google for dis- playing analysis warnings before code is committed.'* So far, inter-procedural analysis or sophisticated, logic-based analy-sis techniques have not been a priority at Google according to [64]; the analyses are intra-procedural and relatively simple. But, it is remarkable how strongly similar the conclusions reached by Facebook and Google are on ROFL (batch) and code review (continuous) deployments.

Altran Altran (formerly, Praxis) is a pioneer of industrial program verification, having been in the proof-producing business for over 20 years [14]. They developed the Spark programming language, based on Ada, which uses a mostly automatic verifier. Spark restricts programs to features for which traditional Hoare logic works well (eg.: no pointers, no concurrency), and is used in a part of industry (safety-critical computing) that one might think of as far removed from the 'high momentum' development practices of internetcompanies such as Facebook and Google.

Remarkably, a recent paper explains how verification for safety-critical systems with Spark is compatible with Agile engineering practices [15]. The specification of a product is not assumed to be rarely changing. They recount one expe- rience with software that provides information about flight trajectories and potential conflicts to air traffic controllers in the UK: *'Overnight, the integration server rebuilds an entire proof of the software, populating a persistent cache, accessible to all developers the next morning. Working on an isolated change, the developers can reproduce the proof of the entiresystem in about 15 minutes on their desktop machines, or in a matter of seconds for a change to a single module'.*

We stress that the very notion of what we are calling con- tinuous reasoning is not novel. The general idea is straight- forward, and probably no attribution is necessary. More im- portantly, there are valuable scientific principles which have been advanced that support the general idea, and (as we will argue) there is much more research to be done. And, ultimately, more powerful than novelty is corroboration: continuous reasoning is actually being practiced industrially in several companies, who are reporting similar benefits.

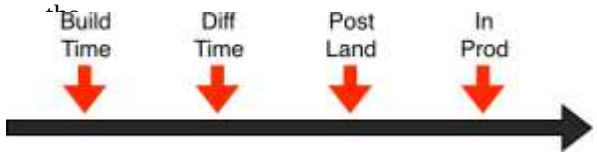


Figure 4. Bug Time Spectrum

2.4 Synthesis

In this paper by (diff time) continuous reasoning we mean:

Reasoning is run on code changes, done automatically in CI, with reporting that is timely enough to partici- pate as a bot in code review.

What 'timely' means can vary from organization to organi- zation. At Facebook we have a rule of thumb that feedback should be reported on average in 20mins for code review.

Benefits of continuous reasoning are as follows.

By being run automatically in a CI system when a programmer submits code, the programmer does not need to consciously or actively run a tool, helping itto scale to many programmers.

By focussing on fragments of code in code changes rather than whole programs, incremental techniques can help deliver timely feedback even for a codebase in the many millions of lines of code being modifiedby thousands of programmers.

By participating in code review, the tool presents feed-back to developers in tune with their workflow, avoid-ing the mental effort of context switch.

By identifying issues 'caused' by a code modification, developers are much more likely than random to re- ceive reports relevant to what they are working on.

In this paper we are focussing on diff-time continuous reasoning, but more generally there is obviously a spectrum running from build time to diff time to post land and later (e.g., Figure 4). Different engineering and scientific problems arise at each point in the spectrum, and at Facebook we are investigating all of them. But, following on from the ROFL episode from earlier the most common question that comes up is: how can we move a technique to diff time from later?

Scientific Challenges

We want to distinguish two questions that can arise when thinking about deploying reasoning techniques.

Question E. Suppose we were to take a *current snap- shop* which effectively freezes development of new techniques for reasoning about programs. For each technique (e.g., BMC, CEGAR, concolic, fuzz, numeric domains ..), where can it be effectively deployed onthe bug time spectrum?

Question S. What advances can be made in reasoning techniques, which create a *new snapshot*

of knowl- edge that might (perhaps dramatically)

Question E is primarily an engineering one. Inputs to answer- ing the question include: effectiveness of signal (including but not limited to precision); resource issues such as power and CPU time; and startup plus ongoing effort for people (both tool developers and tool users). Long-established techniques (e.g., typechecking) are as relevant to Question E as more recent, less developed, ones; sometimes even moreso. Answering Question E is difficult because it inherently in- volves uncertainty: fully comprehensive data on all these inputs across the spectrum is hard to come by. As a con- sequence, experimentation, creativity, and judgement areneeded in approaching it.

Question S, in comparison, is primarily a scientific ques- tion. It is the one we will concentrate on in this section.But, it helps to keep the engineering question in mind when formulating and studying problems related to Question S.

What should be the alarms

Given a fixed whole program and a kind of error that a global program analyzer is attempting to find or prevent, the issue of what should be an alarm is often straightforward: e.g., potential errors together with evidence (say, traces starting from a `main()` function) that they can occur. This idea, while clear enough, depends upon the

Closed World Assumption (CWA), that analyzers apply only to complete programs.

This is to be contrasted with the

Open World Assumption (OWA), that analyzerscan apply to incomplete program fragments.

Adopting the OWA as the basis for research opens up many applications that are difficult to approach under CWA.

But, for an OWA reasoning tool the question of what the alarms should be is not as obvious as under CWA. Suppose we

Infer finds a trace with 36 steps before a potential returnof `NULL` is encountered in the call to have a function like `f()` to

the right which immediately dereferences a pointer. When

deployments have gone beyond the usual analyzing this function on its own, without knowing any

callers, an analyzer would be ill-advised to report a null deref- erence at line 3; else, in analyzing any non-trivial codebase the programmer would be flooded with spurious warnings. Infer suppresses the potential null alarm by inferring a pre-condition, which says that `ptr` holds an allocated pointer.

On the other hand, analyzers for large-scale software de-

ployments will often report issues without having a path back all the way back to a `main()` function. For example, Infer reported[3] an issue in OpenSSL as follows:

alter answers to Question E in the future.

`X509_gmtime_adj()`[4]. However, Infer does not know that the specific call at line 2778 will return `NULL` in some actual run of a program. Rather,it fails to synthesize a precondition ruling out `NULL` and uses this failure, together with some reporting heuristics which involve the source of `NULL`, to decide to raise an alarm. In fact, Infer makes this report on fragments of OpenSSL that are not even placed in a running context (there is no `main()`). (Note that this discussion is not about null pointers *per se*; issues of reporting without a trace to `main()`, and also of filtering alarms to reduce false positives, arise with array bounds errors, data races, and many other runtime errors, when we do not make the monolithic system assumption.)

What are we to make of this from a scientific point of view? One possible reaction is to say that Infer, Prefix, the offering from Coverity, and other analyzers running on large- scale code are simply heuristic: they do reasoning, but fallback on ad hoc heuristics when deciding what to raise as an alarm, or even what paths to explore.

Another reaction is to say we should only surface 'purelylocal alarms', ones that we know will arise in any evaluation of a function; a kind of mini-CWA. This would simply miss too many bugs, and would therefore give up impact, though how many missed varies from category to category. For example, we have observed very many local memory and resource leak errors reported by Infer; memory or a resourceis allocated and then it is freed along a main execution path within a function, but the programmer forgets to free onan exceptional path. On the other hand, extremely few null dereference errors or information flow alarms raised by Inferhave been purely local in this sense.

A final reaction, the one that we prefer, is to think that

there is possibly something to understand here. Industrial

assumptions of thescientific field, something that is expected and natural when science and industrial engineering bump into one another.

This presents an opportunity to the scientist to re-examine assumptions and, consequently, provide a better basis for the engineering of reasoning tools in the future.

One idea that occurs is that any alarm-surfacing strategy be paired with a code transformation that closes the program with surrounding code that 'explains' the error reports. A tool would not literally complete the code, but this wouldbe part of the conceptual explanation of the alarms. Note that one is not looking for, say, a most general in some sense surrounding program, but one that helps understand, judge and guide analyzer choices.

Whether this speculation or any other ideas bear fruit in explaining alarms remains to be seen. In any case, we want to stress to the reader that the task of

deciding what the alarms should be is a very important one faced when designing an analyzer to be used by people rather than for the purpose of experiments, especially for analysis of large-scale codebases. In our experience the science of reasoning about programs does not yet provide much guidance in this important task.

What and when to report

As mentioned in Section 2.1, the ROFL assumption would have it that all a reasoning tool needs to do is report a list of alarms. But we also saw there than *when* the alarm is reported has a large effect on whether the signal is effective.

Table 1 lists some of the reporting possibilities.

- *Lean* reporting of only new errors only on changed files is Infer's default at diff time. It is a low friction deployment: it avoids reporting pre-existing issues to an author of a diff, which typically will not be relevant to the diff. It supports the first axiom of industrial staticanalysis: *Don't spam the developer*.
- *Bulky* reporting can, when run on a large legacy code-base, result in numerous pre-existing issues being re-ported. Sometimes these can be overwhelming, andirrelevant to the diff author, so care is needed in this reporting mode. (With Infer, we are
- experimentingwith it for certain bug types and certain projects.) *Cautious* fits well with periodic global analyzer runs on an entire codebase, as opposed to at diff time. It has been used by Coverity [18], and versions of it are usedfor both static and dynamic analysis at Facebook.
- *Clean* is used for deployments that seek to keep a code- base entirely free of certain issues. The Amazon s2n effort uses this deployment, Infer has used it with the source code of the Buck build system, and it is com-monly used with type systems.

Of the dimensions in Table 1, focussing on changed files is somewhat arbitrary; it is what we do with Facebook Infer currently, but one could equally choose the granularity of changed lines, procedures, or even build targets. The new- versus-old dimension has some subtleties, in that identifying a 'new' issue can be challenging in the face of refactoring or code moves. Also, there is more than one way to identify 'old' issues (e.g., by running on base as well as parent commits, or by keeping a database of known issues), and they are not equivalent. Identifying new issues could itself be the subject of theoretical definitions and experimental analysis.

In this discussion we are taking 'report' to mean an active indication to a programmer of an issue. This could, say, be a comment on a diff (in Lean reporting) or an email to an intended not to disturb programmers.

We are not proposing any deep scientific questions on reporting modes. Rather, having these distinctions in mind is useful for framing other questions.

3.1 Automatic Program Analysis

In this section we discuss challenges for automatic programanalysis of large codebases at diff time. Our focus is mainly on compositional and incremental techniques.

Compositionality and Incrementality, Intuitive Basis Compositional program analysis helps scale a reasoning tech- nique to large codebases and it naturally supports diff-time continuous reasoning. The intuition for scalability is that each procedure (or unit of modularity, such as a class) only needs (ignoring recursion) to be visited once by a compo- sitional analysis, and furthermore that many of the proce-dures in a codebase can often be analyzed independently, thus opening up opportunities for parallelism. Therefore, if we knew a bound on the time to analyze each procedure (or if we imposed one) then the analysis time would be an additive linear combination.

Dealing exhaustively with code pointers or with recursion, which causes a procedure to be visited multiple times if a fixed-point is to be sought, can complicate this basic intu- ition. But they do not completely undermine it. Recursion cycles can be arbitrary broken pragmatically if they are too large, while still obtaining useful results, and alarm-surfacing decisions can mitigate the precision effects of unknown code pointers. In applications it is useful to start from an analysis that scales to large code because of the additivity intuition above, with appropriate mitigations in place, rather than start from one that does not scale. Then one can achieve non-zero impact early, followed by iterative improvement in reaction to developer feedback and other data.

Compositional techniques naturally lend themselves to incremental algorithms, where changing a small part of code requires only re-analyzing that code (or that code plus not too much more). The reporting model, as in Table 1, has a signif- icant effect, though. If alarms are reported only on changed files, then only those files plus procedures transitively called from them need to be re-analyzed; and if summaries for the transitive dependents are in cache, their analysis can (mod-ulo mutual recursion) be avoided too. However, if all alarms are to be reported, then it might be necessary to analyze many files other than those that have changed.

Technical Challenge Although compositional static anal- yses exist [24], the area is extremely under-developed. Valu- able reasoning techniques such as interpolation, abstraction refinement, and various numerical abstract domains have been mostly studied in the context of whole-

program analy- sis. This leads to to the following challenge.

Automatic Compositionality Challenge. For each program reasoning technique – e.g., BMC [21], CEGAR [20], interpolation [55], symbolic and concolic testing [6], fuzz [56], SBSE testing [54], numeric abstract domains [26] – formulate an automatic compositional variant.

Demonstrate experimentally and explain theoretically its scalability to large codebases (millions of LOC).

Demonstrate experimentally and explain theoretically per-diff scalability, where running time is proportional to the size of a code change and not an entire codebase; or document assump- tions sufficient for such incremental scalability.

Demonstrate effective signal.

Let us discuss the demonstrations the challenge asks for.

Concerning scaling to large code, consider (for example) that: (i) The celebrated ASTREE analyzer [26], which in- cludes sophisticated numeric domains as well as techniques to balance precision and speed, verified the absence of run- time errors in 400k lines of C code in a run of over 11 hours[5];

The SLAM tool, which represented a leading example of CEGAR in action, sometimes timed out (ran for more than 2000 seconds) on device drivers in the low 10k's LOC [3];

the CBMC tool [21], a leading bounded model checker, has been reported as running for over a day on 100k LOC programs [17]. In fact, according to Daniel Kroening (per- sonal communication): *'CBMC isn't designed to scale (simply since SAT isn't designed to scale).'*

On the other hand, if we think of the jump from Space Invader [68] to Abductor [10] – from 10k LOC (and timing out due to memory exhaustion on slightly larger programs) to 2.4M LOC in 2 hours – then we might hope for these andother techniques to scale to a much higher degree. As an indication of potential, were there a tool for precise numer- ical program analysis, in the spirit of ASTRÉE but able to deliver fast results on codemods to a 10MLOC codebase, then it could prove useful. Admittedly, this is not necessarily an easy goal.

Concerning incrementality, several of the techniques men-tioned in the challenge have been married with summary- based inter-procedural program analysis [61]. This gives a potential route to explore the diff scalability requirement. In fact, there have been many non-compositional, summary- based program analyses, which start from a `main()` functionand use summaries only to enhance scalability, not to compute specs in a context-independent way: compositionality is not necessary to incrementality. Compositionality allows to analyze incomplete code (such as checking a class for poten-tial thread safety errors, future-proofing it before it is placedinto a concurrent context), but this is an additional capabil- ity over the incrementality that is needed to support diff- time analysis. Deploying whole-program, non-compositonal, summary-based analyses at diff time is an obvious idea to try, but not one that we are aware has been explored.

There are also some relevant works on compositional (and not just incremental) analysis. For instance, [65] describes a way of inferring preconditions which fits together with CEGAR and predicate abstraction, while [25] does so for a numerical abstract domain. In short, there are good starting points for investigating the global and per-diff scalabilityrequirements in the challenge.

Concerning effective signal, this is the least clear part of the challenge. The mentioned techniques have been mostly developed in a whole-program setup, and so answers to the question of what alarms to raise have been (unconsciously, perhaps) driven by the Closed World Assumption. We leavethis part of the challenge underspecified, hoping that re-searchers will use such concepts as false positive rate, fix rate, alarm volume, or even fresh ideas to address it. See [40] for further discussion on effective signal.

Care in comparison: OWA versus CWA. It is tempting when comparing techniques to, unintentionally, focus on dimensions that are the home turf of one but not the other. We therefore offer remarks on the relation between closed and open world techniques, which reflect conversations we have had with scientific researchers and industrial engineers in analysis over the past several years.

First, it is important to remember that developing com- positional versions of any of the techniques (interpolation, CEGAR, etc) mentioned in the challenge could necessitate new answers to the question 'what are the alarms'. The resul-tant alarms would then need to be evaluated for the fidelity of the signal that they provide. So, if we were to try to com- pare compositional (open world) and whole-program (closedworld) versions of a technique, even ones that used the sameabstract domain, it might not be an apples-to-apples compar- ison if the decisions on what should be alarms are different.This remark about not being apples-to-apples applies to proof as well as to alarm surfacing. An example is again in the comparison of Space Invader [68] and Abductor [10], tools that use the same abstract domain but differ in theirapproach to summaries. Space Invader was able to provepointer safety of a 10k LOC Windows device driver (the IEEE 1394 Firewire driver). The proof used a 'verification harness', a fake `main()` function that called the dispatch routines of the driver repeatedly after non-deterministically allocating representative data structures accessed by the driver. Abduc-tor, applied to the same driver without the harness, was able to find specifications for all 121 functions in its code.

[5]We will quote any running times in this paper with the understanding that they refer to the 'hardware of the time'; in none of the cases is the detail ofthat hardware very pertinent to the point being made

But, one of the preconditions discovered was overly

Abductor could not complete the verification when the har- ness was added to the code. So, Space Invader was superior to Abductor when it came to complete proof of absence of pointer errors in small (10k LOC) programs, and it resulted in more that 40 memory safety bugs being fixed in a Microsoft driver. On the other hand, Abductor could scale to millions of LOC and it supported per-diff incrementality. These latter features were eventually used in Infer at Facebook, and led to much greater impact.

An important lesson is that it is incorrect to view compo- sitional algorithms as merely optimized versions of slower global algorithms in a CWA scenario: the compositional algo- rithm may embrace the Open World Assumption, and might therefore answer different questions. A second lesson is that, if the compositional algorithm is less precise (under CWA) than the global one, we should not therefore conclude that the compositional algorithm is not up to standard. If you have two algorithms, one with a great deal more impact than the other, you shouldn't blame the more impactful algorithm for not matching the less impactful one; if either algorithm needs to have its *raison d'être* explained, it's the less impact- ful one. (This discussion calls to mind Vardi's principle: *'the brainiac often loses to the speed demon'*.)

Dynamic and Bounding For over-approximating static techniques it would be useful to look at bounding variants (say, limiting the number of times recursion cycles are tra- versed) while measuring quantities such as alarms/minute, to

for the function `h()` to the right. How can we reason about this using static analysis, in particular to know whether

quantify the benefit of reaching a fixed-point. This is not to suggest that the fixed-point should be avoided; rather, some alarms might be delivered early, at diff time, while others (or the announcement 'proof') could wait for a post-land run.

The techniques mentioned in the Compositionality Chal- lenge include ones from dynamic as well as static program analysis. For bounded and under-approximate methods, the potential to be unlocked by diff-time reporting is possibly even greater, because techniques such as fuzz, concolic, and bounded model checking are often considered in the time- consumig category, which can take hours of running time to achieve much code coverage on large codebases. It is possi-ble to run an analysis for a shorter time by exploring fewer paths, so genuinely answering this challenge should involve metrics such as code coverage or alarms/min which indicate that speedups are not obtained only by delivering less.

One promising direction for work here involves mixing static and dynamic analysis. Indeed, there have been works that transport the concept of procedure summary from static analysis to testing and specific, and

incremental-on-code-mode scalability question, but there is a very real prospect of increasing their impact if the techniques can be used for effective incremental diff-time reporting. For example, if a tool were to run for hours or overnight to obtain procedure summaries for a large codebase, but it was possible to get quick results on diffs by re-using cached summaries for unchanged code parts, then the impact of all these techniques could be raised. Further work in this direction could turnout to be very worthwhile.

There have even been automatic dynamic analyzes that attempt to analyze code fragments. One of the leading works in the area runs on a procedure in isolation from its call sites and uses a technique called 'lazy initialization' to build a description of a needed data structure [60]. In this respect it is similar to the precondition inference of Infer; see [9]. It also runs into the problem of Section 3.1, on what the alarms should be. They report good results on memory leak errors on one project (BIND), but have many false positives (over 70% and even 90%) for other bug types and other code. Generally speaking, the technical and conceptual challenges of dynamic analysis for incomplete code fragments are great, but progress in the area could be useful.

On Higher Order and Unknown Code Consider the code

bounded model checking [17, 38, 39][6]: these have been used to approach the global rather than the

[6]Note that these works are using the term 'compositional' for what we would call a whole-program, non-compositional, summary-based analysis. They use summaries to avoid repeating computations, but require a (whole)

to report a null dereference for either of the dereferences? A typical approach is to say that first one runs a global alias analysis which collects all of the potential targets of the call

`*goo()`, before reasoning about them. But this relies on CWA. Code pointers are a fundamental problem under OWA.

A compromise is used by Infer. When Infer sees a deref- erence that it does not know to be safe, it *abduces* an as- sumption of non-nullness and attempts to re-express the assumption using data from the function's pre-state. At line 3 we abduce that `goo` should not be null to avoid a null dereference, and this becomes part of the function's inferred precondition. At line 4 we abduce that `ptr` should not be null, and we can recognize that this in effect means that the *call* to `*goo` at line 3 should not return null. In effect, we ab- duce an angelic assumption about the behaviour of `*goo()`, suppressing a potential null dereference alarm at line 4.

This issue is similar for any kind of unknown code, and not just parameters that are code pointers.

Angelic reasoning [27], as about `*goo()` above, helps avoid spamming developers with reports

about unknown
code. While useful, it is not always the right solution. Where Infer makes angelic assumptions and we make it only to avoid confusion: the results in these

as when reasoning about thread safety of Java classes. Blan- ket angelic or demonic assumptions about unknown code can be useful for making engineering compromises, but neither provides a general solution to the problem of compositional reasoning about code pointers under OWA.
An interesting idea is, instead of inferring an assertion expressing a blanket assumption, to try to abduce more ex- pressive specifications (such as pre/post specs) about how unknown code should behave [1, 37]. In addition to hav- ing more expressive potential, this direction might lead to greater parallelism and incrementality in an analysis.
Note that the issues here are are problematic even for tools where the programmer is asked to manually insert specifications, as pointed out in this remark by Chlipapa:
'Verifications based on automated theorem prov- ing have omitted reasoning about first-class code pointers, which is critical for tasks like certifying implementations of threads and processes.' [16]
The response of researchers in interactive verification has been to use higher order logic, in a verification analogue of the use of higher types in the account of data abstraction [62]. This suggests another research problem: develop abstract domains that utilize higher order logic, and demonstrate how they can be used to reason fully automatically about first class code pointers under OWA.

3.2 Mostly Automatic and Interactive Verifiers

Impressive advances have been made in mechanized verifi- cation in recent years. We have seen example verifications of OS microkernels [48, 67], a file system [13], crypto algo- rithms [4] and distributed systems [41]. Some of this work develops the program and the proof side by side, while otherwork proves existing code. It is now feasible, in (say) several person years or even months, to prove functional correctness properties of programs in the range of 10k LOC.

This is a remarkable situation to have gotten to. But ... now what? Will it be possible for full verification to scale to many more programs and programmers? Or will and should verification be an activity reserved for a tiny proportion of the great many programs in the world? While we don't have answers to these questions, it is well to additionally ask: what are the main impediments to scale?

The expertise bottleneck A fundamental bottleneck is the level of expertise needed to drive a tool. Some leading propo- nents of the mostly automatic

when reasoning about null pointers, it makes demonic assumptions in other cases such

papers are veryvaluable independently of terminological matters.
approach have put it starkly:

> *'The quality of feedback produced by most veri-fication tools makes it virtually impossible to use them without an extensive background in formalmethods [53].'*

And if this is an issue for mostly automatic verification, it islikely moreso for interactive.

One reaction to this problem is to restrict attention to spec- ifications of (much) less than full correctness, for example by supporting simple annotations that are more like types. This approach has been extraordinarily successful and will likely continue to be. Having accepted this point, let's move on to consider reactions to the problem that maintain the focus on full functional verification.

Another reaction is that we should grow the collection of people who are experts in formal methods. This reaction has its merits, and education should certainly be a priority. It would be wishful thinking to hope that all programmers willbecome formal methods experts any time soon, but growingthe community can still be positive.

The third reaction is that we should strive to make the tools more usable. Partly, this is a tool engineering problem (quality of feedback), but there are scientific aspects as well. Advances in program logic (e.g., [46, 59, 63]) have led to simpler ways to specify programs, and advances in theorem proving (e.g., [28, 29]) can lead to greater automation.

The specs bottleneck Much research in verification pro- ceeds from the assumption: the formal specifications are given. But, specifications are non-trivial to come by. Cre- ating a formal specification is time consuming, and people need to be convinced that the value obtained is worth the effort put in. Even if we had perfect, push button verifiers, mostly automatic tools would still be non-trivial to deploy to large programming populations.

It is hard to know whether the expertise or specs bottle-neck is the greater problem for scaling (likely, they are not independent). But in discussions we have often observed questions about the need for specs being expressed well prior to discussion of the detailed tools. This can be taken as providing motivation for technical research.

The 'where do we get specs' question suggests further research in guessing specs from dynamic information [2, 34,66], as aiding the human's specification-making activity. Re- latedly, further integration of verifiers with static program analyses, which can sometimes infer the intermediate specifi- cations (including pre/post specs) needed to make a proof gothrough, could limit the number of annotations that the pro- gramer needs to place, or even suggest such specifications to the programmers [12]. Finally, there is a frustrating du- plication of effort in

writing tests and writing specifications; unification of these activities could be valuable.

Engineering principles for proof Proofs as engineering artifacts raise similar problems to those which motivated abstraction and modularity in software design. For example, if proofs have sufficient modularity, then it stands to reason that a small changes to code is less likely to necessitate global changes to a proof than if proofs are monolithic.

Suppose that a verified program changes. If re-proof failure uncovers an error, with actionable information, then the verifier provides benefit. If a human expert needs to update the proof when the code continues to satisfy the top-level specification, then this is like a false positive. The less human interaction is needed to update a proof, the easier it is for verification to be used by more programmers.

It is important to note, though, that there is no need to wish that verification technology becomes so easy to use that deployment doesn't need to be supported by subject-matter experts. Often, when a technology is deployed, ongoing sup- port is required. This is true for programming languages, build systems, and automatic program analyzers. We want to minimize the number of experts needed, which is not to say eliminate them. So, even if experts need to be on hand occasionally to help with a proof, if we can drive down the ratio of formal methods specialists to programmers, then we can more easily support a greater number of programmers.

These directions suggest fundamental work aimed at sim- plicity, increased automation, abstraction and modularity of specs and proofs. One small suggestion is that research pa- pers might usefully address the question of effort to update a proof when code changes, in addition to the effort to produce one in the first place. We are not proposing revolutionary advance here, but rather continued steady improvement.

3.3 Abstract Theory for an Open World

Concepts from programming theory – including semantics (compositionality), Hoare logic (incrementality of procedure call rules) separation logic (frames and footprints), and ab- stract interpretation (approximation) – have played a role in influencing program analyzers for continuous reasoning about large codebases. But, our experience is that the engi- neering of such tools is running into problems that theory does not much help with.

How to design a notion of summary to support effec- tive analysis under the open world assumption (OWA)? How to resolve the tension between needing to infer a concise summary and code fragments where logically no best spec exists ?

What should the alarms be, and how can the choice be justified?

What general assumptions are needed for global and per-diff scalability?

These considerations suggest opportunities for science.

Develop a general program analysis framework, in the spirit of abstract interpretation [23], which embraces OWA. It should

include a general notion of summary, which does not specify their form (e.g., input/output pairs) in advance, but that accounts for creation and instantiation of summaries;

formulate its notion of summary in general terms which allow the choice of a 'unit of modularity' such as a function, a class, or a process;

include an account of the frame problem (locality) for summaries, and independence between analysis of program parts;

include a notion of alarm (not just over and under approximations), and ways to justify (understand) the alarms;

include theorems related to scalability which explain the practice of existing observably-scalable OWA analyses and suggest future ones.

Concerning (i), frameworks for global inter- procedural anal- ysis have been developed, but typically the summaries take a special form (such as based on tabulation of input/output pairs, [61]). Part of the art of designing a compositional anal- ysis is choosing the notion of summary, and often the good choice is not simply a tabulation of pairs [69]. Possibly, a theory should view summaries as abstract (in the sense of abstract algebra), their structure not specified in advance.

Summary instantiation is an area where bugs can easily arise in design of inter-procedural analysis. This parallels the experience with procedures in Hoare logic, where it was reported that bugs in proof rules occurred with regularity [22]. Also, precision questions could be studied, related to the concept of adaptation completeness in Hoare logic [57]. Gen- erally speaking, theory surrounding summaries, parameters, and summary instantiation would be helpful.

Concerning (ii), summaries are not for procedures only.

Concerning (iii), in order for reasoning to scale, it is often desirable to ensure that summaries be concise [69], and that the abstract states describe localized pieces of memory. These desiderata are related to the frame problem from artificial intelligence, and to the local reasoning idea from separation logic [58]. Related ideas have been used in a semi-formal way in many specialized program analyses (e.g. [30], and the RacerD and Quandary instantiations of Infer.AI).

A paper of Cousot and Cousot [24] from 2002 presents a framework for compositional analysis. It does not address how to obtain summaries, or

how to instantiate them, but it gives a very clean formulation of what it means for an abstract interpretation to be compositional. The paper identifies crucial issues to do with the independence of the separate analysis of the components, which is connected to the effi- ciency of the 'means to combine them' part of compositional- ity. The intervening years have seen many advances related to independence, as well as to the means of combination.

For example, separation logic provides primitives for ex- pressing independence, and bi-abduction [10] implements a means of combining which takes (partial) independence into account. For another example, the Views theory [31]provides a framework for compositional reasoning about concurrency, which one might hope could be transported to a theory of summaries and their independence and in- stantiation. And for yet another, in recent work Hoare andcolleagues have been investigating an algebraic theory of concurrency which identifies modularity principles for con- current processes related to separation logic, but based on separation of processes rather than states [44]. The theory isvery abstract, in that it does not say what kinds of entities the processes are; they are just elements of an algebra (subject to axioms). Perhaps there is an abstract algebra of summaries and their independence to be discovered.

It may be that the time is ripe to revisit or extend the ideasof [24], in light of the theoretical and practical developmentsthat have occurred since 2002.

Finally, *concerning (iv) and (v)*, these problems could wellbe more challenging than the others, but our earlier dis- cussions should make clear their relevance to the task of designing an OWA program analysis which can operate at diff time for large code.

Of course, not all of (i)-(v) need to be addressed in one go.

Conclusion

I moved to industry just under five years ago with the acquisi- tion by Facebook of the formal reasoning startup Monoidics, after spending over 20 years in academic positions mostly doing theoretical computer science. I've learnt many lessons in those five years, but two stand out.

One is the importance of scale: Scale in terms of code sizes and other resources, but even moreso in terms of people. In Facebook Engineering we are always on the lookout for techniques where deploying them to 10x people takes appre- ciably less that 10x in CPU time, watts and other resources ... but especially in terms of the ongoing effort needed by engineers (people) to maintain the technique. For example, if a static analysis team of size 10 supports 1000 product engi- neers, we wouldn't want to need 100 static analysis experts if the product engineering population were to grow to 10000. The other lesson is the importance of integration with pro- grammer workflow. A guiding principle: we want reasoning tools to serve people, not the other way around.

Of course, I had heard repeatedly about scale and workflow integration, but I did not really appreciate their significance back when I was a theorist. It all came together clearly for me in continuous reasoning. The continuous deployment of Infer showed striking benefits, in terms of developers responding to analyzer warnings, compared to the batch deployment. And the continuous deployment naturally supports scaling in terms of people. Scaling in terms of power or code sizes or other computing resources is still a hard problem, but one that is made a little easier by compositionality.

Reasoning about programs has come a long way since Hoare's grand challenge 15 years ago [43, 45], I daresay sur- prisingly far to those of us that were involved in discussions surrounding it. But there is still a long way to go for for- mal reasoning to have deep impact for many programmers. Certainly there are plenty of problems to work on, only one of which is continuous reasoning, but it is the problem that I have personally observed which, were progress to be made, seems like it could help further scale the impact of formal methods. I have described outstanding challenges in this paper, some more and others less precise, to give an idea of work that could be done, but my real hope is that if researchers engage with these issues they will be able to formulate better questions as well as answers over time.

Acknowledgments

Thanks to Josh Berdine and Sam Blackshear for advice on the material in this paper, and all my colleagues at Facebook for teaching me about applying formal reasoning in industry.

References

A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specificationsynthesis. In *POPL*, 2016.

G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*,pages 4–16. ACM, 2002.

T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey,

B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Eurosys*, pages 73–85, 2006.

L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *USENIX Security Symposium*, 2015.

S. Blackshear and P. O'Hearn. Open-sourcing RacerD: Fast static race detection at scale. code.facebook.com blog post.

C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca,
P.W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11, 2015.
C. Calcagno, D. Distefano, and P. O'Hearn. Open-sourcing Facebook Infer: Identify bugs before you ship. code.facebook.com blog post, 11June 2015.
C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, 2007.
C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
Q. Cao, L. Beringer, A. Gruetter, J. Dodds, and A. W. Appel. A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*., 2018.
S. A. Carr, F. Logozzo, and M. Payer. Automatic contract insertion with ccbot. *IEEE Trans. Software Eng.*, 43(8):701–714, 2017.
T. Chajed, H. Chen, A. Chlipala, M. F. Kaashoek, N. Zeldovich, and

E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample- guided abstraction refinement. In *CAV*, pages 154–169, 2000.
E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4th POPL, pp238-252, 1977.
P. Cousot and R. Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.
P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic infer- ence of necessary preconditions. In *VMCAI*, pages 128–148, 2013.
P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. 14th ESOP, pp21-30, 2005.
A. Das, S. K. Lahiri, A. Lal, and Y. Li. Angelic verification: Precise verification modulo unknowns. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2015.
L. de Moura and N.Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340. Springer, 2008.
D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.
T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300, 2013.
D. Ziegler. Certifying a file system using crash Hoare logic: correctness in the presence of crashes. *Commun. ACM*, 60(4):75–84, 2017.
R. Chapman and F. Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In *ITP*, pages 17–26, 2014.
R. Chapman, N. White, and J. Woodcock. What can agile methods bring to high-integrity software development? *CACM*, 60(10), 2017.
A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.
C. Y. Cho, V. D'Silva, and D. Song. BLITZ: compositional bounded model checking for real-world programs. In *ASE*, pages 136–146, 2013.
A. Chou. Static analysis in industry. POPL'14 invited talk. http://popl.mpi-sws.org/2014/andy.pdf.
A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffmanand C. Mac- Carthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and
E. Westbrook. Continuous formal verification of amazon s2n. In *CAV*, 2018.

L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. R. Murphy- Hill. Just-in-time static analysis. In *ISSTA*, pages 307–317, 2017.
R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and
A. Tomb. Constructing semantic models of programs with the software analysis workbench. In *VSTTE*, pages 56–72, 2016.
M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at Facebook. *Internet Computing, IEEE*, 17(4):8–17, 2013.
B. Fitzgerald and K.-J. Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 2017.
R. Giacobazzi. Abductive analysis of modular logic programs. In *Proceedings of the 1994 International Logic Programming Symposium*, pages 377–392. The MIT Press, 1994.
P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.
P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
M. Harman and P. O'Hearn. From start-ups to scale-ups: Open prob- lems and challenges in static and dynamic program analysis for testing and verification (keynote paper). In *International Working Conference on Source Code Analysis and Manipulation*, 2018.
C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, 2017.

C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In
E. Engler, editor, *Symposium on the Semantics of Algebraic Languages*, pages 102–116. Springer, 1971. Lecture Notes in Math. 188.
C.A.R. Hoare. The verifying compiler: A grand challenge for comput- ing research. *J. ACM*, 50(1):63–69, 2003.
T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6), 2011.
C. B. Jones, P. W. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.
I. T. Kassios. The dynamic frames theory. *Formal Asp. Comput.*, 23(3):267–288, 2011.
J. C. King. A program verifier. In *IFIP Congress (1)*, 1971.
G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin,
D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.
S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: opportunities, applications, and challenges. In *Workshop on Future of Software Engineering Research*, pages 201–204, 2010.
C. Larman and V. R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):47–56, 2003.
J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. D. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.
K. R. M. Leino. Accessible software verification with dafny. *IEEE Software*, 34(6):94–97, 2017.
K. R. M. Leino and M. Moskal. Usable auto-active verification. In
Usable Verification Workshop. http://fm.csl.sri.com/UV10/, 2010.
K. Mao, M. Harman, and Y. Jia. Sapienz: multi-objective automated testing for android applications. In *ISSTA*, pages 94–105. ACM, 2016.
K. L. McMillan. Applications of Craig interpolants in model checking. In *TACAS*, pages 1–12, 2005.
B. P. Miller, L. Fredriksen, and B. So. An empirical study of the relia- bility of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
D. A. Naumann. Deriving sharp rules of adaptation for Hoare logics, 1999. Stevens Institute of Technology, CS Rept 9906.
P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, pp1–19, 2001.
Amir Pnueli. The temporal semantics of concurrent programs. *Theor. Comput. Sci.*, 13:45–60, 1981.
D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium*, pages 49–64, 2015.
T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
J. C. Reynolds. Types, abstraction and parametric polymorphism. In
IFIP Congress, pages 513–523, 1983.
J. C. Reynolds. Separation logic: A logic for shared mutable datastructures. In *17th LICS*, pp 55-74, 2002.
C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at Google. *CACM*, 2018.
M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *ESOP*, volume 7792, pages 451–471, 2013.
F. W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017.
F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. CAV, 2016.
H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and
P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.
G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234. ACM, 2008.