NISTIR 8274

Formal Methods for Statistical Software

Paul E. Black

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8274

**National Institute of
Standards and Technology**
U.S. Department of Commerce

NISTIR 8274

Formal Methods for Statistical Software

Paul E. Black
Software and Systems Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8274

October 2019

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Undersecretary of Commerce for Standards and Technology

National Institute of Standards and Technology Interagency or Internal Report 8274
Natl. Inst. Stand. Technol. Interag. Intern. Rep. 8274, 26 pages (October 2019)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8274

Abstract

"Statistical software" encompasses several distinct classes of software. This report explains what formal methods, tools, and approaches may be able to increase assurance of results of using statistical software and implementing differential privacy. To provide context, we present an exemplary process for assured results. The parts are data assurance, algorithm design, software production, correctness proofs, postproduction assurance of software, and result checking. We note a workshop we organized to support this paper and finish with recommended formal methods, tools, and researchers doing particularly pertinent work.

Key words

correctness proofs; formal methods; software assurance; differential privacy; software quality; static analysis; static source code analyzers; statistical software.

Table of Contents

1. Introduction

Software may produce outputs that appear reasonable, but are nonetheless incorrect. For instance, a simplistic pseudorandom number generator may offer sufficient randomness for a video game, but its use in a Monte Carlo simulation yields invalid results. Such problems are hard to find through traditional testing. Formal methods can complement testing to gain greater assurance that critical portions or aspects of programs are correct.

Statistical software has not been well studied by those using formal methods. By "statistical software," we include software that has particular features or characteristics that challenge typical means of software assurance. These features are:

- randomized algorithms, such as Monte Carlo simulations,
- algorithms effecting differential privacy[1], where the proper operation depends on the availability of unbiased and unpredictable random numbers, and
- summarizing algorithms, which compute mathematical properties of large sets of data, such as sums, averages, and regressions.

To be comprehensive, we also indicate formal methods that may help problems that statistical software shares with many other kinds of software. These include gaining assurance that software faithfully implements algorithms and models. While there have been significant efforts within the statistical community to examine underlying statistical models and algorithmic efficiency, there has not been a commensurate study to verify correctness.

We use the term "formal methods" to include analytic approaches relying on mathematical and logical reason to establish the applicability of properties. Testing demonstrates results only in discrete circumstances. However, like proofs of trigonometric identities, formal methods employ rules of inference, along with assumptions and models, to argue for assurance in all cases meeting those assumptions. Formal methods applications range from basic and automated, like LALR parsing or type checking, to sophisticated and undecidable, like program correctness.

This is a report for improving the quality of statistical software through the use of formal methods and techniques that are:

- Currently available in tools, in prototypes, or as proofs of concept,
- Cost-effective for the challenges faced, and
- Complementary to current testing approaches.

This report also lists some key research and researchers working in the field of formal methods for statistical software. We note the current state of the art and recommend research, tools, and techniques for increasing confidence in the results of statistical software.

[1]Section 2.1 describes differential privacy.

Section 4 is a brief summary of the 1 May workshop that we organized to provide input to this report.

Note: we often discuss software implementations, chances of errors in execution, and so forth. Detailed discussions of real systems must consider myriad causes of errors: compiler bugs, malicious intrusion, transient hardware faults, and human misunderstanding of results. In general, our discussions ignore such external influences in order to present and discuss concepts without tedious and repetitive phrases such as "assuming no hardware faults" or "in the absence of compiler or library bugs".

2. Scope: What is "Statistical Software"?

We use the term "statistical software" as a short phrase to cover many aspects of statistics, data handling, privacy, data analysis, software, and computation. The scope could be argued to be a "grab bag" of subjects. We make no pretense that these are somehow a coherent whole, except that they are of interest to those in the U.S. Census Bureau. Nevertheless, we can name characteristics and attributes that arise.

Projects involving statistical processing of data may include many steps [1]. Some of the steps are design of experiments or surveys, planning how to collect or generate the data, collecting or generating it, data cleaning and munging [2], modeling and analyzing the data, pattern mining and refining algorithms, iterating through these steps, and learning from results. Although critical for valid results and deserving of further research, these concerns are outside our scope. Typical approaches to software assurance, such as verification and validation, provenance, and proof-carrying code, might be incorporated into such data science. Algorithms, and the software to implement them, appear in many forms and at many places in those steps.

Random numbers are not thought of as input data, nor are they strictly part of algorithms. But the quality of the random numbers used for sampling or simulation are vital for good results. Considerations of random numbers are outside our scope, too.

2.1 Differential Privacy to Limit Privacy Loss

A particular challenge at the U.S. Census Bureau is publishing statistics that guarantee limited loss of individual privacy. Suppose someone examines the published statistics for a tract that has just a few people. Queries about such small sets should not yield correct information. Queries about large sets, such as whole states, can yield information that is close to correct without compromising privacy. One way to accomplish this "is to make sure that every computation on sensitive data satisfies *differential privacy*, a very strong guarantee: if an individual's data is used in a differentially private computation, the probability of any given result changes by at most a factor of e^{ϵ} (compared to the situation where this individual's data is not used), where ϵ is a parameter controlling the trade-off between privacy and accuracy." [3]. Differential privacy includes a mathematical definition of privacy loss.

Differential privacy can be maintained by adding structured noise—values that are random at small scales but have large scale properties. Many algorithms and specific mechanisms have been developed allowing the addition of the smallest amount of noise necessary for a given privacy outcome and with structure to have minimal impact on large scale statistics. Algorithms in this field typically have good mathematical foundations, and there is much active research, see for instance Shaikh and Patil [4].

Although there are solid theoretical foundations for computations maintaining differential privacy, implementations can have subtle bugs. These bugs leak confidential data into supposedly privatized outputs. This problem is similar to the usual software assurance question, that is, does the implementation satisfy the specification? Formalized principles of differential privacy allow confidence when creating a system for applying differential privacy to the national census, see Sec. 3.4.3.

2.2 Randomized Algorithms

In the class of statistical software, we include randomized algorithms. A *randomized algorithm* is an "algorithm that makes some random (or pseudorandom) choices." [5] For our purposes, randomized algorithms fall into three groups: Monte Carlo, Las Vegas, and probabilistic algorithms.

2.2.1 Monte Carlo Algorithms

Monte Carlo algorithms produce results that differ from the correct results by some small amount. Often for numeric algorithms, the longer they run, the smaller the expected difference.

Monte Carlo algorithms may be illustrated by the following problem. Given two unbiased coins, what is the chance of flipping two heads at the same time? We may estimate the answer by flipping the coins and recording the results. More flips tend to produce a fraction closer to the correct result, which is $1/4$.

2.2.2 Las Vegas Algorithms

Las Vegas algorithms are guaranteed to always produce the correct results, but the run time may vary.

Las Vegas algorithms may be illustrated by the following. Suppose Alice has a deep drawer with loose—unpaired—blue socks and brown socks. The drawer is too dark to see the socks, so Alice picks socks randomly. Alice can get two blue socks eventually by continuing to pick socks until she has two blue ones. The outcome is guaranteed, but the procedure is slow if she is unlucky.

2.2.3 Probabilistic Algorithms

Probabilistic algorithms almost always produce the correct results. There is a small chance that they produce incorrect results, but the chance is vanishingly small (or can be made as small as desired).

Probabilistic result may be illustrated by the following. Suppose you need to choose the size of table for a marathon water station. It is impractical to have a table big enough to have cups of water for *all* the runners at once. A small table will be emptied by a surge of runners before it can be refilled. A moderate sized table is big enough for all practical cases, even though it could theoretically be overwhelmed by a highly unlikely surge.

2.3 Libraries of Functions

Much statistical software uses libraries of standardized mathematical functions, which come with most programming languages. Although most are generally serviceable, they may not be adequate for very demanding applications. This is particularly true of random number generators. In addition, there may be implementation bugs that generate results appearing to be correct, but which fail in unpredictable ways. An example is the Pentium FDIV hardware bug [6].

3. An Exemplary Process for Assured Results

To provide a context, this section presents an exemplary process for achieving assured results. We focus on topics that are especially applicable to the U.S. Census Bureau.

Confidence in the validity of results comes from four sources:

1. confidence in the quality of the data, treated in Sec. 3.1,
2. good algorithm design or selection, Sec. 3.2, and good software production, Sec. 3.3, both accompanied by proofs of correctness, Sec. 3.4,
3. post-production assurance of software quality, that is, testing and static analysis techniques, Sec. 3.5, and
4. computation or result checking, Sec. 3.6.

3.1 Data Assurance or Checking

The best software in the world is of little use if the data has not been collected, cleaned, and prepared properly. This is a vital matter, but is outside the scope of this paper. There are several introductions and references such as Groves et al. "Survey Methodology" [7] and Pfaff et al. "Data Validation in the Presence of Stochastic and Set-membership Uncertainties" [8].

For reliable querying after differential privacy operations, raw data often requires extensive cleaning. This can be a considerable burden on the data provider to ensure that data is completely clean before adding differential privacy noise. Krishnan et al. propose [9] a technique for creating private datasets of numerical and discrete-valued attributes, and answering `sum`, `count`, and `avg` queries after data cleaning.

3.2 Design or Selection of Algorithm

Good software begins with an overall structure, or architecture, that reflects constraints such as the structure of problem to be addressed, modularity and abstraction, anticipated changes, and existing packages or systems. Such high-level concerns are critical, but outside the scope of this paper.

The next level of detail is to design, develop, or select acceptable algorithms. Some analysis algorithms are sensitive to certain patterns or discrepancies in input data. As a very simple example, consider finding the sum of a large set of real numbered values. The simplest approach is to add up the values as encountered. However, very large values can make the running total so large that smaller values are rounded and precision lost. Here are the first few and last few values of an artificial example with hundreds of thousands of data values:

12 345 678 912 345 678 912
.173 101 13
.162 085 22
.208 242 39
.
.
.
.990 043 03
.169 798 8
.219 873 47
-12 345 678 912 345 678 912

Between the huge beginning and ending values are values from 0 to 1. The sum of all the values is approximately 53 194.907.

Here is a simple algorithm in R to sum the values. The result is far from correct, but might mistakenly be considered to be reasonable.

```
> data <- scan("examp1.data")
> sum(data)
[1] 38581
```

However, if we add up the smallest values first (that is, sort by magnitude), we get a much more accurate result.

```
> sum(data[order(abs(data))])
[1] 53195
```

In Python 3, the simple approach gives a result that is clearly incorrect.

```
>>> with open("examp1.data") as f:
...     print(sum(list(map(float, f))))
0.0
```

As above, sorting by magnitude gives a more accurate result.

```
>>> with open("examp1.data") as f:
...     print(sum(sorted(list(map(float, f)), key = abs)))
53248.0
```

Another simplistic example is that the arithmetic mean is susceptible to outliers. Median is much less sensitive and is therefore often better to use as an "average" for a data set. A realistic example is that the distribution and extremes of latency of a computer service are as important or more important than any aggregation, such as average [10].

"Robust statistics" is the study of algorithms that are less sensitive to small perturbations. [1]

Methods and approaches for numerical analysis may be crucial to designing acceptable algorithms. Numerical analysis deals with concerns such as convergence, error estimates and error budget, managing computations to increase accuracy (or, widen the types of data sets to which it is applicable).

Yet another general approach is to write, transform, or execute algorithms like the above in such a way that numerical errors are likely to be avoided or detected. We cite a number of tools and approaches for elaborating algorithms to achieve this.

Graillat et al. analyze [11] the impact of random rounding mode on *compensated algorithms*. They show the increase in accuracy of computations when an error-free transformation is used to "enable the computation of rounding errors".

CADNA is a library that allows the developer to estimate round-off errors in numerical simulations [12, 13]. It uses Discrete Stochastic Arithmetic, which runs a computation several times with random rounding modes then estimates the number of significant digits. Using CADNA requires only a few modifications to code: essentially declaring variables differently.

Herbie is "a tool which automatically discovers the rewrites [that] experts perform to improve accuracy." Rewrites are "rearranging expressions and understanding the finer details of floating point arithmetic" "to mitigate rounding error" [14, 15].

Michael Lam has a Floating-Point Analysis Research web page [16], which lists these and other efforts.

In coming up with the appropriate algorithm, one must ask how the results can ultimately be checked. (We treat this in more detail in Sec. 3.6.) How can one check that 23.121 805 *is* the standard deviation of a data set? What if the algorithm works well for a normal or linear distribution of data, but is grossly inaccurate if the data is distributed along a parabola?

Using a second, independent algorithm as an oracle is not always straight forward. Two algorithms or two implementations of the same algorithm may give different results. How can we determine if two computations are consistent?

Even though frequently-used languages such as R, SAS, and Stata are higher level than C or Python, an even higher level, domain-specific language for statistics and publication would reduce errors. It could be a non-procedural specification for frequent activities. For instance, there could be a language to specify plots for publication, from analysis to output format. The specification is then compiled into R or SAS code. Since it is a specification, low-level mistakes are far less likely.

Although language design is not typically associated with formal methods, languages should be designed from the beginning with a formal semantics. Like WebAssembly [17], such languages could be designed with little or no undefined behavior, making it much more amenable to formal analysis.

3.3 Good Software Production

Quality cannot be tested into software. Software must be built well from the beginning. A disciplined team of experienced developers is necessary to the production of good software and ultimately good results.

A good software development process includes defining requirements and specifications, allocating sufficient resources, tracking versions, and incorporating good software test and assurance techniques.

When the development team uses outside software, they must rely much more heavily on appraising the software for assurance, establishing its suitability, and ensuring that it is only used in domains for which it is valid. See Sec. 3.5 for more detail.

No single activity, approach, or tool can assure that the final results produced are of the highest quality. As Jones points out [18], different methodologies are appropriate to small projects in contrast to large projects, different approaches and tools address different concerns, and different techniques and paradigms should be applied at each project stage. Although slightly dated, [18] includes existing and widely-used steps as well as new ideas that can be adopted and emerging technology. It also identifies gaps and needed research.

There are many other good guides to the software development process, such as Ref. [19] and the pair of Guidebooks for Software Assurance for the program manager [20] and for the developer [21]. The Defense Innovation Board's May 2019 report, "Software is Never Done: Refactoring the Acquisition Code for Competitive Advantage" [22] has many useful ideas and suggestions. For instance, one concept paper that it includes, "Is Your Development Environment Holding You Back?", names many useful development practices and tools. Another concept paper, "Do's and Don'ts for Software", advises developers to create automated test environments to enable rapid, continuous, and assured software updates and to require source code as a deliverable. A third concept paper "Metrics for Software Development" gives modern project metrics *with target values*. Vignette 5 showcases an Air Force software company as an example of outstanding software development and delivery.

3.4 Proof of Correctness

A mechanically checked proof of correctness of an algorithm or implementation can add invaluable assurance. Testing is important, but it only checks a minuscule bit of possible behavior. A correctness proof covers all possible cases, at least in theory. Human code review is also an excellent complement to software testing, but is subject to unconscious assumptions or lapses in concentration. In contrast, a proof in an automated system requires explicit assumptions and no unwarranted leaps of logic.

In the past, formal methods, such as proofs, had a reputation for taking far too long and requiring a graduate degree in computer science or mathematics to perform them. This is no longer the case. Today, formal methods are widely used. See Section 2.1 of [23] for examples.

A formal proof of correctness may be broken into several steps, see Fig. 1. The verifier is a tool that produces much of the proof, but may need expert guidance. Witnesses for correctness are invariants for loops, data structures, or critical program state at particular locations in the code. Beyer et al. explain [24] what correctness witnesses are, how their use compares with similar work, and how they may be constructed and validated. The following is their example code in their Fig. 1a:

```
unsigned int y = x;
while (x < 1024) {
    x = x + 1;
    y = y + 1;
}
```

When the loop ends, x and y should be the same. Beyer et al. state, “proving the whole program correct is easy if the loop invariant x = y is given, but finding such a loop invariant is in general hard”.

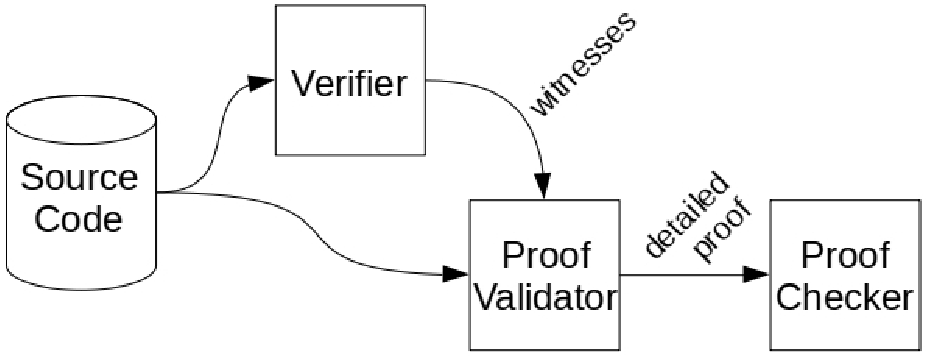

Fig. 1. With human guidance a verifier tool produces witnesses. A proof validator makes sure the witnesses suffice to prove that the code is correct. An independent (and simple!) proof checker may double check each step of the validator.

Invariants and related concepts, such as contracts, preconditions, assertions, and postconditions, can be embedded as executable statements in the code. “Programmers generally have a body of information that gives them confidence that software will perform as expected. A neglected part of formal methods is to unambiguously record such insights.” [23, Sec. 2.1.3].

The code and the verification may be developed together. For instance, if a piece of code cannot be proven correct by the verifier, it might be rewritten so that the verifier *does* prove it. For some purposes, it may be desirable for the verification and development of witnesses to be done by an expert third party.

Why not deliver a proof, instead of witnesses? First, complete mechanical proofs can be *very* long and do *not* help humans understand the properties and invariants of the program. Second, provers have very different internal structures (axioms, rules of inference, representations), so finding a common language is unlikely.

Once the algorithm is verified, implementation can proceed. A proof that the code satisfies the requirements and implements the algorithm should be developed in parallel with the code development. Proving, or trying to prove, chunks of code, functions, and modules tends to inform the coding and test planning about points that may not be clear. Simultaneous proving may also guide the programmer to choices that make proofs clearer and easier.

With subtle and demanding requirements for data handling and computations, a program can easily produce a result that appears to be reasonable, but is far from accurate[2]. There are many resources and tools for software assurance for programs written in widely-used languages, such as C, Java, and C#. Many statistical and analysis programs of concern are written in languages that do not have as much support for program analysis and proofs of correctness, such as SAS, R, Stata, and Python. This is beginning with the R language, see Bodin, Diaz, and Tanter [25].

3.4.1 Model Checkers and Other Lightweight Formal Methods

In addition to proof systems for higher-order logic, model checkers, SAT solvers and other "light weight" decision techniques can answer questions about properties of code and the algorithms they implement. For instance, Katoen provides a survey of model checking of probabilistic models, in particular Markov decision processes (MDPs). An MDP is a Kripke structure in which transitions have "probability distributions over states as targets" [26].

These lightweight formal methods can also analyze higher-level artifacts, such as state machines from the design, before code is written.

For an overview of formal methods, including suggested techniques and further reading, we refer the reader to Section 2.1 of [23].

3.4.2 Proofs of Algorithms and Implementations

Before an algorithm is implemented, that is, before code is written, developers should be confident that the algorithm will satisfy the requirements (if implemented correctly). For instance, if the algorithm does not converge for some distributions of data or gives the wrong answer for sparse graphs, it makes little sense to write code for it.

There are many general methods for correctness proofs. Here we only refer to a few recent resources that may be particularly useful.

[2]recall the examples in Sec. 3.2

In January 2019 Marco Gaboardi gave a three-hour mini-course on “Formal Methods and Proofs of Privacy Properties” at *Data Privacy: Foundations and Applications Boot Camp*, Simons Institute for the Theory of Computing, University of California, Berkeley. Videos of the mini-course are on-line [27–29].

Fenske et al. propose [30] “a cryptographic protocol for efficiently aggregating a count of unique items across a set of data parties privately—that is, without exposing any information other than the count.” They “also show how the output can satisfy differential privacy.”

Rogers et al. study adaptive composition in differential privacy algorithms [31]. They define privacy filters, stopping time rules, and privacy odometers, a method to track realized privacy loss.

Costea et al. propose [32] a differential privacy algorithm that protects sequential data, such as histories.

Wang et al. present two provably private subspace clustering algorithms that respect differential privacy [33].

He et al. propose [34] a new privacy model, output constrained differential privacy, that uses differentially private blocking, “but allows for the truthful release of the output of a certain function applied to the data”, such as private record linkage.

A different domain of concern is that sampling-based (Monte Carlo) techniques, such as statistical model checking, are inefficient for estimating the probabilities of rare events. Zuliani et al. show [35] how to use the cross-entropy method for generating approximately optimal biasing densities for statistical model checking. They apply the method with importance sampling and statistical model checking to estimate the probabilities of rare events in stochastic hybrid systems.

Lan et al. show [36] “that while running a Mirror Descent Stochastic Approximation procedure one can compute, with a small additional effort, lower and upper statistical bounds . . . ”. While this applies to very specific computations, the approach may be more broadly useful.

3.4.3 Proofs of Differential Privacy

There are many approaches to prove the correctness of algorithms for differential privacy. The following prove correctness via probabilistic couplings [37, 38], by *-lifting [39], via modeling as labeled Markov Chains [40], and with shadow execution [41].

Barthe et al. review [3] ways to verify differential privacy of programs. The last way, HOARe^2, “combines the advantages of” the other two ways presented: DFuzz and `CertiPriv` [42]. Zhang and Kifer propose a simple language to express differential privacy algorithms and verify “sophisticated algorithms with little manual effort” [43].

Following are two examples of verifying other kinds of algorithms. Boldo et al. gave [44] a formal correctness proof in Coq of a new renormalization algorithm, which is used in floating-point expansion (FPE). Hasan and Tahar present [45] the formal verification of Markov’s and Chebyshev’s inequalities for discrete random variables in HOL. The paper also gives proofs of mean and variance relations for some of the widely used discrete random variables, such as Uniform(m), Bernoulli(p), Geometric(p) and Binomial(m, p) random variables. Rand and Zdancewic present [46] a Hoare-style logic, which can handle full distributions, to prove the correctness of probabilistic programs.

3.5 Postproduction Assurance of Implementations

There are two distinct ways to approach gaining assurance that a piece of code has desired properties: testing and static analysis. (Hybrid tools combine these two ways.) These two approaches are complementary. Static analysis considers the software, either source code or binaries, from an abstract point of view. In theory *all* possible executions are covered. Manual static analysis, commonly called code reviews, is the single most effective measure for finding problems in code. Automated static analyzers can be run (and rerun!) on millions of lines of code and look for hundreds of different classes of bugs. They can also follow long, subtle lines of reasoning without getting bored or inadvertently losing track of small, crucial details. Static analysis can also be done on blocks of code or subsystems, without waiting for an entire skeletal system to be operational. One limitation is they are based on assumptions about compilers and hardware. This is important because the recent Meltdown bug violated assumptions.

In contrast, testing exercises software as installed and run. Of necessity, tests depend on compilers, libraries, hardware and operating systems, installation environment, and other concrete options. Test cases run end to end, engaging interfaces between modules and subsystem. This may expose mismatches that were overlooked earlier. An important limitation is that testing can only run a microscopic fraction of all possible inputs and states. Another critical limitation is that some kinds of aberrant code cannot feasibly be found through testing. A backdoor in code that grants full access when the user enters the name, JoshuaCaleb, cannot possibly be discovered by testing. There are far too many such booby traps to cover. These can only be found by static examination of the code or executable.

3.5.1 A Model of Automated Testing

A conceptual model of automated testing has three parts: test creation or generation, an oracle, and a comparator, see Fig. 2. Thorough or extensive testing may require automated or semi-automated generation of thousands of test cases. Testing requires some oracle to produce the correct result for each case. This oracle may be a previous presumed-good version of the code, a very simple algorithm that is far too slow for production, the code itself run on a property-preserving variation of the test case, or specially written code implementing only the most critical computations. A final comparator part reports any difference between the code's result and result from the oracle. Comparison may be quite involved if results are allowed to differ by some ϵ or only part of the result is checked.

Each of the three parts may be trivial, straight forward, or extremely complex, and the parts may be combined. For example, the generator may produce the correct result as each case is created, eliminating the need for a separate oracle.

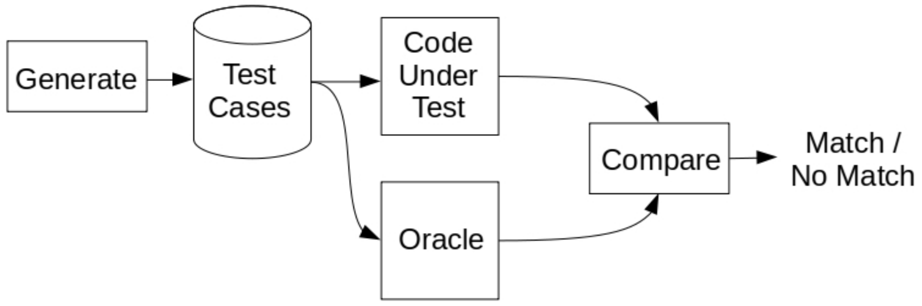

Fig. 2. Conceptually, automated testing has three parts. A generator creates test cases. An oracle produces the correct result for each test case. A comparator determines whether the code's result matches the comparator's.

One example of a generator is GENTHAT, which generates unit tests from executions of R code [47]. The oracle may be known-good references. For instance, Keeling and Pavur test [48] the accuracy of six spreadsheet packages using the NIST Statistical Reference Datasets [49] and Wilkinson's Tests.

Instead of repeating or trying to summarize the vast literature on testing, we encourage the reader to consult publications devoted to it. For instance, Jones [18] Chap. 2 Best Practice #37 lists many forms of testing and Pressman and Maxim [19] devote chapters 22 through 26 to testing.

3.5.2 Testing Non-Deterministic Programs

Elbaum and Rosenblum point out [50] that statistical software has special challenges for testing. Checking test results must accept ranges of results in some cases. However, liberal acceptance may miss correctable bugs, such as an unintended bias. They offer suggestions about possible ways to perform and analyze tests to detect bugs.

To unit test a randomized algorithm, it may be split into two concerns. First, does it do the right thing *given a specific set of "random" numbers*? This tests the function built *on top of* the random number generator. Second, any random number generator used to feed the algorithm must be tested, for example, with NIST's Statistical Test Suite (STS) [51, 52], ent [53], dieharder [54], etc.

Simulations and sampling are *expected* to produce different results every time. Formal verification, proofs of correctness, and mathematical arguments of expected values and error limits are the most reliable approaches. To supplement such static analysis, statistical testing, like that done for random or pseudorandom number generators, may provide additional assurance. Rerunning software with the same pseudorandom number generator seed and test set of data is a weak test. It shows that the software did not get *worse*, but it produces little information about what the software does for other sets of data or what bugs might be revealed by other seeds.

3.5.3 Sound and Heuristic Static Analyzers

"... static analysis has the potential to efficiently preclude several classes of errors in newly-developed software and to reduce the uncertainty regarding resources needed to reach higher levels of assurance through testing." [23, Sect. 2.1.1]

Automated tools use two types of static analysis: heuristic and sound. Heuristic static analysis checks rules that are usually followed, but not strictly required for a bug-free program. An example rule is that if a file is opened in a function, it should be closed in that same function. Perfectly fine software might not follow this rule, but obeying this guideline greatly reduces the chance of a bug leaking resources.

Sound analysis is based on strict models and formal mathematical reasoning. Sound analysis can provide far higher levels of assurance. However detailed processing of possible control flows and variable values may prevent software larger than about one hundred thousand lines of code from being checked. In addition, some rules, such as "no hard-coded passwords" or "always store sensitive data encrypted" cannot have rigorous definitions or depend on external specifications and policies. As such, heuristic static analysis complements sound static analysis.

There are static analyzers for languages like R. The work of Bodin, Diaz, and Tanter [25] lays a strong foundation for further analyzers that are sound. Several static code analyzers handle Python, but they generally only detect low-level bugs.

3.6 Result Checking

Verifiable computation requires that a result be accompanied by a proof of correctness. To be broadly practical, transforming the program to compute the result, preparing the input, and checking the proof must cost much less than performing the computation. Verifiable computation is also called "checking computations" or "certified algorithms". A 2015 survey [55] reports that some approaches return succinct proofs and others can be used with any computer program. It also notes rapid recent progress and is optimistic about the future.

The ultimate assurance is achieved if the result can be cross checked in some fashion or the computation also yields a certificate. Designing programs to check their results is a classic approach [56] that has not been widely employed. Limited checking, such as Costea and Tapus' "differential privacy algorithm that signals the user when relative errors surpass a predefined threshold" [57], provides some assurance that results have desired properties.

4. Workshop Summary

To provide input to and comment on this report, we organized a one-day workshop. It was held 1 May 2019. Simson L. Garfinkel, U.S. Census Bureau, Research and Methodology Directorate, was a co-chair. The workshop, Formal Methods for Statistical Software (FMfSS), met in Annapolis, Maryland. We thank the High Confidence Software and Systems Conference, which provided registration support, the meeting space, and breaks. The

URL of the workshop page is https://samate.nist.gov/FMSwVRodeo/FMfSS2019.html

The purpose of the workshop was to identify formal methods, tools, and techniques that can be used now (or have the potential in the near future) to gain higher assurance of statistical software and results with reasonable resources.

We called for brief position statements on topics like:

- types of errors, bugs, and failures in statistical software or its use.
- formal methods that can assure statistical software or its results.
- correct-by-construction approaches.
- lists or definitions of important properties, such as convergence and differential privacy.
- gaps in current capabilities and specific research needed.
- higher-level, non-procedural languages or tools so a statistician can specify the desired analysis instead of how to achieve it.
- approaches to gain assurance in statistical software and its use.

The program committee reviewed submitted statements. On the basis of those statements, we invited some submitters to make a presentation. The program consisted of those who accepted, discussion, and additional invited presentations. About 50 people from academia, industry, and government attended.

Here is the final agenda.

0900 Welcome and definitions; Individual introductions and expectations (all participants), and a presentation of Keeling and Pavur, *Statistical Accuracy of Spreadsheet Software*.

1005 *A Trustworthy Mechanized Formalization of R*, Martin Bodin, Imperial College

1030 break

1100 *Formalizing Statistical Computation in HOL4*, Jared Yeager, University of Massachusetts Amherst

1120 *Trustworthy Data Wrangling and Analysis*, Eric Davis

1140 *The Challenges of Using SAST for Validation and Verification*, Arthur Hicken, Parasoft

1200 lunch

1330 *Intro to Differential Privacy*, Simson L. Garfinkel

1340 *Assuring Real-World Differential Privacy*, José Calderón, Galois

1405 *Detecting Violations of Differential Privacy*, Zeyu Ding, Penn State

1430 *Discussion: Future of Formal Methods in Differential Privacy*, Simson L. Garfinkel

1500 brief break

1515 *A Formal Methods Tools Rodeo*, Paul E. Black

1530 *Static Analysis Challenges: Code Coverage, Warning Classes, and Types of Checkers*, Arthur Hicken, Parasoft

1555 *Formal Methods in Statistical Work*, Paul E. Black

The workshop ended at 1600.

There were many good discussions and exchanges of information between participants. Many of the comments, questions, and suggestions improved this report.

5. Recommendations and Conclusions

The following formal methods should particularly contribute to U.S. Census Bureau work. We recommend that they be adopted and research be funded and otherwise encouraged.

Sound static analysis of software, Sec. 3.5.3. It has the potential to automatically find many classes of bugs.

Higher-level, domain-specific languages designed from the beginning with a formal semantic, Sec. 3.2. Model-based software development uses such languages. They could completely eliminate the possibility of low-level bugs.

Correctness proofs, Sec. 3.4. Developing proofs along with code exposes assumptions and can yield very low bug rates.

Collaborate with other federal agencies to characterize formal methods tools and their use in software development and acceptance. Part of this collaboration is to use and extend existing test sets, such as those for SV-COMP and RERS.

The following researchers are doing work particularly pertinent to applications in the U.S. Census Bureau. We list them in the groups in which they collaborate. We recommend maintaining close contact with them and establishing future collaborations.

Gilles Barthe and Pierre-Yves Strub, Institutos Madrileños de Estudios Avanzados (IMDEA) Software Institute, Marco Gaboardi, University at Buffalo, State University of New York (SUNY), Benjamin Grégoire, Inria Sophia Antipolis, Justin Hsu, University of Pennsylvania, and Tetsuya Sato, Research Institute for Mathematical Sciences, Kyoto University.

Sergiu Costea and Nicolae Tapus, University POLITEHNICA of Bucharest.

Ryan Rogers and Aaron Roth, University of Pennsylvania, Jonathan Ullman, Northeastern University, and Salil Vadhan, Harvard University.

Sebastian Elbaum, University of Nebraska - Lincoln, and David S. Rosenblum, National University of Singapore.

Michael Lam, James Madison University.

Formal methods for software development are based on mathematics and logic. They have the potential to reduce bugs and errors and increase assurance of statistical and database products. Considering the time needed to test software and the schedule uncertainties resulting from bugs, adoption of formal methods may actually reduce the time and resources needed to develop high-quality software and results worthy of the U.S. Census Bureau.

Acknowledgments

This work was funded in part by the U.S. Census Bureau under interagency agreement 0095-2018-PAY-01. Simson L. Garfinkel, U.S. Census Bureau, Research and Methodology Directorate, was invaluable in directing this work and making comments on this report.

We thank the High Confidence Software and Systems Conference, which provided registration support, the meeting space, and breaks for the associated workshop.

References

[1] Scott JH (2018), private communication.

[2] Skiena SS (2017) *The Data Science Design Manual*. Texts in Computer Science (Springer, Cham), Chapter 3 Data Munging, pp 57–93. https://doi.org/10.1007/978-3-319-55444-0_3

[3] Barthe G, Gaboardi M, Hsu J, Pierce B (2016) Programming language techniques for differential privacy. *ACM SIGLOG News* 3(1):34–53. https://doi.org/10.1145/2893582.2893591

[4] Shaikh A, Patil S (2018) A survey on privacy enhanced role based data aggregation via differential privacy. *2018 International Conference On Advances in Communication and Computing Technology (ICACCT)*, pp 285–290. https://doi.org/10.1109/ICACCT.2018.8529634

[5] Black PE (2001) randomized algorithm. *Dictionary of Computer Science, Engineering and Technology*, ed Laplante PA (CRC Press LLC), p 404.

[6] Cipra B (1995) How number theory got the best of the Pentium chip. *Science* 267(5195):175–175. https://doi.org/10.1126/science.267.5195.175

[7] Groves RM, Fowler FJ Jr, Couper MP, Lepkowski JM, Singer E, Tourangeau R (2009) *Survey Methodology* (John Wiley & Sons, Inc.), 2nd Ed.

[8] Pfaff F, Noack B, Hanebeck UD (2013) Data validation in the presence of stochastic and set-membership uncertainties. *Proc. 16th International Conference on Information Fusion (FUSION)*, pp 2125–2132.

[9] Krishnan S, Wang J, Franklin MJ, Goldberg K, Kraska T (2016) Privateclean: Data cleaning and differential privacy. *Proc. 2016 International Conference on*

Management of Data (SIGMOD'16) (ACM, New York, NY, USA), pp 937–951. https://doi.org/10.1145/2882903.2915248

[10] Jones C, Wilkes J, Murphy N, Smith C (2016) *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly Media, Inc.), Chapter 4 Service Level Objectives. Available at https://books.google.com/books?id=81UrjwEACAAJ.

[11] Graillat S, Jézéquel F, Picot R (2015) Numerical validation of compensated summation algorithms with stochastic arithmetic. *The Seventh and Eighth International Workshops on Numerical Software Verification (NSV)*, eds Bogomolov S, Martel M (Elsevier B. V.), *Electronic Notes in Theoretical Computer Science*, Vol. 317, pp 55–69. https://doi.org/10.1016/j.entcs.2015.10.007

[12] What is CADNA? Accessed 10 May 2019 Available at http://cadna.lip6.fr/.

[13] Eberhart P, Brajard J, Fortin P, Jézéquel F (2015) High performance numerical validation using stochastic arithmetic. *Reliable Computing* 21:35–52.

[14] Panchekha P, Sachez-Stern A, Wilcox JR, Tatlock Z (2015) Automatically improving accuracy for floating point expressions. *Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, eds Grove D, Blackburn S (ACM), pp 1–11. https://doi.org/10.1145/2737924.2737959

[15] Herbie at PLDI'15. Accessed 2 May 2019 Available at https://herbie.uwplse.org/pldi15.html.

[16] Lam M (2017) Floating-point analysis research. Accessed 13 May 2019 Available at https://w3.cs.jmu.edu/lam2mo/fpanalysis.html.

[17] Rossberg A, Titzer BL, Haas A, Schuff DL, Gohman D, Wagner L, Zakai A, Bastien JF, Holman M (2018) Bringing the web up to speed with WebAssembly. *Comm ACM* 61(12):107–115. https://doi.org/10.1145/3282510

[18] Jones C (2010) *Software Engineering Best Practices: Lessons From Successful Projects in the Top Companies* (McGraw-Hill).

[19] Pressman RS, Maxim B (2015) *Software Engineering: A Practitioner's Approach* (McGraw-Hill), 8th Ed.

[20] Nidiffer KE, Woody C, Chick TA (2018) Program manager's guidebook for software assurance (Software Engineering Institute, Carnegie Mellon University), Special Report CMU/SEI-2018-SR-025. Available at https://resources.sei.cmu.edu/asset_files/SpecialReport/2018_003_001_538779.pdf.

[21] Nichols WR Jr, Scanlon T (2018) DoD developer's guidebook for software assurance (Software Engineering Institute, Carnegie Mellon University), Special Report CMU/SEI-2018-SR-013. Available at https://resources.sei.cmu.edu/asset_files/SpecialReport/2018_003_001_538761.pdf.

[22] (2019) Software is never done: Refactoring the acquisition code for competitive advantage. Parts accessible at https://innovation.defense.gov/software/. Available at https://media.defense.gov/2019/Apr/30/2002124828/-1/-1/0/SOFTWAREISNEVERDONE_REFACTORINGTHEACQUISITIONCODEFORCOMPETITIVEADVANTAGE_FINAL.SWAP.REPORT.PDF.

[23] Black PE, Badger L, Guttman B, Fong E (2016) Dramatically reducing software vulnerabilities: Report to the white house office of science and technology policy (National Institute of Standards and Technology), NISTIR 8151. https://doi.org/10.6028/NIST.IR.8151

[24] Beyer D, Dangl M, Dietsch D, Heizmann M (2016) Correctness witnesses: Exchanging verification results between verifiers. *Proc. 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* FSE 2016 (ACM, New York, NY, USA), pp 326–337. https://doi.org/10.1145/2950290.2950351

[25] Bodin M, Diaz T, Tanter E (2018) A trustworthy mechanized formalization of R. *Proc. 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2018)* (ACM, New York, NY, USA), pp 13–24. https://doi.org/10.1145/3276945.3276946

[26] Katoen JP (2008) Perspectives in probabilistic verification. *Proc. 2008 Second IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '08)*, pp 3–10. https://doi.org/10.1109/TASE.2008.44

[27] Gaboardi M (2019) Formal methods and proofs of privacy properties, part I. Accessed 22 May 2019, Presented at "Data Privacy: Foundations and Applications Boot Camp", Simons Institute for the Theory of Computing, University of California, Berkeley. Available at https://simons.berkeley.edu/talks/formal-methods-and-proofs-privacy-properties.

[28] Gaboardi M (2019) Formal methods and proofs of privacy properties, part II. Accessed 22 May 2019, Presented at "Data Privacy: Foundations and Applications Boot Camp", Simons Institute for the Theory of Computing, University of California, Berkeley. Available at https://simons.berkeley.edu/talks/formal-methods-and-proofs-privacy-properties-part-ii.

[29] Gaboardi M (2019) Formal methods and proofs of privacy properties, part III. Accessed 22 May 2019, Presented at "Data Privacy: Foundations and Applications Boot Camp", Simons Institute for the Theory of Computing, University of California, Berkeley. Available at https://simons.berkeley.edu/talks/formal-methods-and-proofs-privacy-properties-part-iii.

[30] Fenske E, Mani A, Johnson A, Sherr M (2017) Distributed measurement with private set-union cardinality. *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)* (ACM, New York, NY, USA), pp 2295–2312. https://doi.org/10.1145/3133956.3134034

[31] Rogers R, Roth A, Ullman J, Vadhan S (2016) Privacy odometers and filters: Pay-as-you-go composition. *Proc. 30th International Conference on Neural Information Processing Systems (NIPS'16)*, eds Lee DD, von Luxburg U, Garnett R, Sugiyama M, Guyon I (Curran Associates Inc., USA), pp 1929–1937. Available at https://arxiv.org/abs/1605.08294.

[32] Costea S, Ghinita G, Rughinis R, Tapus N (2015) Reduced relative errors for short sequence counting with differential privacy. *Proc. 20th International Conference on Control Systems and Science (CSCS 2015)*, eds Dumitrache I, Florea AM, Pop F, Dumitraşcu A (IEEE), pp 475–482. https://doi.org/10.1109/CSCS.2015.83

[33] Wang Y, Wang YX, Singh A (2015) Differentially private subspace clustering. *Proc. 28th International Conference on Neural Information Processing Systems (NIPS'15)* (Research Collection School Of Information Systems), pp 1000–1008. Available at https://ink.library.smu.edu.sg/sis_research/3469/.

[34] He X, Machanavajjhala A, Flynn C, Srivastava D (2017) Composing differential privacy and secure computation: A case study on scaling private record linkage. *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)* (ACM, New York, NY, USA), pp 1389–1406. https://doi.org/10.1145/3133956.3134030

[35] Zuliani P, Baier C, Clarke EM (2012) Rare-event verification for stochastic hybrid systems. *Proc. 15th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '12)* (ACM, New York, NY, USA), pp 217–226. https://doi.org/10.1145/2185632.2185665. Available at http://www.cs.cmu.edu/~emc/papers/ConferencePapers/Rare-EventVerificationforStochasticHybridSystems.pdf

[36] Lan G, Nemirovski A, Shapiro A (2012) Validation analysis of mirror descent stochastic approximation method. *Mathematical Programming* 134(2):425–458. https://doi.org/10.1007/s10107-011-0442-6

[37] Barthe G, Gaboardi M, Grégoire B, Hsu J, Strub PY (2016) Proving differential privacy via probabilistic couplings. *Proc. 31st Annual ACM-IEEE Symposium on Logic in Computer Science (LICS)*, pp 749–758.

[38] Barthe G, Fong N, Gaboardi M, Grégoire B, Hsu J, Strub PY (2016) Advanced probabilistic couplings for differential privacy. *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (ACM, New York, NY, USA), pp 55–67. https://doi.org/10.1145/2976749.2978391. Available at https://arxiv.org/abs/1606.07143

[39] Barthe G, Espitau T, Hsu J, Sato T, Strub PY (2017) *-Liftings for Differential Privacy. *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, eds Chatzigiannakis I, Indyk P, Kuhn F, Muscholl A (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany), *Leibniz International Proceedings in Informatics (LIPIcs)*, Vol. 80, pp 102:1–102:12. https://doi.org/10.4230/LIPIcs.ICALP.2017.102

[40] Castiglioni V, Chatzikokolakis K, Palamidessi C (2018) A logical characterization of differential privacy via behavioral metrics. *Proc. Formal Aspects of Component Software (FACS 2018)*, eds Bae K, Ölveczky PC (Springer), *Lecture Notes in Computer Science*, Vol. 11222, pp 75–96. https://doi.org/10.1007/978-3-030-02146-7_4. Available at https://hal.archives-ouvertes.fr/hal-01966870

[41] Wang Y, Ding Z, Wang G, Kifer D, Zhang D (2019) Proving differential privacy with shadow execution. Accessed 13 May 2019 Available at https://arxiv.org/pdf/1903.12254.pdf.

[42] Gaboardi M (2018) Formal verification of differential privacy. *Proc. 13th Workshop on Programming Languages and Analysis for Security (PLAS'18)*, ed Bailey MW (ACM), p 1. https://doi.org/10.1145/3264820.3264829. Invited Talk

[43] Zhang D, Kifer D (2017) Lightdp: Towards automating differential privacy proofs. *Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)* (ACM, New York, NY, USA), pp 888–901. https://doi.org/10.1145/3009837.3009884

[44] Boldo S, Joldes M, Muller JM, Popescu V (2017) Formal verification of a floating-point expansion renormalization algorithm. *Proc. 8th International Conference on Interactive Theorem Proving (ITP 2017)*, eds Ayala-Rincón M, Muñoz CA (Springer, Cham), *Lecture Notes in Computer Science*, Vol. 10499, pp 98–113. https://doi.org/10.1007/978-3-319-66107-0_7

[45] Hasan O, Tahar S (2009) Formal verification of tail distribution bounds in the HOL theorem prover. *Mathematical Methods in the Applied Sciences*, Vol. 32 No. 4, pp 480–504. https://doi.org/10.1002/mma.1055

[46] Rand R, Zdancewic S (2015) VPHL: A verified partial-correctness logic for probabilistic programs. *Electronic Notes in Theoretical Computer Science* 319:351–367. https://doi.org/10.1016/j.entcs.2015.12.021

[47] Křikava F, Vitek J (2018) Tests from traces: Automated unit test extraction for R. *Proc. 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)* (ACM, New York, NY, USA), pp 232–241. https://doi.org/10.1145/3213846.3213863

[48] Keeling KB, Pavur RJ (2011) Statistical accuracy of spreadsheet software. *The American Statistician* 65(4):265–273. https://doi.org/10.1198/tas.2011.09076

[49] (1999) Statistical reference datasets (StRD). https://doi.org/10.18434/T43G6C.

[50] Elbaum S, Rosenblum DS (2014) Known unknowns: Testing in the presence of uncertainty. *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, eds Cheung SC, Orso A, Storey MA (ACM, New York, NY, USA), pp 833–836. https://doi.org/10.1145/2635868.2666608

[51] Bassham LE III, Rukhin AL, Soto J, Nechvatal JR, Smid ME, Barker E, Leigh SD, Levenson M, Vangel M, Banks DL, Heckert NA, Dray J, Vo S (2010) A statistical test suite for random and pseudorandom number generators for cryptographic applications (NIST), Special Publication 800-22 revision 1a. https://doi.org/10.6028/NIST.SP.800-22r1a

[52] (2017) Improved version of the NIST statistical test suite (STS). Accessed 15 March 2019 Available at https://github.com/arcetri/sts.

[53] Walker J (2008) ent: A pseudorandom number sequence test program. Accessed 9 May 2019 Available at http://www.fourmilab.ch/random/.

[54] Brown RG, Eddelbuettel D, Bauer D (2019) Dieharder: A random number test suite. Accessed 9 May 2019 Available at http://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[55] Walfish M, Blumberg AJ (2015) Verifying computations without reexecuting them. *Commun ACM* 58(2):74–84. https://doi.org/10.1145/2641562

[56] Blum M, Kannan S (1995) Designing programs that check their work. *J ACM* 42(1):269–291. https://doi.org/10.1145/200836.200880

[57] Costea S, Tapus N (2015) Input validation for the laplace differential privacy mechanism. *Proc. 20th International Conference on Control Systems and Science (CSCS 2015)*, eds Dumitrache I, Florea AM, Pop F, Dumitraşcu A (IEEE), pp 469–474. https://doi.org/10.1109/CSCS.2015.84