INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Software Reliability via Run-Time Result-Checking*

Manuel Blum† Hal Wasserman‡

TR-94-053

October 1994

Abstract

We review the field of **result-checking,** discussing **simple checkers** and **self-correctors.** We argue that such checkers could profitably be incorporated in software as an aid to efficient debugging and reliable functionality. We consider how to modify traditional checking methodologies to make them more appropriate for use in real-time, real-number computer systems. In particular, we suggest that checkers should be allowed to use **stored randomness:** i.e., that they should be allowed to generate, pre-process, and store random bits prior to run-time, and then to use this information repeatedly in a series of run-time checks. In a case study of checking a general real-number linear transformation (for example, a Fourier Transform), we present a simple checker which uses stored randomness, and a self-corrector which is particularly efficient if stored randomness is allowed.

*A preliminary version of this paper appears as: "Program result-checking: a theory of testing meets a test of theory," *Proc. 35th IEEE FOCS,* 1994.

†Computer Science Division, University of California, Berkeley, CA 94720, blum@cs.berkeley.edu. This work was supported in part by NSF grant CCR92-01092 and in part by a MICRO grant from Hughes Aircraft Corporation and the State of California.

‡Computer Science Division, University of California, Berkeley, CA 94720, halw@cs.berkeley.edu. Supported by NDSEG Fellowship DAAH04-93-G-0267.

1 Result-checking and its applications

1.1 Assuring software reliability

Methodologies for assuring software reliability form an important part of the technology of programming. Yet the problem of efficiently identifying software bugs remains a difficult one, and one to which no perfect solution is likely to be found.

Software is often debugged via **testing suites:** one runs a program on a wide variety of carefully selected inputs, identifying a bug whenever the program fails to perform correctly. This approach leaves two important questions incompletely answered.

First, how do we *know* whether or not the program's performance is correct? Generally, some sort of an "oracle" is used here: our program's output may be compared to the output of an older, slower program believed to be more reliable, or the programmers may subject the output to a painstaking (and likely subjective and incomplete) examination by eye.

Second, given that a test suite feeds a program only selected inputs out of the often enormous space of all possibilities, how do we assure that every bug in the code will be evidenced? Indeed, special combinations of circumstances leading to a bug may well go untested. Furthermore, if the testing suite fails to accurately simulate the input distribution which the program will encounter in its lifetime, a supposedly debugged program may in fact fail quite frequently.

One alternative to testing is **program verification,** a methodology in which mathematical claims about the behavior of a program are stated and proved. Thus it is possible to demonstrate once and for all that the program *must* behave correctly on all possible inputs. Unfortunately, constructing such proofs for even simple programs has proved unexpectedly difficult. Moreover, many programmers would likely find it galling to have to formalize their expectations of program behavior into mathematical theorems.

Occasionally, the correctness of critical software is assured by having several groups of programmers create separate versions of the same program. At run-time, all of the versions are executed, and their outputs are compared. The gross inefficiency of this approach is evident, in terms of programming manpower as well as either increased run-time or additional parallel-hardware requirements. Moreover, the method fails if common misconceptions among the several development groups result in corresponding errors.

We find more convincing inspiration in the field of communications, where **error-detecting** and **error-correcting codes** allow for the identification of arbitrary run-time transmission errors. Such codes are highly efficient and are backed by strong mathematical guarantees of their reliability.

We would wish to provide such run-time error-identification in a more general computational context. This motivates us to review the field of **result-checking.**

1.2 Simple checkers

It is a matter of theoretical curiosity that, for certain computations, the time required to carry out the computation is asymptotically greater than the time required, given a tentative answer, to determine whether or not the answer is correct. For instance, consider the following factoring task: given as input a composite integer c, output any non-trivial factor d of c. Carrying out this computation is currently believed to be difficult, and yet, given **I/O pair** $\langle c, d \rangle$, it takes just one division to determine whether or not d is a correct output on input c.

These ideas have been formalized into the concept of a **simple checker** [12]. Let f be a function with smallest possible computation time $T(n)$ (or, if a strong lower bound cannot be determined, we informally set $T(n)$ equal to the smallest *known* computation time for f). Then a **simple checker for f** is an algorithm (generally randomized) with I/O specifications as follows:

- **Input:** I/O pair $\langle x, y \rangle$.
- **Correct output:** If $y = f(x)$, ACCEPT; otherwise, REJECT.
- **Reliability:** For all $\langle x, y \rangle$: on input $\langle x, y \rangle$ the checker must return correct output with probability (over internal randomization) $\geq p_c$, for p_c a constant close to 1.
- **"Little-o rule":** The checker is limited to time $o(T(n))$.

As an example, consider a sorting task: input $\vec{x}$ is an array of n integers; output $\vec{y}$ should be $\vec{x}$ sorted in increasing order. Completing this task—at least via a general comparison-based sort—is known to require time $\Omega(n \log n)$. Thus, if we limit our checker to time $O(n)$, this will suffice to satisfy the little-o rule.

Given $\langle \vec{x}, \vec{y} \rangle$, to check correctness we must first verify that the elements of $\vec{y}$ are in increasing order. This may easily be done in time $O(n)$. But we must also check that $\vec{y}$ is a permutation of $\vec{x}$. It might be convenient here to modify our sorter's I/O specifications, requiring that each element of $\vec{y}$ have attached a pointer to its original position in $\vec{x}$. Many sorting programs could easily be altered to maintain such pointers, and once they are available we can readily complete our check in time $O(n)$.

But what if $\vec{y}$ cannot be augmented with such pointers? Similarly, what if $\vec{x}$ and $\vec{y}$ are only available on-line from sequential storage, so that $O(1)$-time pointer-dereferencing is not possible? Then we may still employ a randomized method due to [39, 12].

We randomly select a deterministic hash-function h from a suitably defined set of possibilities, and we compare $h(x_1) + \cdots + h(x_n)$ with $h(y_1) + \cdots + h(y_n)$. If $\vec{y}$ is indeed a permutation of $\vec{x}$, the two values must be equal; conversely, it is readily proven that, if $\vec{y}$ is not a permutation of $\vec{x}$, the two values will differ with probability $\geq \frac{1}{2}$. Thus, if we ACCEPT only $\langle \vec{x}, \vec{y} \rangle$ pairs which pass t such trials, then our checker has probability of error $\leq \left(\frac{1}{2}\right)^t$, which may be made arbitrarily small.

1.3 Self-correctors

Again let f be a function with smallest (known) computation time $T(n)$. Let $\mathcal{D}$ be a well-defined probability distribution on inputs to f, and let $\mathbf{P}$ be a program such that, if x is chosen $\mathcal{D}$-randomly, $\mathbf{P}(x)$ equals $f(x)$ with probability of error limited to a small value p. In the current paper, $\mathcal{D}$ will always be the uniform-random distribution; so we are requiring simply that $\mathbf{P}$ computes f correctly on most inputs.

That $\mathbf{P}$ indeed has this property may be determined (with high probability) by testing $\mathbf{P}$ on $\gg \frac{1}{p}$ $\mathcal{D}$-random inputs and using some sort of an oracle—or a simple checker—to determine the correctness of each output. We envision this testing stage as being completed prior to run time. Another possibility is the use of a **self-tester** [14], which can give such assurances with less reliance on an outside oracle.

A **self-corrector for f** [14, 25] is then an algorithm (generally randomized) with I/O specifications as follows:

- **Input:** x (an input to f), along with $\mathbf{P}$, a program known to compute f with probability of error on $\mathcal{D}$-random inputs limited to a small value p. The corrector is allowed to call $\mathbf{P}$ repeatedly, using it like a subroutine.
- **Correct output:** $f(x)$.
- **Reliability:** For all $\langle x, \mathbf{P} \rangle$: on input $\langle x, \mathbf{P} \rangle$ the corrector must return correct output with probability (over internal randomization) $\geq p_c$, for p_c a constant close to 1.
- **Time-bound:** The corrector's time-bound, including subroutine calls to $\mathbf{P}$, must be limited to a constant multiple of $\mathbf{P}$'s time-bound; the corrector's time-bound, counting each subroutine call to $\mathbf{P}$ as just one step, must be $o(T(n))$.

As an example [14], consider a task of multiplication over a finite field: input is $w, x \in F$; output is product $wx \in F$. We assume addition over F to be a quicker and more reliable computation than multiplication.

Say we know that program $\mathbf{P}$ computes multiplication correctly for all but a $\leq \frac{1}{100}$ fraction of possible $\langle w, x \rangle$ inputs. (Note that this knowledge is in itself only a very weak assurance of the reliability of $\mathbf{P}$, as that $\leq \frac{1}{100}$ fraction of "difficult inputs" might well appear far more than $\frac{1}{100}$ of the time in the life of the program.) Then a self-corrector may be specified as follows: on input $\langle w, x \rangle$,

- Generate uniform-random $r_1, r_2 \in F$.
- Call $\mathbf{P}$ four times to calculate $\mathbf{P}(w - r_1, x - r_2)$, $\mathbf{P}(w - r_1, r_2)$, $\mathbf{P}(r_1, x - r_2)$, and $\mathbf{P}(r_1, r_2)$.
- Return, as our corrected value for wx,

$$y_c = \mathbf{P}(w - r_1, x - r_2) + \mathbf{P}(w - r_1, r_2) + \mathbf{P}(r_1, x - r_2) + \mathbf{P}(r_1, r_2).$$

Why does this work? Note that each of the four calls to $\mathbf{P}$ is on a pair of inputs uniform-randomly distributed over F, and so is likely to return a correct answer: probability of error is $\leq \frac{1}{100}$. Thus, all four return values are likely to be correct: probability of error is $\leq \frac{4}{100}$. And if all the return values are correct,

$$\begin{aligned} y_c &= \mathbf{P}(w - r_1, x - r_2) + \mathbf{P}(w - r_1, r_2) + \mathbf{P}(r_1, x - r_2) + \mathbf{P}(r_1, r_2) \\ &= (w - r_1)(x - r_2) + (w - r_1)r_2 + r_1(x - r_2) + r_1 r_2 \\ &= [(w - r_1) + r_1][(x - r_2) + r_2] \\ &= wx, \text{ as desired.} \end{aligned}$$

Note that corrected output y_c is superior to unmodified output $\mathbf{P}(w, x)$ in that there are no "difficult inputs": for *any* $\langle w, x \rangle$, each time we compute a corrected value y_c for wx, the chance of error (over choice of r_1, r_2) is $\leq \frac{4}{100}$. Moreover, observe that, if we compute several corrected outputs $y_c^1, \ldots, y_c^t$ (using new random values of r_1, r_2 for each y_c^i) and we pick the majority answer as our final output, we can thereby make the chance of error arbitrarily small.

One may also define a **complex checker** [12], which outputs an ACCEPT/REJECT correctness-check similar to that of a simple checker, but which, like a self-corrector, is allowed to poll $\mathbf{P}$ at several locations. For more information on checking, refer to the annotated bibliography.

1.4 Debugging via checkers

While checking emerges naturally from the theory of computation, the software applications of the field are clear. We contend that the incorporation of checkers into a program's code should be a substantial aid in debugging the program and assuring its reliability. Checkers may thus provide the basis for a debugging methodology more rigorous than the testing suite and more pragmatic than verification.

Throughout the process of software testing, checkers may be used to identify incorrect outputs. They will thereby serve as an efficient alternative to a conventional testing oracle.

Even after software is put into use, leaving the checkers in the code will allow for the effective detection of lingering bugs. We envision that, each time a checker finds a bug, an informative output will be written to a file; this file will then be periodically reviewed by software-maintenance engineers. Another possibility is an immediate warning message, so that the user of a critical system will not be duped into accepting erroneous output.

While it could be argued that buggy output would be noticed by the user in any case, an automatic system for identifying errors should in fact reduce the probability that bugs will be

ignored or forgotten. Moreover, a checker can catch an error in a program component whose effects may not be readily apparent to the user. Thus, a checker might well identify a bug in a critical system before it goes on to cause a catastrophic failure. The utility of self-correctors in critical systems is even more evident.

By writing a checker, one group of programmers may assure themselves that another group's code is not undermining theirs by passing along erroneous output. Similarly, the software engineer who creates the specifications for a program component can write a checker to verify that the programmers have truly understood and provided the functionality he required. Since checkers depend only on the I/O specifications of a computational task, one may check even a program component whose internals are not available for examination, such as a library of utilities running on a remote machine. Thus checkers facilitate the discovery of those misunderstandings at the "seams" of programs which so often underlie a software failure. Evidently, all of this applies well to object-oriented paradigms.

Unlike verification, checking reveals incorrect output originating in any cause—whether due to software bugs, hardware bugs, or flaky run-time errors. Moreover, since a checker concerns itself only with the I/O specifications of a computational task, the introduction of a potentially unreliable new algorithm for solving an old task may not require any change to the associated checker. And so, as a program is modified throughout its lifetime—often without an adequate repeat of the testing process—its checkers will continue to guard against errors.

1.5 Checker-based debugging concerns

• **Buggy checkers:** It may be objected that the use of a checker begs the question of debugging, for checking may fail due to the checker's code being as buggy as the program's. To this objection we can respond with two heuristic arguments.

First, checkers can often be simpler than the programs they check, and so, presumably, less likely to have bugs. For example, some computers now use intricate, arithmetically unstable algorithms to compute matrix multiplication in time, say, $O(n^{2.4})$, rather than the standard $O(n^3)$. And yet an $O(n^2)$-time checker for matrix multiplication [20] uses only the simplest of multiplication algorithms, and so is seemingly likely to be bug-free.

Second, observe that one of the following four conditions must hold each time a (possibly buggy) simple checker $\mathbf{C}$ attempts to check the output of a (possibly buggy) program $\mathbf{P}$:

(I). $\mathbf{P}(x)$ is correct; $\mathbf{C}$ correctly accepts it.

(II). $\mathbf{P}(x)$ is incorrect; $\mathbf{C}$ correctly rejects it.

(III). "False alarm": $\mathbf{P}(x)$ is correct; $\mathbf{C}$ incorrectly rejects it.

(IV). "Simultaneous error": $\mathbf{P}(x)$ is incorrect; $\mathbf{C}$ incorrectly accepts it.

Note that only "simultaneous error" (IV) is a truly bad outcome: for a "false alarm" (III) at least draws our attention to a bug needing to be fixed in $\mathbf{C}$.

Thus, what we must avoid is a strong correlation between x values at which $\mathbf{P}$ fails and x values at which $\mathbf{C}(x, \mathbf{P}(x))$ fails. And it is our heuristic contention that such a correlation is unlikely. For note that one effect of the "little-o rule" in our definition of a simple checker is that $\mathbf{C}$ doesn't have sufficient time to merely reduplicate the computation done by $\mathbf{P}$. In other words, $\mathbf{C}$ must be doing something *essentially different* from what $\mathbf{P}$ does, and so may reasonably be expected to make *different errors* than $\mathbf{P}$.

We can easily assure via testing that $\mathbf{P}$ and $\mathbf{C}$ each have only a small probability p of error. Thus, if the two sets of error positions indeed have little correlation, we would hope for the probability of simultaneous error to be little more than p^2. Moreover, since bugs are thus likely to produce far more non-simultaneous errors than simultaneous errors, a given bug will probably be identified and fixed before it can generate any simultaneous errors at all.

• **False alarms:** We have claimed above that a "false alarm" (III) is not a truly bad outcome. Nevertheless, programmers have communicated to us that, while a checker which doesn't catch every bug may still be useful, a checker which continually gives false alarms is worse than useless. Note that such false alarms might be generated either by bugs in the checker's code or by the small probability of failure of a randomized checker.

To this objection we can respond again that checkers are likely to be simpler than the programs they check and so less buggy, and that, as will be seen in Section 2, the small probability of failure of a randomized checker may readily be made very small indeed. Thus we would expect condition (III) to be far less frequent than condition (II).

• **Real-number issues** (see [21, 2]): Traditional checkers, such as our example self-corrector in Section 1.3, often rely on the orderly properties of finite fields. In many programming situations, we are more likely to encounter real numbers—or, actually, approximate reals represented by a fixed number of bits.

Such numbers will be limited to a legal subrange of $\Re$. In Section 2.3, we will see how to modify a traditional self-correcting methodology to account for this. Moreover, limited precision will likely result in round-off errors within $\mathbf{P}$. We must then determine how much arithmetic error may allowably occur within $\mathbf{P}$, and must check correctness only up to the limit of this error-delta. In Section 2, we will see that, while certain checking algorithms do not perform as well as we might wish on a limited-precision computer, limited precision is by no means necessarily fatal to our checking project.

• **Performance loss:** Simple checkers are inherently efficient: due to their little-o rule, they should not significantly slow the programs they check. Self-correctors, on the other hand, require multiple calls to $\mathbf{P}$, and so multiply execution-time by a small constant (or, alternatively, require additional parallel hardware). It is our assumption here that, in a real-time computer system, such a slowdown is not acceptable. Thus, in Section 1.6 we will consider a change to our definitions which will allow for self-correcting with the same real-time properties as simple checking.

1.6 Stored randomness

We here suggest an extension to the standard definition of a checker: that randomized checkers, rather than having to generate fresh random bits for each check, should be allowed to use **stored randomness.** That is, prior to run-time—say while the program is idle, or during boot-up—we generate random bits and do some (perhaps lengthy) pre-processing; we then use this stored information repeatedly in a series of run-time checks.

This method allows for the use of checking algorithms which would otherwise require too much computation at run-time. In Section 2, we will present a checking algorithm which seemingly is possible only if one allows stored randomness, and a self-corrector which, if one allows stored randomness, can run at real-time speeds.

We believe this weakening of our checker definitions to be reasonable—at least in real-world software-debugging situations. For software could well be expected to have a pre–run-time period during which time is at far less of a premium. And, while repeatedly using "stale" randomness does involve some inevitable compromise of our checkers' reliability, several results in Section 2 will indicate that our checkers still exhibit satisfactory performance.

1.7 Variations on the checking paradigm

It may still be objected that many programming tasks are less amenable to checking than are the clean mathematical problems with which theoreticians usually deal. Nevertheless, it is our experience that almost any computation may be subjected to interesting checks. The following is a list of checking ideas which may suggest to the reader how definitions can be loosened up to make checking more suitable as a tool for software debugging.

• **Partial checks:** One may find it sufficient to check only certain aspects of a computational output. Programmers may focus on properties they think particularly likely to reveal bugs—or particularly critical to the success of the system. Certain checkers might only be capable of identifying outputs incorrect by a very large error-delta. Even just checking inputs and outputs separately to verify that they fall in a legal range may prove useful.

• **Timing checks:** We can easily do a run-time check of the execution-time of a software component; an unexpectedly large (or small) time could then point us to a bug. For instance, a programmer's expectation that a particular subroutine should take time $\approx 10n^2$ on input of variable length n could be checked against actual performance.

• **Checking via interactive proofs:** If there exists an **interactive proof** of correctness for a given computation, this proof may be extendable to a complex checker. For more on interactive proofs, refer to the bibliography.

• **Changing I/O specifications:** In Section 1.2, we saw how an easy augmentation to the output of a sorting program might make the program easier to check. Similarly, consider the problem of checking an $\Omega(\log n)$-time binary-search task. As traditionally stated (see [10, p. 35], where it features in a relevant discussion of the difficulty of writing correct algorithms), binary search has as input a key k and a numerical array $a[1], \ldots, a[n]$, $a[1] \leq \cdots \leq a[n]$, and as output i such that $a[i] = k$, or 0 if k is not in the array.

Note that the problem as stated is uncheckable: for, if the output is 0, it will take $O(\log n)$ steps to confirm that k is indeed not in the array. But say that we modify the output specification of our binary-search task to read: output i such that $a[i] = k$, or, if k is not in the array, $\langle j, j+1 \rangle$ such that $a[j] < k < a[j+1]$. Any natural binary-search program can easily be modified to give such output, and completing an $O(1)$-time check is then straightforward.

• **Weak or occasional checks [11]:** Certain pseudo-checkers have only a small probability of noticing a bug. For example, if a search program, given key k and array $a[1], \ldots, a[n]$, claims that k does not occur in a, we could check this by selecting an element of a at random and verifying that it is not equal to k. This weak checker has probability $\frac{1}{n}$ of identifying an incorrect claim.

Alternatively, a lengthy check might be employed, to save time, only on occasional I/O pairs. Given that some bugs cause quite frequent errors over the lifetime of a program, such weak or occasional checkers may well be of use.

• **Batch checks [30]:** For certain computational tasks, if one stores up a number of I/O pairs, one may check them all at once more quickly than one could have checked them separately. While it may be too late to correct output after completing such a retroactive check, this method can when applicable provide a very time-efficient identification of bugs.

• **Inefficient auto-correcting:** When a checker identifies an error, we can simulate self-correcting by, say, loading and running an older, slower, but more reliable version of the program. This should hopefully be necessary only on rare occasions, so the overall loss of speed may not be unreasonable.

2 Case study: checking real-number linear transformations

We will now consider the problem of checking and self-correcting a general linear transformation, defined as follows: input is n-length vector $\vec{x}$ of real numbers, output n-length vector $\vec{y} = \vec{x}A$, where A is a fixed $n \times n$ matrix of real coefficients. Extensions of our results to more general cases—e.g., $\vec{y}$ of different length than $\vec{x}$, or components complex rather than real—are straightforward.

Evidently, any such transformation may be computed in time $O(n^2)$. Depending on the nature of A, the least possible time to compute the transformation may range from $\theta(n)$ to $\theta(n^2)$. Our simple checker will take time $O(n)$ (as will our self-corrector, not counting one or more calls to $\mathbf{P}$), and

so the little-o rule is satisfied for all but the $\theta(n)$-time transformations.[1] For example, the Fourier Transform is linear and may be computed in time $\theta(n \log n)$ via the Fast Fourier Transform; if one makes the reasonable assumption that an $O(n)$-time Fourier Transform is not possible, then our algorithms qualify as checkers for this Transform.

Assume now that program **P** is intended to carry out the $\vec{x} \mapsto \vec{x}A$ transformation on a computer with a limited-accuracy, fixed-point representation of real numbers. **P**'s I/O specifications could then reasonably be formulated as follows:

- **Input:** $\vec{x} = (x_1, \ldots, x_n)$, where each x_i is a fixed-point real number and so is limited to a legal subrange of $\Re$: say, to range $[-1, 1]$.
- **Output:** $\mathbf{P}(\vec{x}) = \vec{y} = (y_1, \ldots, y_n)$, where each y_i is a fixed-point real.
- Let $\delta \in \Re^+$ be a constant; let $\vec{\Delta} = (\Delta_1, \ldots, \Delta_n)$ be the "error vector" $\vec{y} - \vec{x}A$. Then:
  - I/O pair $\langle \vec{x}, \vec{y} \rangle$ is **definitely correct** iff each $|\Delta_i| \leq \delta$.
  - I/O pair $\langle \vec{x}, \vec{y} \rangle$ is **definitely incorrect** iff there exists i such that $|\Delta_i| \geq 6\sqrt{n}\,\delta$.

Note that, due to arithmetic round-off errors within **P**, a small error-delta is to be regarded as acceptable by our checkers: specifically, we model **P**'s acceptable error as a flat error of at most $\pm\delta$ in each component of $\vec{y}$. Also note that we will only require our checkers to accept I/O which is *definitely correct* and to reject I/O which is *definitely incorrect.* It is seemingly unavoidable that there is an intermediate region of I/O pairs $\langle \vec{x}, \vec{y} \rangle$ which could allowably be either accepted or rejected.

2.1 A simple checker

Motivation: To develop a simple checker for **P**, we will take the approach of generating a randomized vector $\vec{r}$ and trying to determine whether or not $\vec{y} \approx \vec{x}A$ by calculating whether or not $\vec{y} \cdot \vec{r} \approx (\vec{x}A) \cdot \vec{r}$. This method is a variant of that in [2, Section 4.1.1].

To facilitate the calculation of $(\vec{x}A) \cdot \vec{r}$, we will employ the method of *stored randomness:* by generating $\vec{r}$ and pre-processing it together with A prior to run-time, we are then able to complete the calculation of $(\vec{x}A) \cdot \vec{r}$ with just $O(n)$ run-time arithmetic operations. Thus we achieve the strong result of checking a $\theta(n^2)$-time computation in time just $O(n)$.

Algorithm: Our checker's pre-processing stage, to be completed prior to run-time, is specified as follows:

- For $k := 1$ to 10:
  - Generate and store $\vec{r}^k$, an n-length vector of form $(\pm 1, \pm 1, \ldots, \pm 1)$, where each $\pm$ is chosen positive or negative with independent 50/50 probability.
  - Calculate and store $\vec{v}^k = \vec{r}^k A^T$ (where A^T denotes the transpose of A).

Then, to check an I/O pair $\langle \vec{x}, \vec{y} \rangle$ at run-time, we employ this $O(n)$-time check:

- For $k := 1$ to 10:
  - Calculate $D^k = \vec{y} \cdot \vec{r}^k - \vec{x} \cdot \vec{v}^k$.
  - If $|D^k| \geq 6\sqrt{n}\,\delta$, return REJECT.
- If all 10 tests are passed, return ACCEPT.

[1] In the more general case that $\vec{y}$ has length m, any non-trivial linear transformation would have least possible computation-time on range $\theta(\max\{m, n\})$ to $\theta(mn)$. It is easily verified that our simple checker and our self-corrector each take time $O(\max\{m, n\})$; and so, once again, the little-o rule is satisfied for all but the fastest of transformations.

Why does this work? First note that

$$\begin{aligned} D^k &= \vec{y} \cdot \vec{r}^k - \vec{x} \cdot \vec{v}^k \\ &= \vec{y} \cdot \vec{r}^k - \vec{x} \cdot (\vec{r}^k A^T) \\ &= \vec{y} \cdot \vec{r}^k - (\vec{x}A) \cdot \vec{r}^k \\ &= (\vec{y} - \vec{x}A) \cdot \vec{r}^k \\ &= \vec{\Delta} \cdot \vec{r}^k \\ &= \pm\Delta_1 \pm \cdots \pm \Delta_n \end{aligned}$$

where each $\pm$ is chosen positive or negative with independent 50/50 probability. Note that we have employed a linear-algebra identity: the dot-product of two vectors may equivalently be thought of as their matrix-product once the second vector (i.e., row-matrix) has been transposed; thus $\vec{x} \cdot (\vec{r}^k A^T) = \vec{x}\,(\vec{r}^k A^T)^T = \vec{x}(A\,[\vec{r}^k]^T) = (\vec{x}A)\,[\vec{r}^k]^T = (\vec{x}A) \cdot \vec{r}^k$.

Lemma 1 *For any definitely correct* $\langle \vec{x}, \vec{y} \rangle$,

$$\Pr_{\vec{r}^k}\left[|D^k| \geq 6\sqrt{n}\,\delta\right] \leq \frac{1}{10,000,000}.$$

Proof: Each $|\Delta_i| \leq \delta$. Thus, by a Chernoff Bound [1, Theorem A.16],

$$\begin{aligned} \Pr\left[|\pm\Delta_1 \pm \cdots \pm \Delta_n| \geq 6\sqrt{n}\,\delta\right] &\leq 2e^{-(6\sqrt{n})^2/2n} \\ &< \frac{1}{10,000,000}. \end{aligned}$$

■

Lemma 2 *For any definitely incorrect* $\langle \vec{x}, \vec{y} \rangle$,

$$\Pr_{\vec{r}^k}\left[|D^k| < 6\sqrt{n}\,\delta\right] \leq \frac{1}{2}.$$

Proof: We know there exists i such that $|\Delta_i| \geq 6\sqrt{n}\,\delta$. Fix all components of $\vec{r}^k$ except for r_i^k. Then observe that, for at least one of the two possible values $\{\pm 1\}$ of r_i^k, $|D^k| = |\Delta_1 r_1^k + \cdots + \Delta_n r_n^k|$ must be $\geq |\Delta_i| \geq 6\sqrt{n}\,\delta$.

Thus $|D^k| < 6\sqrt{n}\,\delta$ for at most $\frac{1}{2}$ of the equally likely choices of random vector $\vec{r}^k$. ■

Lemma 3 (I). *Given any definitely correct* $\langle \vec{x}, \vec{y} \rangle$,

$$\Pr_{\vec{r}^1,\ldots,\vec{r}^{10}}\left[\textit{checker mistakenly rejects}\right] \leq \frac{1}{1,000,000}.$$

(II). *Given any definitely incorrect* $\langle \vec{x}, \vec{y} \rangle$,

$$\Pr_{\vec{r}^1,\ldots,\vec{r}^{10}}\left[\textit{checker mistakenly accepts}\right] \leq \frac{1}{1,000}.$$

Proof: (I). By Lemma 1, the probability of a mistaken reject at each $\vec{r}^k$ is $\leq \frac{1}{10,000,000}$. Thus, the probability of a mistaken reject on *any* of the 10 tries is $\leq 10 \cdot \frac{1}{10,000,000} = \frac{1}{1,000,000}$.

(II). By Lemma 2, the probability of failing to reject at each $\vec{r}^k$ is $\leq \frac{1}{2}$. Thus, the probabililty of rejecting on *none* of the 10 independent tries is $\leq \left(\frac{1}{2}\right)^{10} < \frac{1}{1,000}$. ■

While Lemma 3 embodies our intuition of the checker's correctness, it does not deal with the implications of repeatedly using the same stored random vectors $\vec{r}^1, \ldots, \vec{r}^{10}$. The following claims suggest that our use of *stored randomness* does not seriously undermine the reliability of our checker:

Lemma 4 *Say we select random $\vec{r}^1, \ldots, \vec{r}^{10}$ and use these stored vectors to check ℓ runs of* **P**. *Then, for probabilities over the choice of $\vec{r}^1, \ldots, \vec{r}^{10}$:*

(I). *Say that a bug causes at least one definitely incorrect I/O pair. Then the probability that our checker will miss this bug is $\leq \frac{1}{1{,}000}$.*

(II). *If $\ell \leq 10{,}000$, then the probability that* any *definitely correct I/O pair will be mistakenly rejected is $\leq \frac{1}{100}$.*

(III). *The probability that $\geq \frac{1}{2}$ of all definitely incorrect I/O pairs will be mistakenly accepted is $\leq \frac{1}{500}$.*

(IV). *The probability that $\geq \frac{1}{10{,}000}$ of all definitely correct I/O pairs will be mistakenly rejected is $\leq \frac{1}{100}$.*

Proof: (I) and (II) follow readily from Lemma 3. To prove (III), note a consequence of Lemma 3 (II): if we select $\vec{r}^1, \ldots, \vec{r}^{10}$ randomly and $\langle \vec{x}, \vec{y} \rangle$ uniform-randomly from the list of all *definitely incorrect* I/O pairs out of our ℓ runs, then the probability that our checker accepts on input $\langle \vec{x}, \vec{y} \rangle$ and randomization $\vec{r}^1, \ldots, \vec{r}^{10}$ is $\leq \frac{1}{1{,}000}$. And, if (III) were false, then this same probability would, on the contrary, be $> \frac{1}{500} \cdot \frac{1}{2} = \frac{1}{1{,}000}$. (IV) is proved analogously. ■

2.2 Simple checker—variants

• A fuller statement of our algorithm would include consideration of arithmetic round-off error in the checker itself as well as in **P**. However, under the reasonable assumption (for a fixed-point system) that addition and negation do not generate round-off errors, our checker could only generate error in the n real-number multiplications needed to calculate $\vec{x} \cdot \vec{v}^k$. And it is our heuristic expectation that such error would not be large enough to significantly affect the validity of our checker.

• Evidently, different values could be used for the constants in our algorithm. Our goal was to particularize a pragmatic checker—one which satisfies reasonably well each of the following desirable conditions: small gap between the definitions of *definitely correct* and *definitely incorrect;* small chance of a mistaken ACCEPT; very small chance of a mistaken REJECT; and small run-time.

To generalize: let our definition of *definitely incorrect* be that there exists $|\Delta_i| \geq c\sqrt{n}\,\delta$. Let our algorithm calculate t values of D^k and reject iff any $|D^k| \geq c\sqrt{n}\,\delta$. (We used $c = 6$, $t = 10$ above.) Then the error bound in Lemma 1 is $2e^{-c^2/2}$. The error bound in Lemma 3 (I) is $2te^{-c^2/2}$, and that in Lemma 3 (II) is $\left(\frac{1}{2}\right)^t$. Lemma 4 (III) generalizes: for any $q \in [0, 1]$, the probability that an at least q portion of all *definitely incorrect* I/O pairs will be mistakenly accepted is $\leq \frac{\rho}{q}$, where ρ is the error bound from Lemma 3 (II). Lemma 4 (IV) generalizes analogously.

2.3 A self-corrector

Motivation: To develop a self-corrector for **P**, we start with the traditional approach of adding a random displacement to the locations at which we must poll **P**. However, to prevent our inputs to **P** from going out of legal range, we here find it necessary to poll **P** at a location which is a weighted sum of desired input $\vec{x}$ and a random input $\vec{r}$. If we weight the sum so that $\vec{r}$ dominates, the resulting sum-vector is near-uniformly distributed, allowing us to prove the reliability of our self-corrector. This method may be compared to those employed in [21].

Algorithm: Assume that we have tested **P** on a large number of **random input vectors:** i.e., vectors generated by choosing each component uniform-randomly from the set of fixed-point real numbers on legal range $[-1, 1]$. For each input $\vec{x}$, we verify that $\mathbf{P}(\vec{x})$ is correct ($\pm$ an allowable error-delta); a version of our simple checker might be employed to facilitate this verification. Through such a testing stage (or by use of a self-tester [18]), we can satisfy ourselves that (with high probability) the fraction of inputs on which **P** returns incorrect output is very small: say, $\leq \frac{1}{10{,}000{,}000}$.

Once we have this assurance, we can employ the following self-corrector for $\mathbf{P}$, whose time-bound is two calls to $\mathbf{P}$ plus $O(n)$: on input $\vec{x}$,

- Generate random input vector $\vec{r}$.
- Call $\mathbf{P}$ to calculate $\mathbf{P}(\vec{r})$.
- Call $\mathbf{P}$ to calculate $\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)$. (Note that this is possible because, for any legal input vectors $\vec{x}$ and $\vec{r}$, $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$ must also be in legal range.)
- Return, as our corrected value for $\mathbf{P}(\vec{x})$,

$$\vec{y}_c = n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot\mathbf{P}(\vec{r}).$$

Lemma 5 *For any $\vec{x}$:*

$$\mathrm{Pr}_{\vec{r}}\left[\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) \text{ is incorrect}\right] \leq \frac{3}{10,000,000}.$$

Proof: Fix $\vec{x}$. As $\vec{r}$ varies over all legal input vectors, each component of vector $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$ varies over a $\left(1-\frac{1}{n}\right)$ fraction of legal range $[-1,1]$. Thus, since $\vec{r}$ is a random input vector, the value of $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$ is distributed uniform-randomly[2] over a "neighborhood" of input vectors whose size, as a fraction of the space of all legal input vectors, is $\left(1-\frac{1}{n}\right)^n \approx \frac{1}{e} > \frac{1}{3}$.

So the probability of hitting an incorrect output at $\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)$ is at most three times the probability of hitting an incorrect output at a truly uniform-random location, which we know from testing to be $\leq \frac{1}{10,000,000}$. ■

Lemma 6 *For any $\vec{x}$: $\vec{y}_c$ will, with high probability, (approximately) equal desired output $\vec{x}A$: chance of error (over choice of $\vec{r}$) is $\leq \frac{4}{10,000,000}$.*

Proof: Based on testing and on Lemma 5, we know that $\mathbf{P}(\vec{r})$ and $\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)$ are both likely to be correct: chance of error is $\leq \frac{1}{10,000,000} + \frac{3}{10,000,000} = \frac{4}{10,000,000}$. And if they are indeed both correct, then

$$\begin{aligned}
\vec{y}_c &= n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot\mathbf{P}(\vec{r}) \\
&= n\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)A - (n-1)\vec{r}A \\
&= n\left(\frac{1}{n}\right)\vec{x}A + n\left(1-\frac{1}{n}\right)\vec{r}A - (n-1)\vec{r}A \\
&= \vec{x}A.
\end{aligned}$$

■

Unfortunately, our self-corrector reduces the arithmetic precision of $\mathbf{P}$. For, in calculating $\vec{y}_c = n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot\mathbf{P}(\vec{r})$, significant digits are lost when we divide $\vec{x}$ by n; moreover, the final multiplications by n and $(n-1)$ will magnify the $\pm\delta$ errors in the values returned from $\mathbf{P}$. For this reason, our self-corrector works best on a program $\mathbf{P}$ which provides several more digits of arithmetic accuracy than are seemingly warranted by the accuracy of its inputs. In particular, it seems that about $\log n$ additional digits would be sufficient.

[2] If arithmetic rounding is not done with care to assure this uniformity, the error bound in Lemma 5 rises to $\frac{6}{10,000,000}$.

2.4 A faster self-corrector

Another problem with our self-corrector is that it slows execution speed by a factor of at least 2. We can fix this problem by employing the method of *stored randomness:*

Prior to run-time, we generate and store a random input vector $\vec{r}$. We also calculate and store $(n-1)\mathbf{P}(\vec{r})$. By repeatedly using these stored values, we can now calculate

$$\vec{y}_c = n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot\mathbf{P}(\vec{r})$$

with only one call to $\mathbf{P}$. Thus, our modified self-corrector—like our simple checker—only adds $O(n)$ arithmetic operations to the execution-time of $\mathbf{P}$. Assuming that $\mathbf{P}$ is an $\omega(n)$-time program, the resulting loss of performance should be negligible.

Proceeding as in Lemma 4, we can easily derive the following results—results which suggest that the use of stored randomness does not seriously undermine the reliability of our checker:

Lemma 7 *Say we select random $\vec{r}$ and use this stored vector to self-correct ℓ runs of $\mathbf{P}$. Then, for probabilities over the choice of $\vec{r}$:*

(I). *If $\ell \leq 10{,}000$, then the probability that* any *corrected output will be erroneous is $\leq \frac{4}{1{,}000}$.*

(II). *The probability that $\geq \frac{1}{10{,}000}$ of the corrected outputs will be erroneous is $\leq \frac{4}{1{,}000}$.*

2.5 Self-corrector—variants

• To generalize: let p be an upper bound on the portion of input vectors $\vec{x}$ for which $\mathbf{P}(\vec{x})$ is incorrect. (We used $p = \frac{1}{10{,}000{,}000}$ above.) Then the error bound in Lemma 5 is ep, and that in Lemma 6 is $(e+1)p$. Lemma 7 (II) generalizes: for any $q \in [0,1]$, the probability that an at least q portion of the corrected outputs will be erroneous is $\leq \frac{(e+1)p}{q}$.

• Having calculated corrected output $\vec{y}_c$ as described above, we could then employ a version of our simple checker to double-check the correctness of $\vec{y}_c$. If we signal a bug whenever the simple checker finds $\vec{y}_c$ to be incorrect, we then provide both checking and self-correcting for $\mathbf{P}$, while still adding only $O(n)$ arithmetic operations to $\mathbf{P}$'s execution-time.

• The checking situation would be somewhat different if our computer used floating-point, rather than fixed-point, representations. Our model of $\mathbf{P}$'s arithmetic errors as a flat $\pm\delta$ in each component of output would need modification, and the problem of designing expressions such as $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$—i.e., randomized expressions meant to be near-uniformly distributed over the set of legal inputs—would be more complex.

References

[1] N. Alon, J. H. Spencer, and P. Erdős, *The Probabilistic Method,* John Wiley & Sons, 1992.

[2] S. Ar, M. Blum, B. Codenotti, and P. Gemmell, "Checking approximate computations over the reals," *Proc. 25th ACM STOC,* pp. 786–795, 1993. Extends checking methodologies to computations on limited-accuracy real numbers. Examples: matrix multiplication, inversion, and determinant; solving systems of linear equations.

[3] S. Arora and S. Safra, "Probabilistic checking of proofs; a new characterization of NP," *Proc. 33rd IEEE FOCS,* pp. 2–13, 1992. Equates NP languages with those in interactive-proof class $\mathrm{PCP}(\log n, \sqrt{\log n})$.

[4] L. Babai, L. Fortnow, and C. Lund, "Non-deterministic exponential time has two-prover interactive protocols," *Computational Complexity,* Vol. 1, pp. 3–40, 1991. Proves that NEXP = 2IP, and hence that NEXP-complete problems have complex checkers.

[5] L. Babai and L. Fortnow, "Arithmetization: a new method in structural complexity theory," *Computational Complexity,* Vol. 1, pp. 41–46, 1991. Preliminary version: "A characterization of #P by arithmetic straight line programs," *Proc. 31st IEEE FOCS,* pp. 26–34, 1990. Further develops a technique from [4] of translating Boolean formulae into multivariate polynomials.

[6] L. Babai, L. Fortnow, L. Levin, and M. Szegedy, "Checking computations in polylogarithmic time," *Proc. 23rd ACM STOC,* pp. 21–31, 1991. A variant of [4] with lower time-bounds, this paper introduces an unusual sort of very fast checker for NP computations. Such checkers could be regarded as "hardware checkers," in that they ensure that the hardware follows instructions correctly, but don't ensure that the software is correct.

[7] D. Beaver and J. Feigenbaum, "Hiding instances in multioracle queries," *Proc. 7th Annual Symposium of Theoretical Aspects of Computer Science,* pp. 37–48, 1990.

[8] R. Beigel and J. Feigenbaum, "On the complexity of coherent sets," *AT&T Technical Report,* 1990.

[9] R. Beigel and J. Feigenbaum, "On being incoherent without being very hard," *Computational Complexity,* Vol. 2, pp. 1–17, 1992. Response to questions of [13, 40], including a proof that all NP-complete languages are coherent.

[10] J. Bentley, *Programming Pearls,* Addison-Wesley, 1986.

[11] Avrim Blum, personal communication.

[12] M. Blum, "Designing programs to check their work," *ICSI Technical Report TR-88-009,* 1988. Introduces result-checking.

[13] M. Blum and S. Kannan, "Designing programs that check their work," *Proc. 21st ACM STOC,* pp. 86–97, 1989.

[14] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems," *Journal of Computer & System Sciences,* Vol. 47, pp. 549–95, 1993. Preliminary version: *Proc. 22nd ACM STOC,* pp. 73–83, 1990. Introduces self-testers and self-correctors.

[15] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," *Algorithmica,* Vol. 12, pp. 225–244, 1994. Demonstrates that, given a small, secure data base, one may check the correctness of a large, adversarial data base.

[16] R. Butler and G. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Transactions on Software Engineering,* Vol. 19, pp. 3–12, 1993. Proves inherent limitations of conventional software testing and of the attempt to assure reliability by running several versions of a program.

[17] R. Cleve and M. Luby, "A note on self-testing/correcting methods for trigonometric functions," *ICSI Technical Report 90-032,* 1990.

[18] Funda Ergün, "Testing multivariate linear functions: overcoming the generator bottleneck," manuscript, Department of Computer Science, Cornell University, 1994. Specifies self-testers for functions including the Fourier Transform.

[19] L. Fortnow and M. Sipser, "Are there interactive protocols for co-NP languages?" *Information Processing Letters,* Vol. 28, pp. 249–251, 1988. Suggests that co-NP may not be contained in IP. [35] later proved the contrary.

[20] R. Freivalds, "Fast probabilistic algorithms," *Springer Verlag Lecture Notes in Computer Science #74: Mathematical Foundations of Computer Science,* pp. 57–69, 1979. Early instances of result-checking, including a simple checker for matrix multiplication.

[21] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson, "Self-testing/correcting for polynomials and for approximate functions," *Proc. 23rd ACM STOC,* pp. 32–42, 1991. Commences the extension of traditional checking methodologies to computations on limited-accuracy reals.

[22] P. Gemmell and M. Sudan, "Highly resilient correctors for polynomials," *Information Processing Letters,* Vol. 43, pp. 169–174, 1992. Gives near-optimal self-correctors for programs supposed to compute multivariate polynomials. As long as a program is correct on a $\frac{1}{2}+\delta$ fraction of inputs (for $\delta \in \Re^+$), self-correcting is possible.

[23] S. Kannan, *Program Result-Checking with Applications,* Ph.D. thesis, Department of Computer Science, University of California, Berkeley, 1990. Includes checkers for several hard group-theoretic problems.

[24] R. Lipton, "Efficient checking of computations," *Proc. 7th Annual Symposium of Theoretical Aspects of Computer Science,* pp. 207-215, 1990.

[25] R. Lipton, "New directions in testing, distributed computing and cryptography," *DIMACS Series on Discrete Mathematics and Theoretical Computer Science,* Vol. 2, pp. 191–202, 1991. Proves that #P-complete problems have checker/correctors.

[26] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," *Proc. 31st IEEE FOCS,* pp. 2–10, 1990.

[27] S. Micali, "Computer science proofs and error-detecting computation," *MIT Lab for Computer Science Technical Report,* 1992, and "Computer science proofs," *MIT Lab for Computer Science Technical Report TM-510,* 1994. Gives result-checkers for NP-complete problems, subject to the assumptions that we have available a random oracle which can serve as a cryptographically-secure hash-function, and that the program being checked has insufficient time to find collisions in this hash-function.

[28] N. Nisan, "Co-SAT has multi-prover interactive proofs," e-mail message, 1989. Initiated events leading to [35, 4].

[29] R. Rubinfeld, *A Mathematical Theory of Self-Checking, Self-Testing, and Self-Correcting Programs,* Ph.D. thesis, Department of Computer Science, University of California, Berkeley, 1990. Incorporates efficient random testing into the run-time checking/correcting process.

[30] R. Rubinfeld, "Batch checking with applications to linear functions," *Information Processing Letters,* Vol. 42, pp. 77–80, 1992.

[31] R. Rubinfeld and M. Sudan, "Self-testing polynomial functions efficiently and over rational domains," *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms,* pp. 23–32, 1992. Extends checking methodologies from finite fields to integer and rational domains.

[32] R. Rubinfeld and M. Sudan, "Robust characterizations of polynomials and their applications to program testing," *IBM Research Report RC19156,* 1993, and *Cornell Computer Science Technical Report 93–1387,* 1993.

[33] R. Rubinfeld, "Robust functional equations with applications to self-testing/correcting," *Proc. 35th IEEE FOCS,* 1994. Gives self-testers and self-correctors for a variety of functions. Examples: $\tan x$, $1/(1+\cot x)$, $\cosh x$.

[34] J. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," *Journal of the ACM,* Vol. 27, pp. 701–717, 1980. A fundamental result: to determine (with high probability) whether two polynomials are identical, it generally suffices to check their equality at a random location. Applications include: testing multiset equality; proving that two straight-line arithmetic programs compute the same function.

[35] A. Shamir, "IP = PSPACE," *Proc. 31st IEEE FOCS,* pp. 11–15, 1990. It follows from this result that all PSPACE-complete problems have complex checkers.

[36] F. Vainstein, "Error detection and correction in numerical computations by algebraic methods," *Proc. 9th International Symposium on Applied Algebra, Algebraic Algorithms and Error-Detecting Codes,* 1991. *Springer-Verlag Lecture Notes in Computer Science #539,* pp. 456–464, 1991.

[37] F. Vainstein, *Algebraic Methods in Hardware/Software Testing,* Ph.D. thesis, EECS Department, Boston University, 1993. Uses the theory of algebraic and transcendental fields to design partial complex checkers for rational functions constructed from x, e^x, $\sin(ax+b)$, and $\cos(ax+b)$ using operators + - x / and fractional exponentiation.

[38] L. Valiant, "The complexity of computing the permanent," *Theoretical Computer Science,* Vol. 8, pp. 189–201, 1979. Defines #P-completeness and proves that computing the permanent of a matrix is #P-complete. Also see [25].

[39] M. Wegman and J. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer & System Sciences,* Vol. 22, pp. 265–279, 1981. Includes an idea for a simple check of multiset equality (completed in [12]).

[40] A. Yao. "Coherent functions and program checkers," *Proc. 22nd ACM STOC,* pp. 84–94, 1990. Function f is **coherent** iff on input $\langle x, y\rangle$ one can determine whether or not $f(x) = y$ via a BPP^f algorithm *which is not allowed to query f at x.* Author proves the existence of incoherent (and thus uncheckable) functions in EXP. See also [9].