On the Effectiveness of Run-Time Checks

Meine J.P. van der Meulen, Lorenzo Strigini[1] and Miguel A. Revilla[2]

[1] City University, Centre for Software Reliability, London, UK
WWW home page: http://www.csr.city.ac.uk
[2] University of Valladolid, Valladolid, Spain
WWW home page: http://www.mac.cie.uva.es/~revilla

Abstract. Run-time checks are often assumed to be a cost-effective way of improving the dependability of software components, by checking required properties of their outputs and flagging an output as incorrect if it fails the check. However, evaluating how effective they are going to be in a future application is difficult, since the effectiveness of a check depends on the unknown faults of the program to which it is applied. A programming contest, providing thousands of programs written to the same specifications, gives us the opportunity to systematically test run-time checks to observe statistics of their effects on actual programs. In these examples, run-time checks turn out to be most effective for unreliable programs. For more reliable programs, the benefit is relatively low as compared to the gain that can be achieved by other (more expensive) measures, most notably multiple-version diversity.

1 Introduction

Run-time checks are often proposed as a means to improve the dependability of software components. They are seen as cheap compared to other means of increasing reliability by run-time redundancy, e.g. N-version programming.

Run-time checks (also called *executable assertions* and other names) can be based on various principles (see e.g. Lee and Anderson [3] for a summary), and have wide application. For instance, the concept of design by contract [5] enables a check on properties of program behaviour.

Some run-time checks can detect all failures, for example checks that perform an inverse operation on the result of a software component [1,2]. If the program computes $y = f(x)$, an error is detected if $x \neq f^{-1}(y)$. This is especially attractive when computing $f(x)$ is complex, and the computation of the inverse f^{-1} relatively simple. The argument is then that because computing f^{-1} is simple, the likelihood of failure of this run-time check is low. Also, it seems unlikely that both the primary computation and the run-time check would fail on the same invocation and in a consistent fashion. Together, these factors lead to a high degree of confidence that program outputs that pass the check will be correct. However—as these authors readily admit—such theoretically perfect checks do not exist in many cases, maybe even not in the majority of cases. Run-time checks can then still be applied, but they will in general not be capable of finding all failures. Examples of these partial run-time checks are given by e.g. [12].

Previous empirical evaluation of run-time checks have generally used small samples of programs, or single programs [4,7,11]. Importantly, we run these measures on a large *population* of programs. Indeed, if we wish to learn something general about a run-time check, we need this statistical approach. Measuring the effectiveness of a run-time check on a single program could, given a certain demand profile and enough testing, determine the fraction of failures that the check is able to detect (coverage) for that program, given that demand profile. But in practice, this kind of precise knowledge would be of little value: if one could afford the required amount of testing, at the end one would also know which bugs the program has, and thus could correct them instead of using the run-time check. However, a software designer wants to know whether a certain run-time check is worth the expense of writing and running it, without the benefit of such complete knowledge. The run-time check can detect certain failures caused by certain bugs: the coverage of the check depends on which faults the program contains; and the designer does not usually know this. What matters are the statistics of the check's coverage, given the statistics of the bugs that *may* be present in the program. If a perfect check cannot be had, a check that detects most of the failures caused by those bugs that are likely to be in a program has great value. A check that detects many failures that are possible but are not usually produced, because programmers do not make the mistakes that would cause them, is much less useful. In conclusion, the coverage of a check depends on the distribution of possible programs in which it is to be used.

Here, we choose three program specifications for which we have large numbers of programs, and for each of the three we choose a few run-time checks, then study their coverage. We thus intend to provide some example "data points" of how the coverage can vary between populations of programs. In addition to such anecdotal evidence—evidence that certain values or patterns of values *may* occur—such experiments may contribute to software engineering knowledge if they reveal either some behaviour that runs contrary to the common-sense expectations held about run-time checks, and/or some apparent common trend among these few cases, allowing us to conjecture general laws, to be tested by further research.

For lack of space, we only discuss coverage, or equivalently the probability of undetected failure. We will also not discuss other dependability issues like availability (possibly reduced by false alarms from run-time checks), although these should be taken into account when selecting fault tolerance mechanisms.

2 The Experiment

2.1 The UVa Online Judge

The "UVa Online Judge"-Website [8] is an initiative of one of the authors (Revilla). It contains program specifications for which anyone may submit programs in C, C++, Java or Pascal intended to implement them. The correctness of a program is automatically judged by the "Online Judge". Most authors submit

Table 1. Some statistics on the three problems.

	3n+1			Factovisors			Prime Time		
	C	C++	Pascal	C	C++	Pascal	C	C++	Pascal
Number of authors	5,897	6,097	1,581	212	582	71	467	884	183
First submission correct	2,479	2,434	593	112	294	41	345	636	125

programs repeatedly until one is judged correct. Many thousands of authors contribute and together they have produced more than 3,000,000 programs for the approximately 1,500 specifications on the website.

We study the C, C++ and Pascal programs written to three different specifications (see Table 1 for some statistics, and `http://acm.uva.es/problemset/` for more details on the specifications). We submit every program to a test set, and compare the effectiveness of run-time checks in detecting their failures.

There are some obvious drawbacks from using these data as a source for scientific analysis. First, these are not "real" programs: they solve small, mostly mathematical, problems. Second, these programs are not written by professional programmers, but typically by students, which may affect the amount and kind of programming errors. We have to be careful not to overinterpret the results.

All three specifications specify programs that are memory-less (i.e. earlier demands should not influence program behaviour on later ones), and for which a demand consists of only two integer input values. Both restrictions are useful to keep these initial experiments simple and the computing time within reasonable bounds. The necessary preparatory calculations for the analysis of these programs took between a day and two weeks, depending on the specification.

2.2 Running the Programs

For a given specification, all programs were run on the same set of demands. Every program is restarted for every demand, to ensure the experiment is not influenced by history, e.g. when a program crashes for certain demands or leaves its internal state corrupted after execution of a demand (we accept the drawback of not detecting bugs with history-dependent behaviour). We set a time limit on the execution of each demand, and thus terminate programs that are very slow, stall, or crash. We only use the first program submitted by each author and discard all subsequent submissions by the same author. These subsequent submissions have shown to have comparable fault behaviour and this dependence between submissions would complicate any statistical analysis.

For each demand, the outputs generated by all the programs are compared. Programs that produce exactly the same outputs on every demands form an "equivalence class". We evaluate the performance of each run-time check for each equivalence class.

For all three specifications, we chose the equivalence class with the highest frequency as the *oracle*, i.e. the version whose answers we consider correct. We challenged each oracle in various ways, but never found any of them to have

Table 2. Classification of execution results with plausibility checks.

Output of primary	Output valid	Plausibility check	Effect from system viewpoint
Correct	Yes	Accept	Success
Correct	Yes	Reject	False alarm
Incorrect	Yes	Accept	Undetected failure
Incorrect	Yes	Reject	Detected failure
Incorrect	No	-	Detected failure

failed. For each specification, the test data were chosen to exhaustively cover a region in the demand space. In other words, we assume (arbitrarily) a demand profile in which all demands that occur are equiprobable.

2.3 Outcomes of Run-Time Checks

Run-time checks test properties of the output of a software component (the primary), based on knowledge of its functionality. In the rest of this paper we distinguish two types of run-time checks: *plausibility checks* and *self-consistency checks* (SCCs). The latter, inspired by Blum's "complex checkers" [12], use additional calls to the primary to validate its results, by checking whether some known mathematical relationship that must link its outputs on two or more demands does hold.

Checks on the values output by the primary are only meaningful if the output satisfies some minimal set of syntactic properties, one of which is that an output exists. Other required properties will be described with each specification. We call an output that satisfies this minimal set of properties "valid" (in principle this validaty check also constitutes a run-time check). We separate the check for "validity" from the "real" run-time checks, because it otherwise remains implicit and a fair comparison of run-time checks is not possible.

Table 2 shows how we classify the effects of plausibility checks. There are two steps: first, a check on the validity of the output of the primary; second, if this output is valid, a plausibility check on the output. There is an undetected failure (of the primary) if both the primary computes an incorrect valid output and the checker fails to detect the failure. Our plausibility checks did not cause any false alarms. Also note that a correct output cannot be invalid.

With self-consistency checks, the classification is slightly more complex (Table 3): we have to consider that one way the self-consistency check may fail is because its additional calls to the primary do not elicit valid outputs (e.g., they cause the primary to crash). We then assume that the self-consistency check will fail to reject the primary's output, i.e., that an undetected failure ensues. We could have made the decision to reject the output of the primary if the self-consistency check fails in this way; this would lead to slightly different results. False alarms did occur, which we do not analyse here for lack of space.

Table 3. Classification of execution results with self-consistency checks.

Output of primary	Output valid	Output of second call to primary by self-consistency check	Effect from system viewpoint
Correct	Yes	Consistent	Success
Correct	Yes	Inconsistent	False alarm
Correct	Yes	Invalid output	Success
Incorrect	Yes	Consistent	Undetected failure
Incorrect	Yes	Inconsistent	Detected failure
Incorrect	Yes	Invalid output	Undetected failure
Incorrect	No	-	Detected failure

3 Results for the "3n+1" specification

Short specification. A number sequence is built as follows: start with a given number n; if it is odd, multiply by 3 and add 1; if it is even, divide by 2. The sequence length is the number of required steps to arrive at a result of 1. Determine the maximum sequence length (max) for all values of n between two given integers i, j, with $0 < i, j \leq 100,000$. The output of the program is the triple: i, j, max.

We tested "3n+1" with 2500 demands ($i, j \in 1..50$). The outputs of the programs were deemed correct if the first three numbers in the output exactly matched those of the oracle. We consider an output "valid" if it contains at least three numbers. In the experiment we discard non-numeric characters and the fourth and following numbers in the output. The programs submitted to "3n+1" have been analysed in detail in [9]; this paper provides a description of the faults present in the equivalence classes.

3.1 Plausibility Checks

We use the following plausibility checks for the "3n+1"-problem:

1. The maximum sequence length should be larger than 0.
2. The maximum possible sequence length (given the range of inputs) is 476.
3. The maximum sequence length should be larger than $log_2(max(i, j))$.
4. The first output should be equal to the first input.
5. The second output should be equal to the second input.

We measure the effectiveness of a run-time check as the improvement it produces on the average *probability of undetected failure on demand* (*pufd*). Without run-time checks, a program's probability of undetected failure equals its probability of failure per demand (*pfd*).

Figure 1 shows the improvement in average *pufd* given by these plausibility checks, depending on the average *pufd* of a pool of programs. We manipulate this average by removing, one by one, from the original pool of 13575 programs,

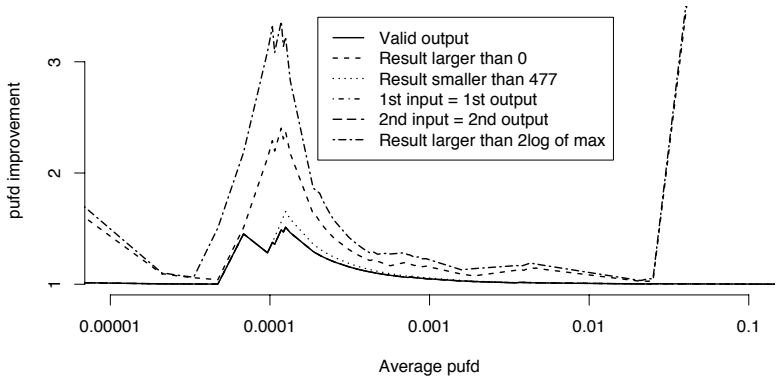


Fig. 1. The improvement of the *pufd* of the primary for the various plausibility checks for "3n+1". The curves for "1st input = 1st output" and "2nd input = 2nd output" are invisible because they coincide with the curve for "Valid output".

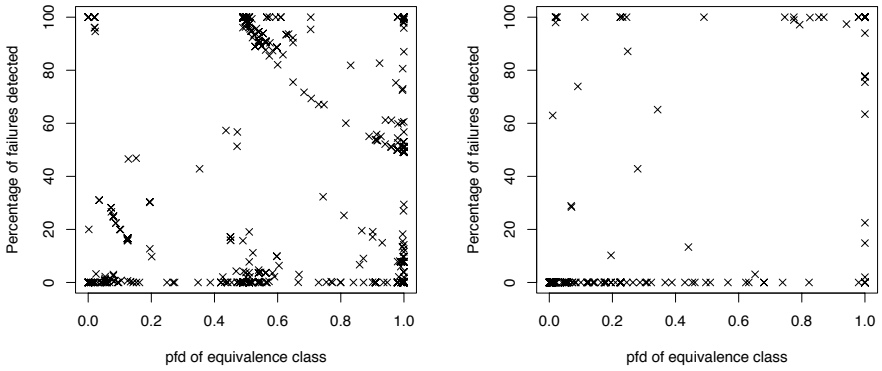
the programs with the highest *pufd*. The more programs have been removed, the lower the average *pufd* of the remaining pool.

The graph clearly shows that many of these run-time check are very effective for unreliable programs (the right-hand side of the graph). More surprising is that the impact is quite pronounced at a *pufd* of the pool around 10^{-4}, while it is much lower for the rest of the graph. Apparently, these checks are effective for some equivalence classes that are dominant in the pool for that particular *pufd* range. Upon inspection, it appears that these programs fail for $i = j$.

The gain in *pufd* is for most of the graph only about 20%, but the peak reaches a factor of 3.2 for the plausibility check "Result $> log_2(max(i, j))$", a significant improvement over a program without checks. The check "Result>0" is mainly effective for programs that initialise the outcome of the calculation of the maximum sequence length to 0 or -1, if they abort the calculation before setting the result to a new value. This appears to be caused by an incorrect "for"-loop which fails when $i > j$. The check "Result<477" is not very effective. The failures it detects have mostly to do with integer overflow and uninitialised variables.

The check "Result $> log_2(max(i, j))$" is the most effective of all. It catches a few more programming faults than "Result > 0", especially of those programs that do not cover the entire range between the two inputs i and j for the calculation of the maximum sequence length.

Figure 2(a) gives some more detail of the performance of this plausibility check. It shows the percentage of failures detected for each equivalence class. We can make various observations. First, for many equivalence classes there is no effect (many crosses with a coverage of 0%). Second, since there are more crosses



(a) (b)

Fig. 2. Values of the error detection coverage of (a) the plausibility check "Result $>$ $log_2(max(i,j))$" for the equivalence classes of "3n+1" programs, and (b) the plausibility check "$i \leq j$" for the equivalence classes of "Factovisors" programs. Each cross represents an equivalence class. The horizontal axis gives the average *pfd* of the equivalence class, the vertical axis the percentage of its incorrect outputs that the check detects.

in the right-hand side of the graph, this check seems to be more effective when the primary programs tend to be less reliable (i.e., for development processes that tend to deliver poor reliability). We must say "seem" here, because this graph lacks information about the frequencies of the various programs (sizes of the equivalence classes). Third, this plausibility check still detects faults in the left-hand side of the graph, i.e. for the more reliable programs.

The plausibility check "First output equals first input" mainly catches problems caused by incorrect reading of the specification: some programs do not return the inputs, or not always in the correct order. These faults lead to very unreliable programs, and the effects of this plausibility check are not visible in Figure 1 because they manifest themselves (i.e. differ from the curve for "Valid output") for average *pufd*s larger than 0.1.

The result of the plausibility check "Second output equals second input" is almost equal to the previous one. There are a few exceptions, for example when the program returns the first input twice.

3.2 Self-Consistency Checks

If we denote the calculation of the maximum sequence length as $f(i,j)$, then:

$$f(i,j) = f(j,i) \tag{1}$$

and:

$$f(i,j) = max(f(i,k), f(k,j)) \quad \text{for} \quad k \in i..j \tag{2}$$

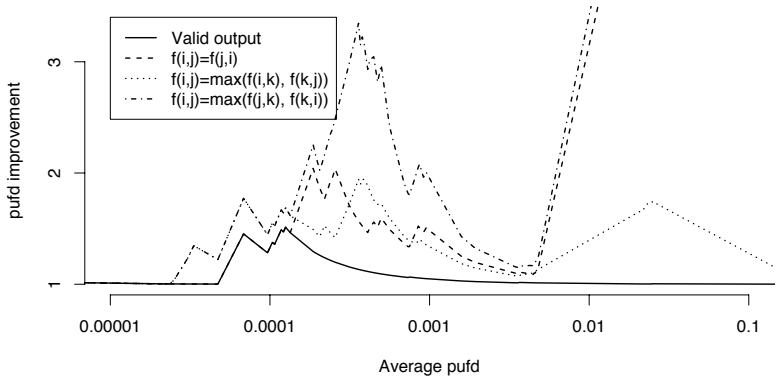


Fig. 3. Improvement in the average *pufd* of the primary for the various self-consistency checks for "3n+1".

and, if we combine these two properties:

$$f(i,j) = max(f(j,k), f(k,i)) \quad \text{for} \quad k \in i..j \tag{3}$$

Figure 3 presents the effectiveness of these self-consistency checks (for the experiment, we choose $k = \lfloor (i+j)/2 \rfloor$). Like our plausibility checks, these self-consistency checks appear to be very effective for unreliable programs.

The first self-consistency check mainly detects failures of programs in which the calculation of the maximum sequence length results in 0 or -1 for $i > j$. The second mainly finds failures caused by incorrect calculations of the maximum sequence length.

The third self-consistency check attains an improvement comparable to that of the plausibility check "Result $> log_2(max(i,j))$", but with a shifted peak. It appears that they catch different faults in the programs. As already stated, the peak of "Result $> log_2(max(i,j))$" is caused by programs failing for $i = j$ (which none of our self-consistency checks can detect) while this self-consistency check detects failures caused by faults in the calculation of the maximum sequence length as well as programs that systematically fail for $i > j$.

The fact that the plausibility checks and the self-consistency checks tend to detect different faults is highlighted by Figure 4, which shows the performances of the combined plausibility checks, the combined self-consistency checks and the combination of all run-time checks.

4 Results for the "Factovisors" specification

Short specification. For two given integers $0 \leq i, j \leq 2^{31}$, determine whether j divides $i!$ (factorial i) and output "j divides i!" or "j does not divide i!".

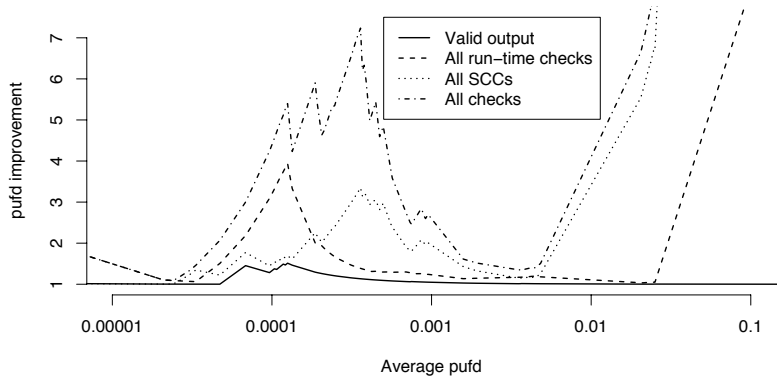


Fig. 4. Improvement in the average *pufd* of the primary for combinations of run-time checks for "3n+1".

We tested "Factovisors" with the 2500 demands ($i, j \in 1..50$). We consider an output "valid" if it contains at least two strings and the second is "does" or "divides". The main reason for invalid outputs appears to be absence of outputs.

4.1 Plausibility Checks

We use the following plausibility check for "Factovisors":

1. If $i \geq j$, the result should be "j divides i!".

Figure 2(b) shows the coverage of the run-time check "$i \geq j$" for each equivalence class. It is remarkable, again, that the crosses are spread over the entire plane: this check has some effect for equivalence classes with a large range of reliabilities. We also again observe the large number of crosses for a coverage of 0%, showing the check to detect no failure at all for that class of programs.

Figure 5 shows the *pufd* improvement caused by the plausibility check. As for "3n+1", we observe that the run-time check is very effective for unreliable programs. For pools of programs with average *pufd* between 10^{-4} and 10^{-2} the reliability improvement varies between 1 and 1.6.

The graph shows a peculiarity for *pufd*s smaller than 10^{-4}: the improvement approaches infinity. This is because as we remove programs from the pool, the faulty programs in the pool eventually become a "monoculture", a single equivalence class, and the check happens to detect all the failures of this class of incorrect programs. Here, the pool with the lowest non-zero average *pufd* contains 447 correct programs and 21 incorrect ones in the same equivalence class; the plausibility check detects the failures of these 21 incorrect programs.

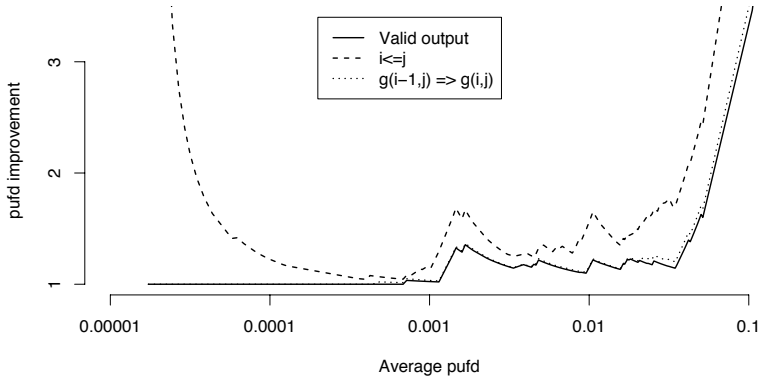


Fig. 5. The effectiveness of the run-time checks for "Factovisors".

4.2 Self-Consistency Checks

If we call $g(i, j)$ the Boolean representation of the output of the program, with $g(i, j) = true \equiv$ "j divides i!", $g(i, j) = false \equiv$ "j does not divide i!", then:

$$g(i-1, j) \implies g(i, j) \quad \text{with} \quad i \neq 1 \tag{4}$$

As can be seen in Figure 5, the effect of this self-consistency check is minimal: the reliability improvement is never substantially greater than that given by the validity check.

5 Results for the "Prime Time " specification

Short specification. Euler discovered that the formula $n^2 + n + 41$ produces a prime for $0 \leq n \leq 40$; it does however not always produce a prime. Calculate the percentage of primes the formula generates for n between two integers i and j with $0 \leq i \leq j \leq 10,000$.

We tested "Prime Time" on 3240 demands ($i \in 0..79$, $j \in i..79$). The outputs were deemed correct if they differed by most 0.01 from the output of the oracle, allowing for round-off errors (the answer is to be given with two decimal digits).

The output is considered "valid" when it contains at least one number. We discard all non-numeric characters and subsequent digits from the output.

5.1 Plausibility Checks

The programs for "Prime Time" calculate a percentage, therefore:

1. The result should be larger than or equal to zero.

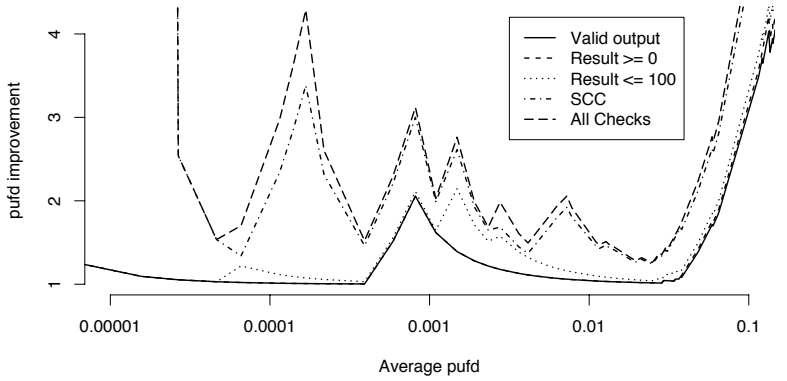


Fig. 6. The effectiveness of the run-time checks for "Prime Time". The curve for the plausibility check "Result $\geq$ 0" is not visible, because it coincides with the one for "Valid output".

2. The result should be smaller than or equal to a hundred.

Figure 6 presents the effectiveness of the plausibility checks for "Prime Time". The plausibility check "Result $\geq$ 0" appears to have virtually no effect. The plausibility check "Result $\leq$ 100" has some effect, but not very large.

5.2 Self-Consistency Checks

If we denote the result of the calculation of the percentage with $h(i, j)$, then:

$$h(i,j) = \frac{h(i,k) \times (k-i+1) + h(k+1,j) \times (j-k)}{j-i+1} \quad \text{for} \quad i \leq k < j \qquad (5)$$

Obviously, this check is not available when $i = j$. It is quite elegant: the computing time will not be excessively more than computing $h(i, j)$. For the experiment, we choose $k = \lfloor (i + j)/2 \rfloor$.

The effectiveness of the self-consistency check is shown in Figure 6. It is much more effective than the plausibility check "Result $\leq$ 100". We observe the same phenomenon for low *pufd*s as for "Factovisors": the effectiveness of the self-consistency check approaches infinity. When we combine the plausibility checks and the self-consistency check, we observe that the two complement each other: the combination is (slightly) more effective than the self-consistency check alone.

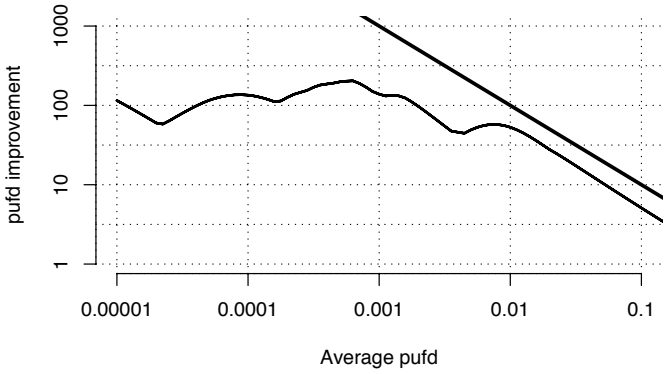


Fig. 7. Improvement of the *pufd* of a pair of randomly chosen C programs for "3n+1", relative to a single version. The horizontal axis shows the average *pufd* of the pool from which both C programs are selected. The vertical axis shows the *pufd* improvement ($pufd_A/pufd_{AB}$). The diagonal represents the theoretical reliability improvement if the programs fail independently, i.e. $pufd_{AB} = pufd_A.pufd_B$. (This figure is based on [10].)

6 Run-Time Checks vs. Multiple-Version Diversity

A question that begs answering is: how do run-time checks compare to other forms of run-time fault tolerance? Using results we reported previously [10], we can compare our run-time checks against multiple-version diversity for "3n+1".

We observed (see Figure 7) that two-version diversity would become more effective with decreasing mean probability of failure on demand of the pool of programs from which the pair is selected, until a "plateau" is reached (between a *pufd* of 10^{-5} and 10^{-3}) with an improvement factor of about a hundred (note that the opposite trend—effectiveness decreasing with decreasing mean *pfd*—is also possible, as proved by models and empirical results [6]). For run-time checks the opposite occurs: their effectiveness decreases with decreasing average *pufd* of the primary reaching a fairly low improvement factor. The improvement factor of using diversity is significantly higher than that of applying run-time checks.

For these programs, it seems that these run-time checks could be the better choice for testing in the early phases of development, when the *pufd* of programs is still high, and multiple-version diversity when *pufd*s of programs become low.

7 Conclusion

The results in this paper are of course specific to these three specifications, the programs submitted by these anonymous authors, the run-time checks we devised, and the demand profiles we used (uniform in a subset of the demand

space). There are however some commonalities among the three sets of results, and we will tentatively discuss these here, while keeping in mind the limitations of this research.

First, we observe that the majority of the run-time checks considered are very effective for unreliable programs or have no effect at all.

Then, if we only look at pools of primaries with average *pufd* between 10^{-4} and 10^{-2}, the *pufd* improvement factor of the primary-checker pair is comparable for all three specifications: in the range 1–4. Over this range, the average improvement is less than 2 for all run-time checks considered.

Some run-time checks provide almost no benefit. It would be of great importance to be able to predict which checks are effective and which are not, but for the time being this seems not to be possible.

These plausibility checks appear to detect a different set of failures than the self-consistency checks, so that combining them is more effective than applying either one alone. So, the apparent "diversity" between the two kinds of checks did bring the benefit of some complementarity.

For pools of primaries with an average *pufd* lower than 10^{-2}, the *pufd* improvement achieved by the run-time checks considered for "3n+1" is far less than would have been achieved by applying multiple-version redundancy. In these analyses, the *pufd* improvement realised by multiple-version redundancy is at least a factor of a hundred better.

A natural comment on this work could be that since we have implemented simple-minded checks, it is not surprising that they only catch the simple-minded programming errors that cause highly unreliable programs. But this is actually a *non sequitur*. It is true that we do not expect expert programmers to produce highly unreliable programs, but our checks are "simple-minded" only in being based on simple mathematical properties of these specifications. There is no a priori reason why they should only catch simple-minded implementation errors: implementation errors are often caused by misunderstanding details of the specification or of the program itself, not of some mathematical property of the specification that is of little interest to the programmers. Likewise, there is no a priori reason for naive errors normally to cause faults which cause very high failure rates.

A tempting conjecture generalising the results we observed is that for some reason simple run-time checks tend (in some types of programs?) only to detect the failures in very unreliable programs. This would be an attractive "natural law" to believe and would simplify many decisions on applying run-time checks, so that it may be worth exploring further, since without some solid, plausible explanation (e.g. based on the psychology of programmers) or overwhelming empirical evidence, it would appear wholly unjustified.

Our measure of effectiveness as average improvement in *pufd* may be questioned. It is such that even if a check C has 100% coverage for the failures produced by a set of dangerous possible bugs, B, it will still be assessed as having negligible effectiveness if the bugs in set B occur with negligible probability in actual software development. Some may object that if C is the only check

that can detect the effects of B-type bugs, and given the uncertainty on the probabilities of B these bugs being actually produced, a prudent designer will still use C. This objection is certainly right if C has negligible cost (implementation cost, cost in run-time resources, risk of bugs in C causing false alarms, etc). But whenever these costs are non-negligible, they must be weighted against C's potential benefits, as we do.

Acknowledgement

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council via the Interdisciplinary Research Collaboration on the Dependability of Computer Based Systems (DIRC), and via the Diversity with Off-The-Shelf Components (DOTS) project, GR/N23912.

References

1. M. Blum and H. Wasserman. Software reliability via run-time result-checking. Technical Report TR-94-053, International Computer Science Institute, October 1994.
2. A. Jhumka, F.C. Gärtner, C. Fetzer, and N. Suri. On systematic design of fast and perfect detectors. Technical Report 200263, École Polytechnique Fédérale de Lausanne (EPFDL), School of Computer and Communication Sciences, September 2002.
3. P.A. Lee and T. Anderson. *Fault Tolerance; Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer, 2nd edition, 1981.
4. N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The use of self checks and voting in software error detection: An empirical study. In *IEEE Transactions on Software Engineering*, volume 16(4), pages 432–443, 1990.
5. B. Meyer. Design by contract. *Computer (IEEE)*, 25(10):40–51, October 1992.
6. P. Popov and L. Strigini. The reliability of diverse systems: A contribution using modelling of the fault creation process. *DSN 2001, International Conference on Dependable Systems and Networks, Göteborg, Sweden*, July 2001.
7. M. Rela, H. Madeira, and J.G. Silva. Experimental evaluation of the fail-silent behavior of programs with consistency checks. In *FTCS-26, Sendai, Japan*, pages 394–403, 1996.
8. S. Skiena and M. Revilla. *Programming Challenges*. Springer Verlag, March 2003.
9. M.J.P. van der Meulen, P.G. Bishop, and M. Revilla. An exploration of software faults and failure behaviour in a large population of programs. In *The 15th IEEE International Symposium of Software Reliability Engineering, 2–5 November 2004, St. Malo, France*, pages 101–12, 2004.
10. M.J.P. van der Meulen and M. Revilla. The effectiveness of choice of programming language as a diversity seeking decision. In *EDCC-5, Fifth European Dependable Computing Conference, Budapest, Hungary, 20–22 April, 2005*, April 2005.
11. J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson. Reducing critical failures for control algorithms using executable assertions and best effort recovery. In *DSN 2001, International Conference on Dependable Systems and Networks, Goteborg, Sweden*, 2001.
12. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.