

Experiences with the Design of a Run-Time Check

Meine J.P. van der Meulen¹ and Miguel A. Revilla²

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. Run-time checks' main point of attractiveness is that they are supposed to be easy to implement. Also, they are implicitly assumed to be effective in detecting incorrect outputs. This paper reports the results of an experiment designed to challenge these assumptions about run-time checks.

The experiment uses a subset of 196 of 867 programs (primaries) solving a problem called "Make Palindrome". This is an existing problem on the "On-Line Judge" website of the university of Valladolid. We formulated eight run-time checks, and posted this problem on the same website. This resulted in 335 programs (checkers) implementing the run-time checks, 115 of which are used for the experiment.

In this experiment: (1) the effectiveness of the population of possibly faulty checkers is very close to the effectiveness of a correct checker; (2) the reliability improvement provided by the run-time checks is relatively small, between a factor of one and three; (3) The reliability improvement gained by using multiple-version redundancy is much higher. Given the fact that this experiment only considers one primary/Run-Time Check combination, it is not yet possible to generalise the results.

1 Introduction

Redundancy is a means to improve the reliability of software components. Much research has been invested in multiple-version redundancy, e.g. 1-out-of-2 systems. Much less research has been invested in asymmetrical redundancy, e.g. the use of run-time checks, RTCs, see Figure 1. In these cases a primary program is checked by an, ideally relatively simple, RTC.

RTCs 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. We are interested in answering questions like whether RTCs are effective and how their performance compares to that of symmetrical redundancy. We also want to confirm or reject

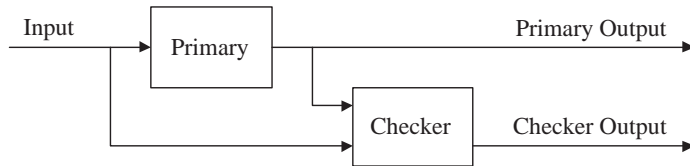


Fig. 1. Primary/Checker model.

common conjectures, such as that RTCs are so simple that we may assume that they are correctly programmed.

RTCs (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 RTCs 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 RTC is low. Also, it seems unlikely that both the primary computation and the RTC 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. RTCs can then still be applied, but they will in general not be capable of finding all failures. Examples of these partial RTCs are given by e.g. [11].

Previous empirical evaluation of RTCs have generally used small samples of programs, or single programs [4,7,10]. Importantly, our experiment involves a *population* of programs, both of the primary and of the checker, because we think that in order to learn something general about RTCs, we need a statistical approach. We can now study the complex interplay between (possible faulty) primaries combined with (partial, possibly faulty) checkers.

The interaction between the primary and the checker is complex because the performance of the checker is dependent on that of the primary. An example of this interaction is that it may be that improving the primary may lead to a rise in the probability of undetected failure. Suppose that a primary (e.g., to compute $f(n) = (n+2)(n-2)$) is incorrect ($f(5) = 27$) and that the checker (e.g., $f(n) \leq n^2$) detects the incorrect outputs of the primary. Now, the programmer changes the primary ($f(5) = 23$); its output is now closer to the correct answer, but still incorrect. It may now be that the checker is unable to detect the incorrect outputs. As a result, the probability of undetected failure may have increased.

Table 1. Sample inputs and sample outputs for the primary.

Sample Input	Sample Output
abcd	3 abcdcba
aaaa	0 aaaa
abc	2 abcba
aab	1 baab
abababaabababa	0 abababaabababa
pqrsabcdpqrs	9 pqrsabcdpqrqpdcbasrqp

2 The Experiment

2.1 The UVa Online Judge

The “Uva Online Judge”-Website (<http://acm.uva.es>, [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 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.

2.2 Specification of the primary

For the primary, we took a specification from the Online Judge formulated by Md. Kamruzzaman: a program to generate palindromes. It takes an input string of 1000 or less lower case characters and makes it into a palindrome by inserting lower case characters into the input string at any position. The number of characters inserted shall be as low as possible. The output is the number of characters inserted, followed by the resulting palindrome. See for a complete specification the Online Judge website, <http://acm.uva.es/p/v104/10453.html>, and Table 1 for some examples of correct input and output combinations.

2.3 Specification of the checker

Based on the specification of the primary, we formulated a specification for its checker. The checker (<http://acm.uva.es/p/v104/10848.html>) takes as its input a string of 5000 or less ASCII characters (5000 to allow for faults in the primary, leading to the output of many characters). It tests this string for the following properties:

- P1. It consists of a first string of lower case characters (length ≤ 1000), a single space, an integer ($\geq 0, \leq 1000$), a single space, and a second string of lower case letters (length ≤ 2000).

Table 2. Sample inputs and sample outputs for the checker.

Sample Input	Sample Output
abcd 3 abcdcba	TTTTTT The solution is accepted
aaaa 3 abcdcba	TTFFTT The solution is not accepted
abc 2 abcdcba	FTTTFT The solution is not accepted
aab b baab	FFFFFF The solution is not accepted
abababaabababa 0 abababaabababa	TTTTTT The solution is accepted
pqrsabcdpqrs 9 pqrsabcdpqrqdcbasrqp	TTTTTT The solution is accepted

- P2. P1 & the second string is a palindrome.
- P3. P1 & all letters of the first string appear in the second string.
- P4. P1 & the frequency of every letter in the second string is at least the frequency of this letter in the first string.
- P5. P1 & the first string can be made out of the second string by removing 0 or more letters (and leaving the order of the letters intact).
- P6. P1 & the length of the second string is equal to the length of the first string plus the value of the integer.
- P7. P1 & the value of the integer is smaller than the length of the first string.

Obviously, when all properties are true, the output of the primary may still be faulty.

The output consists of the value "T" or "F" (for True and False) for every property in the list above, and a statement "The solution is accepted" if all properties are true, and "The solution is not accepted" otherwise. We will call this property P8. See Table 2 for some examples of correct input and output combinations.

Although the checkers implement all eight different properties, we will analyse these separately, as if the checker programs only implement one of these. When we address any of the run-time checks, we will use the abbreviation RTC. When we address the implementations of one of the properties P1-8, we will use the abbreviations RTC1-8.

2.4 System Behaviour

Table 3 shows how we classify the effects on the system based on the outputs of the primary and the run-time checks. The effects from the system viewpoint are rather obvious, except for the consequences of "No output" from the RTC (this includes invalid output). We have chosen to accept the output of the primary in these cases; the other option would also have been possible, it would have increased the number of false alarms. Our choice is based on the assumption that false alarms of RTCs are in general very undesirable.

RTC1 differs from RTC2-8, because it is purely a syntactical check of the input. It is a necessary precondition to be able to do any of the other checks. It

Table 3. Classification of execution results with RTCs.

Output of primary	run-time check	Effect from system viewpoint
Correct	Accept	Correct
Correct	Reject	False alarm
Correct	No output	Correct
Incorrect	Accept	Undetected failure
Incorrect	Reject	Detected failure
Incorrect	No output	Undetected failure

is interesting to separate this check from RTC2-8, because it gives us an idea how much the more application specific RTC2-8 add to this basic syntactic check.

2.5 Equivalence Classes

We subjected the primary to 10,000 demands: strings of lower case characters. Each string has a random length between 1 and 30 characters with random characters from the set “a”..“e”. The reason for the limited character set is that cases with character repetition will more frequently occur. The reason for the maximum length of the string is to limit the execution time of the primary.

We sorted the primaries in “equivalence classes”, i.e. sets of programs producing exactly the same output. There is more than 1 correct equivalence class, because for almost all inputs there is more than one correct solution, e.g. the correct output to the input “ab” may be “1 aba” or “1 bab”.

The primaries gave mostly equal, but also many different outputs to the 10,000 demands; in total there were 529,433 different outputs to the 10,000 inputs. to reduce computing time, we randomly selected 17,241 of these (approximately 1/30). We generated an input file to the checkers by combining each output with the corresponding input. This input file is used to determine the equivalence classes of the checkers and was only used for this purpose; for the rest of the experiment we used the 10,000 demands as used for determining the primary equivalence classes. We assumed that these 17,241 demands are sufficient to discern the different checker equivalent classes.

For every primary equivalence class we made an input file for the checkers by combining the 10,000 demands (the same for every primary) and their outputs. We executed every combination of primary and checker equivalence classes with the appropriate input files. This was computationally quite intensive; the computation took approximately three days.

2.6 Score Functions

Assume a specification for the primary, S_π :

$$S_\pi(x, y) \equiv \text{“}y \text{ is valid primary output for input } x\text{”} \quad (1)$$

Then, we define the score function ω_π for a random primary π as:

$$\omega_\pi(\pi, x) \equiv \neg S(x, \pi(x)) \quad (2)$$

I.e., the score function is true when the primary π fails to compute a valid output y for a given input x .

The behaviour of an RTC σ can be described as:

$$\sigma(x, y) \equiv \text{“}y \text{ is accepted as valid primary output for input } x\text{”} \quad (3)$$

Note the similarity to the specification of the primary, S_π . Whereas the specification is supposed to be correct, we assume that the checker may be faulty: it may erroneously accept an incorrect pair (x, y) . The checker fails if there is a discrepancy with the specification. The score function ω_σ for an RTC is:

$$\omega_\sigma(\sigma, x, y) \equiv S_\pi(x, y) \oplus \sigma(x, y) \quad (4)$$

I.e., the score function is true when the checker fails to recognize whether y is valid primary output for input x or not.

For our system as depicted in Figure 1 and the variables (x, π, σ) for the input, the primary and the checker, there are four possibilities:

1. $\neg\omega_\pi(\pi, x) \wedge \neg\omega_\sigma(\sigma, x, \pi(x))$: Correct operation.
2. $\neg\omega_\pi(\pi, x) \wedge \omega_\sigma(\sigma, x, \pi(x))$: False alarm.
3. $\omega_\pi(\pi, x) \wedge \neg\omega_\sigma(\sigma, x, \pi(x))$: Detected failure.
4. $\omega_\pi(\pi, x) \wedge \omega_\sigma(\sigma, x, \pi(x))$: Undetected failure.

We have calculated the score functions $\omega_\pi(\pi, x)$ and $\omega_\sigma(\sigma, x, y)$ for the primary and the checker equivalence classes.

2.7 Subsets for the Experiment

There are 867 submissions for the primary specification. We included the primaries that are written in C, C++ or Pascal, compile and provide output within one second. This left 566 primaries. Then we excluded primaries that fail for all inputs, that left 484 primaries. From these we used the first submission of each author: 196 primaries. There are various reasons for selecting the first submissions:

1. We do not want to include more than one submission of a single author, because subsequent submissions are shown to be highly similar, and that would corrupt our statistical analyses.
2. The variability between first versions is higher, e.g. because later submissions are more likely to be correct. This gives more room for statistical analyses.

3. A first submission is most comparable to a first submission in a “normal” development process, because feedback to the authors differs from normal feedback in various ways, e.g. the Online Judge will not communicate for which input the program failed.

There are 395 submissions for the checker specification. We included the checkers that are written in C, C++ or Pascal, compile and provide output within one second. This left 335 checkers.

For these checkers we compute the average FAR (false alarm rate, the fraction of false alarms) for a specific primary π for RTC8 (for the calculations: $TRUE = 1, FALSE = 0$):

$$FAR(\pi) = \sum_{x \in D} \sum_{\sigma \in R_\sigma} (1 - \omega_\pi(\pi, x)) \cdot \omega_\sigma(\sigma, x, \sigma(x)) \cdot Q(x) \quad (5)$$

R_σ is the set of checkers, $Q(X)$ is the demand profile over the demand space D . D is the test set of 10,000 demands; we assume that each of the 10,000 demands is equiprobable, and therefore: $Q(x) = 1/10,000$.

We excluded checkers that have a FAR of more than 0.1 for any primary or do not provide sensible output at all (manual check), that left 306 checkers. From these we used the first submission of each author: 118 checkers.

The rationale for excluding checkers with a high FAR is that these will normally be quickly detected during development. This is our modelling of the debugging process of the checkers. There is only one checker left with a FAR larger than zero. This checker fails for RTC1 (and subsequently often for RTC2-8) in a rather erratic way.

3 Observations

3.1 General

The first observation was already done during selection of a suitable primary for this research. It appeared that it is only possible for a small subset of the problems of the Online Judge to formulate meaningful RTCs. In many cases, the output of a program is a (set of) number(s) for which it is not possible to formulate an inverse function to the input, or even an interesting weaker relationship.

We chose this primary because it is possible to define RTCs, and a sufficient number of submissions for the primary is available.

3.2 The Specification of the Checker

The checker specification appeared to be incomplete: we forgot to specify a lower bound on the length of the strings. This leaves it to the programmers to decide whether an empty string is correct input or not. As it appears, some of the authors allow empty strings. This ambiguity has consequences for property 2: is

Table 4. Most frequent equivalence classes in the first submissions for the primary.

Fault	Freq.	PFD	Detection
Correct	129	0.00	No detection.
Adds too many characters to input string.	5	0.54	No detection.
Forgets last character in input string.	2	0.69	P3 (28%), P4 (55%), P5 (100%), P8 (100%)
Adds too many characters to input string.	2	0.55	No detection.
Output string is not always a palindrome.	2	0.12	P2 (100%), P8 (100%)
...			
Often fails to output second string.	1	0.94	P1-8 (100%)
Often outputs very large integers.	1	0.99	P126 (93%), P345 (96%), P8 (100%)
...			
No integer in output.	0	1.00	P1-8 (100%)
Often outputs control character at end of second string.	0	0.94	P1-8 (100%)

an empty string a palindrome or not? As it happens some authors who accept empty strings consider an empty string to be a palindrome, others don't.

A peculiar problem occurs for property 4: P1 & the frequency of every letter in the second string is at least the frequency of this letter in the first string. We intended to write: P1 & the frequency of every letter in the first string is smaller than or equal to the frequency of this letter in the second string. Peculiarly, most authors interpreted it this way. They thus wrote a stronger test than was specified.

We argue that problems with specifications are common, and that this observation does therefore not invalidate the conclusions of the paper. Maybe even to the contrary: they might even be supporting it, since specifying is part of the development process, and a possible source of errors.

There is one equivalence class containing a correct checker, i.e. a program that does not accept empty strings and interprets P4 as written in the specification. Nobody submitted a correct program as their first submission. This implies there is no correct submission in the set of programs we do our analyses on in this paper, c.f. 2.7.

3.3 Faults in the primary

To give an idea of the kinds of faults made, Table 4 presents some of the equivalence classes of the primaries. There are 17 correct equivalence classes, with in total 129 submissions. There are 67 incorrect submissions in 59 equivalence classes. Only five equivalence classes contain more than one submission, these are listed in the table. The fact that equivalence classes tend to only contain one program indicates that authors tend to choose different approaches and tend to make different mistakes. Furthermore, the presence of many different equivalence

Table 5. Most frequent equivalence classes in the first submissions for the checker.

Fault	Freq.
Correct, except that P4 is interpreted more extensively.	54
Fails when input contains control characters.	5
Fails when integer in input string is very large.	4
Always outputs “TTTTTT The solutions is accepted”.	3
Fails when second string is absent.	2
Fails when there is a control character in the second string.	2
Fails when integer in input string is very large.	2
Fails when there is a control character in the integer or in the second string.	2
...	
Fails for P4, behaves partly as specified.	1
Fails for P4, behaves as specified, but strange other fault.	1
Accepts empty second string, assumes empty string is palindrome.	1
...	
Correct.	0
...	

classes for correct solutions indicates that authors do not tend to copy solutions from each other, one of the worries for the usefulness of the data.

The table also shows whether the faults are detected by a correct checker, and how effective these checks are.

3.4 Faults in the checker

Table 5 presents some of the equivalence classes of the checkers. There are 54 correct submissions in one equivalence class (there is only one way to solve this problem). There are 64 incorrect submissions in 51 equivalence classes, only seven of these contain more than one submission. This again indicates that the authors do not tend to make exactly the same mistake.

Seven submissions give no output when the second string is empty.

The most frequent mistake is that a checker fails when there are special ASCII characters in the input string, e.g. the NULL character. This is problematic, because some primaries fail in a way that produces exactly these characters. This is caused by the fact that the solution to the “Make Palindrome”-problem typically includes array manipulation. This, combined with a bug leading to a pointer being out of array bounds, leads to possibly outputting these characters. Important is here that this observation may undermine the conjecture of independence between primaries and checkers.

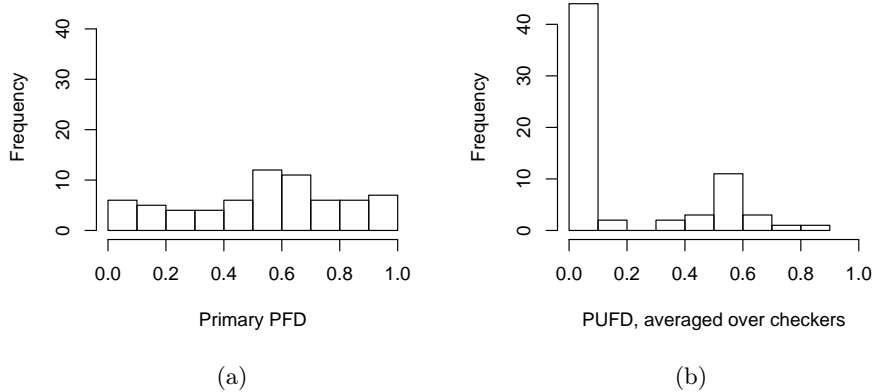


Fig. 2. (a) Histogram of the PFDs of the primaries; (b) Histogram of the average probability of undetected failure for the same primaries, with RTC8. Both graphs exclude the 129 correct primaries.

4 Statistical Calculations

4.1 Reliability Improvement

To know the reliability improvement the checkers give, we first have to compute the probability of failure on demand of the primaries:

$$PFD(\pi) = \sum_{x \in D} \omega_{\pi}(\pi, x) \cdot Q(x) \quad (6)$$

Figure 2(a) presents the distribution of the PFDs of the primaries in a histogram.

The probability of undetected failure of a primary, averaged over the checkers is:

$$PUFDA(\pi) = \sum_{x \in D} \sum_{\sigma \in R_{\sigma}} \omega_{\pi}(\pi, x) \cdot \omega_{\sigma}(\sigma, x, \sigma(x)) \cdot Q(x) \quad (7)$$

Figure 2(b) shows the distribution of the probability of undetected failure after applying RTC8. As can be expected, there is a significant shift to the left.

We now calculate the improvement of the primary PFD for various checkers for subsets of the primary programs:

$$I(PFD_{min}, PFD_{max}) = \frac{\sum_{\pi \in \hat{R}_{\pi}} PFD(\pi)}{\sum_{\pi \in \hat{R}_{\pi}} PUFDA(\pi)} \quad (8)$$

with $\hat{R}_{\pi} = \{\pi | PFD_{min} < PFD(\pi) \leq PFD_{max}\}$.

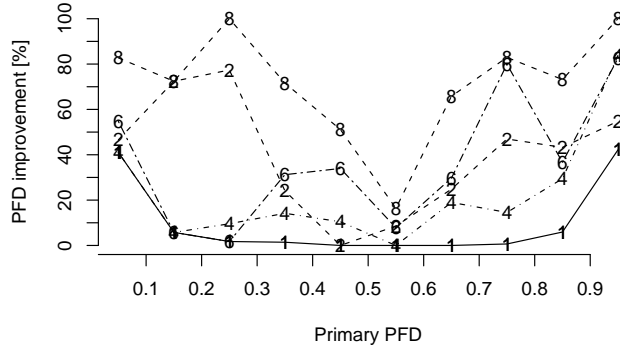


Fig. 3. The improvement of the probability of undetected failure in various PFD ranges for RTCs 1, 2, 4, 6 and 8 for a correct checker. The improvement in the range 0-0.1 excludes correct primaries.

We choose 10 subsets \hat{R}_π of the primary programs such that: $0 < PFD(\pi) \leq 0.1$, $0.1 < PFD(\pi) \leq 0.2$, and so on. Figure 3 shows the improvement of the probability of undetected failure in the various PFD ranges for RTCs 1, 2, 4, 6 and 8 for a correct checker (we do not show all, because this makes the figure unreadable; the other RTCs show similar erratic behaviour). There appears to be no obvious relation between the PFD of the primaries and the effectiveness of RTCs. Who would have expected that RTC6 would be very effective for reliable primaries?

Some RTCs are very effective, others are not. Some are effective for low primary PFDs, others for high. It is however not predictable which RTCs will be effective, since this depends on factors as the demand space and the programming faults made in the primary.

The graph gives rise to one possible conclusion: RTCs may still be effective for reliable primaries.

4.2 Effectiveness of the RTCs for Decreasing Average PUF D

We now investigate the effectiveness of RTCs as a function of the average PFD of primaries. To vary the PFD, we take the pool of 196 primaries and we one after another remove primaries with the highest PFD. The result is shown in Figure 4.

As observed in our earlier paper [9], the effectiveness of RTCs shows a rather unpredictable pattern. The PFD-improvement of RTC1-7 remains well below a factor three in almost the entire graph. RTC6 and 8 become infinity for low PUF Ds, but that is mainly caused by the fact that the number of primaries in this region becomes very small and the checkers manage to capture the faults in

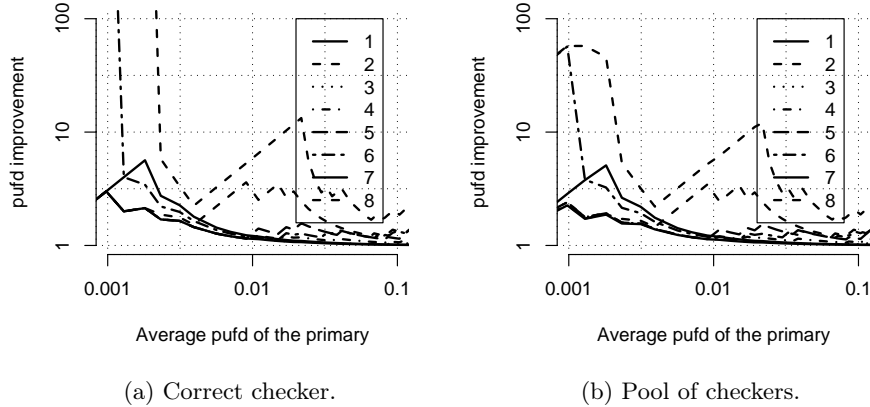


Fig. 4. The effectiveness of the eight RTCs as with decreasing average PFD of the pool of primaries. The PFD is lowered by subsequently removing the most unreliable programs. (a) For a correct checker; (b) averaged over the pool of checkers.

these few programs. The numbers computed in this region of the graph should be considered with care. RTC8 is the conjunction of RTC1-7, and reaches an average PFD improvement of about a factor three.

When we compare the PUFID of the correct checker with the average of the checkers, we can observe that there is little difference, except for highly reliable primaries. Here, the performance of the checkers is reduced, because of faults in some checkers.

There may be a turning point at which a checker’s effectiveness becomes questionable: when the ratio between false alarms and detection of primary failure becomes worse. Here we see a complex interplay between improving the quality of the primary and the checker. Improving the quality of the checker may have low priority, thus possibly resulting in poor specificity of the average checker.

5 RTCs vs. Multiple-Version Diversity

We now compare the effectiveness of RTCs with 1-out-of-2 diversity. We make a graph in the same way as Figure 4, except that we take two primaries from the pool instead of one.

We observe (see Figure 5) that 1-out-of-2 diversity becomes more effective with decreasing PFD of the pool of primaries from which the pair is selected. The reliability improvement ranges from a factor 25 to 100 for primary PFDs between 0.01 and 0.001. The effectiveness seems to reach a peak at a PFD of 0.001. (note that the opposite trend—effectiveness decreasing with decreasing mean PFD—is also possible, as proved by models and empirical results [6]). The

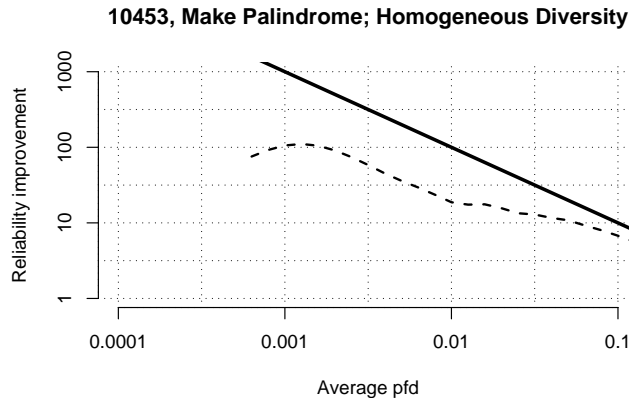


Fig. 5. Improvement of the PUF D of a pair of randomly chosen primaries, relative to a single version. The horizontal axis shows the average PUF D of the pool from which both primaries are selected. The vertical axis shows the PUF D improvement ($PUFD_A/PUFD_{AB}$). The diagonal represents the theoretical reliability improvement if the programs fail independently, i.e. $PUFD_{AB} = PUFD_A \cdot PUFD_B$.

improvement factor of most run-time checks remains fairly constant between 1 and 3 over this range, depending on the RTC. Only RTC8 reaches a factor of 10, when the PFD of the primaries is around 0.02. The improvement factor of using diversity is significantly higher than that of applying RTCs.

These results also confirm those in our earlier publication on the effectiveness of RTCs [9].

6 Conclusions

In this paper, we examined the effectiveness of Run-Time Checks without the assumption that these are fault-free. The effectiveness of the average checker appears to not deviate much from that of a perfect checker.

The effectiveness of the Run-Time Checks is comparable to that in our earlier study [9], a factor between one and three. We also confirmed the earlier result that multiple-version diversity is far more effective for decreasing the PUF D.

As yet, it seems not predictable which Run-Time Checks will be most effective, although in this study it is obvious that RTC8 will have the best coverage, simply because it is the conjunction of all the others. As a side effect, RTC8 also has the highest false alarm rate. Here again, it is hard to make a well-informed choice, because it is virtually impossible to predict the false alarm rate.

The results also show that Run-Time Checks may remain effective, also for the more reliable primaries. It may therefore be useful to keep the RTCs in primaries, also after extensive debugging. This however needs to be a trade-off with the possibility of raising false alarms.

We have to keep in mind that this research only considers one primary/Run-Time Checker combination, and that we can as yet not generalise, except perhaps for those observations that confirm those in our earlier publication on Run-Time Checks.

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 432–43, 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, L. Strigini, and M. Revilla. On the effectiveness of run-time checks. In B.A. Gran and R. Winter, editors, *Computer Safety, Reliability and Security, Proceedings of the 23rd international conference, Safecom 2005*, pages 151–64, Fredrikstad, Norway, September 2005.
10. 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.
11. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–49, 1997.