A FUNDAMENTAL PROBLEM IN IMPLEMENTING N-VERSION PROGRAMMING

Susan S. Brilliant
John C. Knight
Nancy G. Leveson

Computer Science Report No. TR-86-17
June 25, 1986

A FUNDAMENTAL PROBLEM IN IMPLEMENTING *N*-VERSION PROGRAMMING

Susan S. Brilliant

John C. Knight

Nancy G. Leveson

Affiliation Of Authors

Susan S. Brilliant John C. Knight

Department of Computer Science

University of Virginia

Charlottesville

Virginia, 22903

Nancy G. Leveson

Department of Computer Science

University of California

Irvine

California, 12345

Financial Acknowledgement

This work was supported in part by NASA under grant numbers NAG-1-242, NAG-1-605 and NAG-1-606, in part by NSF grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Company, and TRW.

Index Terms

Design diversity, fault tolerant software, multi-version programming, *N*-version programming, software reliability.

Address For Correspondence

Nancy G. Leveson

Department of Computer Science

University of California

Irvine

California, 92717

Abstract

We have identified a fundamental problem in the implementation of *N*-version programming. The problem, which we call the *Consistent Comparison Problem*, arises for applications in which decisions are based on the results of comparisons of finite-precision numbers. We show that, when versions make comparisons involving the results of finite-precision calculations, it is impossible to guarantee the consistency of their results. It is therefore possible that correct versions may arrive at different outputs for an application which does not apparently have multiple correct solutions. Thus an *N*-version system may be unable to reach a consensus even when none of its component versions fail.

I INTRODUCTION

Multi-version or *N*-version programming [2] has been proposed as a method of providing fault tolerance in software. The approach requires the separate, independent preparation of multiple (*i.e.* "*N*") versions of a piece of software for some application. These versions are executed in parallel in the application environment; each receives identical inputs and each produces its version of the required outputs. The outputs are collected by a voter and, in principle, they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority (if there is one) are assumed to be the correct output, and this is the output used by the system.

We have performed a large-scale experiment in *N*-version programming to test its axioms [3,4] and evaluate its performance [5]. As a result of performing this experiment, we have become aware of a fundamental problem in the implementation of *N*-version systems. The problem derives from the use of finite-precision arithmetic and the uncertainty that arises in making comparisons with finite-precision numbers. We refer to this as the *Consistent Comparison Problem.*

II THE CONSISTENT COMPARISON PROBLEM

When finite-precision arithmetic is used, the result of a sequence of computations depends on the order of the computations and the particular arithmetic algorithms used by the hardware. The issue that this raises for *N*-version systems is best illustrated by an example.

Any realistic application will require various comparisons to be made during the computation, and some of these will be based on parameters of the application as defined in the specification. For example, in a control system, the specification may require that the

actions of the system depend upon quantities such as temperatures or pressures that are measured by sensors. The value of temperature or pressure used by a program may be the result of extensive computation on sensor values. Sensors do not supply data in engineering units, and are often the subject of extensive processing under software control. Once these values are computed, however, the actions required at temperatures below 100°C may be different from actions required at temperatures above 100°C, and, similarly, the actions required may differ according to whether the pressure is above or below 15psi.

Now consider such an application implemented using a 3-version software system. Suppose that at some point within the computation, an intermediate quantity has to be compared with some application-specific constant C_1 in order to determine the required processing. As a result of the various limitations of finite-precision arithmetic it is quite likely that the three versions have slightly different values for the computed intermediate quantity, say R_1, R_2, and R_3. If the R_i are very close to C_1 then it possible that their relationships to C_1 are different. Suppose that R_1 and R_2 are less than C_1 and R_3 is greater than C_1. If the versions base their execution flow on these relationships, then two will follow one path and the third a different path. These differences might cause the third version to send to the voter a final output that differs from the other two.

It could be argued that this slight difference is irrelevant because at least two versions will agree, and, since the R_i are very close to C_1, either of the two possible outputs that would result from the use of the R_i would be satisfactory for the application. If only a single comparison is involved then this is correct. However, suppose that a second decision point is required by the application and that the constant involved is C_2. Only two versions will arrive at the decision point involving C_2 having made the same decision about C_1. Now suppose that the values produced by these two versions are on opposite sides of C_2. If the versions base their control flow on this comparison with C_2, then again their behavior will differ. The effect of the two comparisons, one with C_1 and one with C_2, is

that the three versions might obtain three different final outputs, all of which may well have been acceptable to the application, but a vote might not be possible. Despite the fact that this example is expressed in terms of comparison with C_1 and C_2, the order is irrelevant. In fact, since the versions were prepared independently, different orders are likely.

Although an application may seem to have only a single solution (unlike the square root problem, for example, that usually has two), the inconsistency in comparisons leads to the possibility that multiple correct outputs may be produced for a given input. This is an unexpected variant of the well-known "multiple correct results" problem in N-version programming [1]. The problem does not lie in the application itself, however, but in the specification. Specifications do not (and probably cannot) describe required results down to the bit level for every computation and every input to every computation. This level of detail is necessary, however, if the specification is to describe a function, *i.e.* one and only one output is valid for every input.

In summary, the issue is that multiple software versions might arrive at different conclusions because they take different paths based on comparisons that are required by the specification. The reason for the different paths is the inevitable difference in computed values within the versions due to finite-precision arithmetic and the diversity in the algorithms. In the example based on temperatures and pressures, the effect may be that the temperatures computed internally by the versions straddle 100°C and the computed pressures straddle 15psi. Our problem is to avoid this situation, a problem we term the Consistent Comparison Problem.

Consistent Comparison Problem

Suppose that each of N programs has computed a value. Assuming that the computed values differ by less than ϵ ($\epsilon > 0$) and that the programs do not communicate, the programs must obtain the same order relationship when comparing their computed value

with any given constant.

III NON-SOLUTIONS

A solution to the Consistent Comparison Problem requires that if comparison is needed with a constant C obtained from the specification, a given version must conclude that its value is, say, greater than C if and only if all correct versions will make the same decision given their individual roundoff and truncation errors. This must occur no matter in what order each version chooses to make the comparisons.

It might seem that using approximate rather than exact comparison within each version would solve the problem. An approximate comparison algorithm regards two numbers as equal if they differ by less than some tolerance δ [6]. If a computed value is to be compared with C, approximate comparison requires performing comparisons with numbers that differ from C by the tolerance, δ. For example, it can be concluded that the computed value is greater than C only if it is greater than $C + \delta$. Unfortunately, approximate comparison is not a solution because the problem immediately arises again, this time with $C + \delta$ rather than C. Two programs may compute values that are arbitrarily close to each other but have different order relationships with $C + \delta$, just as with C.

In fact, for two versions to obtain the same order relationship when comparing their computed values with a constant requires that the two computed values be identical. To solve the Consistent Comparison Problem, an algorithm is needed that can be applied independently by each correct version to transform its computed value to the same representation as all other correct versions. It is important to keep in mind that the algorithm must operate with a single value and no communication between versions to exchange values can occur since these are values produced by intermediate computation, not final outputs. It is not sufficient that the values be very close to each other since, no

matter how close they are, their relationships to the constant may still be different. However, it is not necessary that all precision be retained and so an approach using rounding or truncation would be satisfactory. Unfortunately, as the following theorem shows, there is no such algorithm, and so there is no solution to the Consistent Comparison Problem.

Theorem

Other than the trivial mapping to a predefined constant, no algorithm exists which, when applied independently to each of two n-bit integers that differ by less than 2^k, will map them to the same m-bit representation ($m + k \leqslant n$).

Proof

Suppose such an algorithm exists. Consider the case $k = 1$ and the set of values that can be represented in n bits, *i.e.* the integers in the range 0 to $2^n - 1$. The algorithm, when applied to each of two integers that differ by one, *i.e.* adjacent integers, will cause them to be reduced to the same m-bit representation. Thus, 0 and 1 must be mapped to the same representation. However, 1 must also be reduced to the same representation as 2, 2 to the same representation as 3, and in general i to the same representation as $(i + 1)$. Thus the algorithm must reduce all of the values to the same representation, *i.e.* the algorithm can only be the trivial mapping to any fixed binary representation, and the information content of the values is lost. A similar argument can be made for any k, and so the original assumption is wrong. ■

It is easy to be fooled into thinking that an approach such as rounding, for example, solves the problem. As the theorem shows, however, it does not. We note that this problem is not limited to comparison with constants. It arises with any comparison involving floating point values. For example, if two computed quantities F_1 and F_2 have to be compared, this is equivalent to comparing their difference with zero, a constant.

IV INADEQUATE AVOIDANCE TECHNIQUES

Exact Arithmetic

Since the difficulty seems to arise from finite-precision arithmetic, a tempting approach is to require some form of exact arithmetic in the versions. However, in addition to being impractical to use for most applications, exact arithmetic will not work in general because many algorithms are themselves capable of producing only approximate solutions. For example, consider the numerical solution of a differential equation which has no closed-form solution. The solution that is obtained is based on a discrete approximation to the equation and different algorithms will very likely use different discretizations. Different solutions may be obtained even though exact arithmetic is used in the calculations.

Cross-Check Points

It might be argued that a way to avoid inconsistency in comparisons would be to establish "cross-check" points [2] to force agreement among the versions on their floating point values before any comparisons are made that involve these values. At each cross-check point, each version would supply its computed result to a cross-check voter. The cross-check voter would select a single value (perhaps the median of the individual results) which would then be used by all of the versions in making the required comparisons.

Clearly, in principle, this avoids the problem, but various practical difficulties arise. First, if cross-check points are needed to avoid inconsistent comparison, they must be required in the specification. This means that the *order* of cross-check points must be specified also since otherwise the versions will deadlock. Requiring that the order of events in each program version be specified is a serious limitation on diversity in a situation where diversity is being sought.

In fact, in many applications, this requirement would limit diversity among versions to the point of absurdity. For example, the launch interceptor problem used in our N-version experiment [3] requires that the program determine whether fifteen "launch conditions" are satisfied. Each launch condition requires the determination of the existence of certain geometric relationships among subsets of up to 100 data points representing points in the plane. One of the launch conditions, for example, is satisfied if three data points can be contained within a circle of radius R, where R is a parameter of the application.

Some of the versions written for the experiment were structured to consider each of the fifteen launch conditions in turn, and, during the evaluation of each condition, the individual data points were considered. Others were structured to work through the sets of data points, considering for each the launch conditions that could be satisfied by the data points. If cross-check points were used to avoid inconsistent comparison, it would be necessary to require one of these two basic program structures. The order in which the launch conditions are considered and the order in which the individual data points are examined would have to be established as well. Cross-check points, as originally proposed, were not intended to be applied at this micro level.

In the application used in our experiment, each launch condition requires only the determination of the *existence* of a set of points satisfying a certain condition. It is possible, in the absence of cross-check points, that two programs might not even compute the same values. Each may find a set of satisfying points without considering the set found by the other version. Therefore the establishment of cross-check points may affect the values to be computed as well as the order in which they are computed.

Modified Voter

Since inconsistent comparison can lead to a situation in which a vote is not possible in an N-version system, it is tempting to consider modifying the voter. For example, since all N outputs are, in principle, acceptable to the application, one could be selected at random and used. However, an N-version system may also be unable to reach a consensus because a majority of the versions fail. In general, these two cases are indistinguishable and so modifying the voter in an attempt to deal with inconsistent comparison will also operate when the disagreement is due to failures. This approach is not satisfactory and could be very serious since the selected output could be completely wrong.

The action that should be taken in the event that no consensus is reached is application-dependent. For some applications, some course of action may exist that will bring the system into a safe, if unusable, shutdown state, and this may be a satisfactory response. For other applications, such as digital fly-by-wire aircraft, the assumption is made that the system will continue operating and there is no safe shutdown state or backup system. In these cases, a response might be to require the voter to select one of the outputs at random. If the disagreement is due to the Consistent Comparison Problem, then this action will be satisfactory since all N outputs would be acceptable. If the disagreement is due to version failures, there is a chance that an erroneous output will be selected and used. Failure of the system may well follow, but it would have happened anyway. For this class of applications, at least this treatment of no consensus avoids the Consistent Comparison problem.

V CONCLUSION

The Consistent Comparison Problem arises whenever a quantity used in a comparison is the product of inexact arithmetic. The problem occurs even when all software versions are

correct. It results from rounding errors, not software faults, and so an *N*-version system built from "perfect" versions may have a non-zero probability of failure.

We conclude that there is no solution to the Consistent Comparison Problem and that the avoidance techniques discussed in Section IV are inadequate. If no steps are taken to avoid it, the Consistent Comparison Problem may cause failures to occur. In general, there is no way of estimating the probability of such failures. The failure probability will depend heavily on the application and its implementation. Although this failure probability may be small, such causes of failure need to be taken into account in analyzing software for critical applications since those are precisely the ones that are, or will be, built using *N*-version programming.

ACKNOWLEDGEMENTS

We acknowledge helpful discussions with Bruce Chartres, Dana Richards, Paul Ammann, and James Ortega. This work was supported in part by NASA under grant numbers NAG-1-242, NAG-1-605 and NAG-1-606, in part by NSF grant DCR 8406532, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Company, and TRW.

REFERENCES

1. T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice*, Prentice Hall, Englewood Cliffs, NJ, 1981.

2. A. Avizienis and L. Chen, On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution, *Proceedings of COMPSAC 77*, November 1977, pp. 149-155.

3. J. C. Knight, N. G. Leveson and L. D. St. Jean, A Large Scale Experiment in N-Version Programming, *Digest FTCS-15: Fifteenth Annual International Conference on Fault Tolerant Computing*, Ann Arbor, Michigan, June 1985.

4. J. C. Knight and N. G. Leveson, An Experimental Evaluation of the Assumption of Independence in Multiversion Programming, *IEEE Transactions on Software Engineering*, January 1986, pp. 96-109.

5. J. C. Knight and N. G. Leveson, *Digest FTCS-16: Sixteenth Annual International Conference on Fault Tolerant Computing*, Vienna, Austria, June 1986.

6. D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, 1969.