Concise Papers

The Consistent Comparison Problem in *N*-Version Software

SUSAN S. BRILLIANT, JOHN C. KNIGHT, AND NANCY G. LEVESON

Abstract—**We have identified a difficulty 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 completely different outputs for an application that does not apparently have multiple correct solutions. If this problem is not dealt with explicitly, an *N*-version system may be unable to reach a consensus even when none of its component versions fails.**

Index Terms—**Design diversity, fault-tolerant software, multiversion programming, *N*-version programming, software reliability.**

I. Introduction

Multiversion or *N*-version programming [1] 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 and submitted to a decision algorithm that selects the output to be used by the system. If all of the outputs are expected to be the same, the decision algorithm might, for example, select the majority value, thereby tolerating faults in the minority. Where numeric outputs are expected to differ slightly because of computational differences between the versions, the decision algorithm might select the median value.

It is possible, of course, that no decision is possible because of multiple version failures. In most practical systems, this possibility is handled by designing a "backup" or failsafe system that takes over in case of failure to tolerate a fault (or faults). In simple systems, this may involve shutting down the computer system gracefully. In other more complex and critical applications, more sophisticated failsafe designs may be required. In any case, the *N*-version system has failed, resulting in at least a partial degradation in service and perhaps more serious consequences.

In the process of performing a large-scale experiment in *N*-version programming [2]–[4], a significant difficulty in the implementation of such systems has come to light that affects the ability of an *N*-version system to reach a consensus. The problem derives from the use of finite-precision arithmetic and the uncertainty that arises in making comparisons to finite-precision numbers. We refer to this problem as the *Consistent Comparison Problem*.

Manuscript received August 5, 1986; revised May 19, 1989. This work was supported in part by NASA under Grants 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.

S. S. Brilliant is with the Department of Mathematical Sciences, Virginia Commonwealth University, Richmond, VA 23284.

J. C. Knight is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903.

N. G. Leveson is with the Department of Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8930904.

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 temperature and pressure that are measured by sensors. The values of temperature and pressure used by a program may be the result of extensive computation on sensor measurements. Once these values are computed, however, the actions required at temperatures below, say, 100°C may be very 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 15 psi.

Now consider such an application implemented using a three-version software system. Suppose that at some point within the computation, an intermediate quantity has to be compared to 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 will 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 greater than C_1 and R_3 is less 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. The differences in the R_i cannot be eliminated with rounding, and might cause the third version to send to the decision algorithm a final output that differs radically 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 to 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 result by the decision algorithm might not be possible. The situation is shown graphically in Fig. 1. Despite the fact that this example is expressed in terms of comparison to 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, 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 [5]. The problem does not lie in the application itself, however, but in the specification.

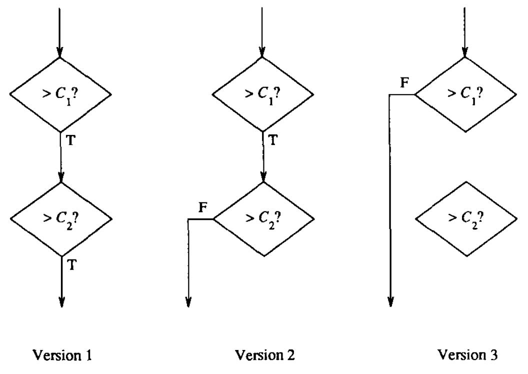

Fig. 1. Three-version disagreement.

Specifications do not (and probably cannot) describe required outputs in terms of the exact bit pattern required 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 15 psi. The Consistent Comparison Problem is to avoid this situation.

Consistent Comparison Problem: Suppose that N programs have been written to implement the same specification and each has computed a value. Assuming that the computed values differ by less than ϵ ($\epsilon > 0$) and that the programs do not communicate, each correct program must obtain the same order relationship when comparing its computed value to any given constant.

It is important to understand that the Consistent Comparison Problem is not related to the well-known problem in which the decision algorithm in an N-version system has to deal with slightly different numeric values due to finite-precision arithmetic. This latter problem has received considerable attention, and there have been a number of proposed solutions such as inexact voting [1]. The Consistent Comparison Problem derives from the need for versions to make isolated comparisons, and it can lead to output values that are completely different rather than values that differ merely by a small tolerance.

A solution to the Consistent Comparison Problem requires that all correct versions make the same decisions when performing comparisons given their individual computational differences. This must occur no matter in what order each version chooses to make the comparisons. In fact, to guarantee that two versions obtain the same order relationship when comparing their computed values to a constant requires that the two computed values be identical. It is not sufficient that the values computed by the various versions be very close to each other since, no matter how close they are, their relationships to the constant may still be different. 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. No communication between versions to exchange values can occur since the value to be used in the comparison by each version is the product of intermediate computation; it is not the final output. 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 ($m + k \leq n$).

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. □

We note that this problem is not limited to comparison to constants. It arises with any comparison involving computed numeric values. For example, if two computed quantities F_1 and F_2 have to be compared, this is equivalent to comparing their difference to zero, a constant.

III. Ineffective or Impractical Avoidance Techniques

Since no solution exists to the Consistent Comparison Problem, we turn our attention to the problem of avoiding its effects. In this section, we discuss avoidance techniques that are generally ineffective or impractical, although they might be used successfully in specific applications. In Section V, we discuss a more general approach to reducing the frequency of inconsistent comparisons, and in Section VI, we discuss the implications for real-time system design.

Approximate Comparison

An approach that has been proposed as a solution is to use approximate rather than exact comparison within each version. An approximate comparison algorithm regards two numbers as equal if they differ by less than some tolerance δ [6]. Approximate comparison is used frequently in order to make computer arithmetic adhere somewhat more closely to the normal rules of arithmetic.

If a computed value is to be compared to a constant C, approximate comparison requires performing comparisons to 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 Consistent Comparison 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.

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 that 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.

Random Selection

When an N-version system fails to reach a consensus because of inconsistent comparison, all N outputs are, in principle, acceptable to the application, and it might seem that the decision algorithm could be modified to select one of the outputs at random. However, an N-version system may also be unable to reach a consensus because, for example, a majority of the versions fail. These two cases are indistinguishable, and so modifying the decision algorithm in an attempt to deal with inconsistent comparison will also operate when the disagreement is due to failures. This approach is not always satisfactory and could be very dangerous since the selected output could lead to an unsafe course of action.

Confidence Signals

An approach using "confidence signals" has been suggested by Bishop [7]. Although any particular version cannot have access to the results of comparisons performed by other versions, it can determine that the values it used for comparison were sufficiently close that a problem might have arisen. As an example of how this might be done, the system could be modified to provide comparisons with three possible outcomes, rather than just two, i.e., "less than" if $R < C - \delta$, "greater than" if $R > C + \delta$, and "no confidence" if $C - \delta \leq R \leq C + \delta$. The third result would imply that an inconsistency could have occurred. In that case, the version might simply signal failure because it knows that its outputs might not be suitable for voting. The possibility then arises that each version will signal failure at some point, and the system may fail to arrive at an output because an insufficient number of versions (perhaps none) have generated results. Another approach is to require that each version pass to the final decision algorithm both its result and a confidence signal.

There are two ways that confidence signals might be used by the decision algorithm. First, if it finds a majority of the versions signaling a possible inconsistency in their results, the decision algorithm could choose any one of these results. The problem with this approach is that there is no way to distinguish between multiple correct results caused by the Consistent Comparison Problem and incorrect results where inconsistent comparison occurred along with a fault or faults in some other part of the algorithm(s). All fault tolerance is lost and the decision algorithm is reduced to random selection, the drawbacks of which were described previously.

A second use of confidence signals would be for the decision algorithm to ignore the outputs of all versions signaling a lack of confidence in their results since, as argued above, it will be unable to distinguish between the occurrence of an inconsistent comparison and a failed version. The versions indicating confidence in their outputs are then used normally. This may mean that fault tolerance is reduced or even eliminated. Note that if the computed value truly *is* close to the comparison parameter, it is likely that the only versions with confidence in their outputs will be those that are incorrect.

Cross-Check Points

It might be argued that a way to avoid inconsistency in comparisons would be to establish "cross-check" points [8] 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 decision algorithm. The cross-check decision algorithm 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. We note that this is not the purpose for which cross-check points were originally intended. They were introduced to allow recovery from software faults at the subsystem level, and were not intended to be applied at this micro level.

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 completely described in the requirements 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, and it may result in similar faults in different versions that are induced by the requirements specification. Ideally, no information about the design should be included in the requirements.

In fact, in many applications, this requirement would reduce diversity among versions to the point where the cost of producing multiple versions is not justified by the limited degree of diversity between the versions. For example, the launch interceptor problem used in our N-version experiment [2] requires that the program determine whether 15 "launch conditions" are satisifed. 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 15 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.

Finally, we note the inefficiency that cross-check points introduce. In highly reliable systems, it is important to take account of Byzantine faults [9]. If the different versions are executing on different processors, it will be necessary for the cross-check points to establish the values to be used by Byzantine agreement. In addition, since all versions must reach a cross-check point before a decision is possible, a decision cannot be made until the slowest version has arrived. Thus, the execution time of the system will be the sum of the execution times of the slowest version between each pair of cross-check points, which is likely to be slower than any of the individual versions and increase the execution time over N-version systems without cross-check points. For these reasons, using cross-check points may very well be impractical for systems with hard real-time limits.

There may be a class of applications that can use cross-check points to deal with the Consistent Comparison Problem. However, there is the serious potential problem of overspecification leading to reduced design diversity, and hence a reduced level of fault tolerance. This is unfortunate since achieving fault tolerance was the purpose of writing multiple versions in the first place. There may be an unacceptable increase in response time also.

IV. Reducing the Frequency of Inconsistent Comparison

Rounding and truncation are not solutions to the Consistent Comparison Problem, but they can be used to reduce its frequency of occurrence. For simplicity, we consider only truncation and integer arithmetic in this discussion. Rounding and truncation are essentially equivalent, and the extension to fixed- and floating-point quantities is straightforward. We show informally that the probability of inconsistent comparison is reduced as the number of digits truncated increases, and it is also reduced as the accuracy of the computed values increases.

Truncation to a specified precision after a computation but before performing comparisons seems promising because examples are easily constructed for which it provides consistent comparisons. However, counterexamples can always be found for any algorithm based on truncation.

Applying any truncation algorithm to the set of quantities that are representable by a given fixed length computer format divides the set of representable quantities into equivalence classes. To solve the Consistent Comparison Problem using truncation, it is necessary to ensure that the N independently computed values end up in the same equivalence class.

There is always a representable quantity, say V_1, that is the largest in any given equivalence class. The next representable number, say V_2, is the smallest in the adjacent equivalence class. If a real-world value of interest has to be computed (in effect, estimated) by a computer using finite-precision arithmetic, it is possible (actually, very likely) that the real-world value lies between two representable values such as V_1 and V_2. If two versions compute their estimate of this quantity to any precision whatsoever, they may compute the values V_1 and V_2, respectively, because they are estimating an unrepresentable quantity. These values will then be assigned to different equivalence classes by the rounding algorithm, and the Consistent Comparison Problem arises. This occurs even if the specification requires that the computed result be in error by no more than some tolerance. Also, changing the starting point for rounding or the number of digits used will not solve the problem in general.

In practice, the probability of inconsistent comparison depends on a number of factors that are application specific. However, in general, the probability of occurrence decreases as the number of digits truncated increases. This occurs because the equivalence classes generated by truncation increase in size as the number of digits truncated increases, and so the chances of two computed values being in the same equivalence class also increases. Similarly, as the required accuracy increases, the difference between the values computed by two different versions decreases, and once again, this reduces the chances of the two values being in different equivalence classes.

The effects of the Consistent Comparison Problem must be included in the reliability analysis of any application using N-version programming. The specific probability of its occurrence can only be computed when details of the application are known. However, truncation (or, equivalently, rounding) might be used deliberately to reduce the effects of the problem to acceptable levels.

V. Implications for System Design

The immediate effect of inconsistent comparison is that a consensus might not be possible. The extent of the damage varies with the application and has a substantial impact on the effectiveness of measures designed to cope with the damage. The major characteristic that needs to be considered is whether or not the application has state information that is maintained from frame to frame, i.e., "history".

Some simple control systems have no history. They compute their outputs for any given frame solely from constants and the inputs for that frame. If the inputs are changing, it is extremely unlikely that a situation in which no consensus is possible would last for more than a short time. After a brief period, the inputs will change and leave the region of difficulty. Subsequent comparisons will be consistent among the versions because the values used for comparison will be sufficiently different. The effects of the Consistent Comparison Problem in such systems are transient. How the system should recover (and even *if* it can recover) from such transient failures is application-dependent, but it may be possible to treat the situation as a single-cycle failure by just not producing any results for this particular cycle or providing results consistent with the last cycle.

For systems with history, the situation is much more complex. Since such systems maintain internal state information over time, an unfortunate consequence of inconsistent comparison is that failure to reach a consensus might be accompanied by differences in the internal state information among the versions. The duration of internal state differences varies among applications.

In some applications, the state information is revised as time passes, and once the inputs have changed so that comparisons are again consistent, the versions may revise their states to be consistent as well. In this case, the entire system is once again consistent and operation can proceed safely. An example of this type of application is an avionics system in which the mode of flight is maintained as internal state information. If this flight mode is determined by height above ground, for example, then if a measurement is taken that is close to the value at which the mode is changed, different versions might reach different conclusions about which mode to enter, along with having different values for various other related version-specific variables. However, it is likely that this situation will be corrected rapidly if the versions continue to monitor the height sensor. We will refer to such systems as having *convergent* states.

For systems having convergent states, inconsistent comparisons may cause a temporary discrepancy among the internal states of the versions that could last for more than one cycle. A confidence signal approach in which each version maintains confidence information as part of its state might be used in this case. This approach requires fairly extensive modifications and additions to the system structure. Not only would the individual versions need to include confidence information on every comparison and each component of its internal state, but outputs would have to be supplemented by the required confidence signal and the decision algorithm would have to be modified to take these signals into account. If a part of its state information is based on a comparison that is subject to doubt, then the version must indicate "no confidence" in all of the results it sends to the decision algorithm until the state is reevaluated. The decision algorithm will then treat this as a failed component until the version indicates confidence in its results. The time required to reevaluate the state is application-dependent and may be unacceptably long in many cases, requiring the use of failsafe and backup procedures.

Recovery [10] is not a practical solution here. Recovery involves modifying the internal states of the incorrect versions based on that of a correct version. However, in this case, the lack of information about which version is correct is the whole problem. Furthermore, the recovery process may require restricting the algorithmic and data-structure diversity between versions.

For some applications, state information is determined and then never reevaluated. An example of this is sensor processing where one version may determine that a sensor has failed and subsequently ignore it. Other versions may not make the same decision at the same point in time, and depending on subsequent sensor behavior, may never conclude that the sensor has failed. In this case, although the inputs change, versions may continue to arrive at different outputs long after comparisons become consistent because the sets of state information maintained by the individual versions are not the same. We will refer to these systems as having *nonconvergent* states.

Once the versions in a system with nonconvergent states acquire different states, the inconsistency may persist indefinitely. Although no version has failed, the versions may continue to produce different outputs, and in the worst case, the N-version system may never again reach a consensus on a decision. We see no simple avoidance technique that can be used in this case. Recovery is again not a practical or safe solution. The only practical approach in systems of this type seems to be to design the system to be failsafe.

VI. 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 nonzero probability of being unable to reach a consensus.

This paper has presented some possible solutions for particular types of simple applications, but has also shown that no general, practical solution to the Consistent Comparison Problem exists. This result is important because if no steps are taken to avoid it, the Consistent Comparison Problem may cause failures to occur that would not have occurred in nonfault-tolerant systems. Although the authors have observed the phenomenon in several different multiversion programming experiments, there is, in general, 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 estimating the reliability of N-version software, particularly for critical applications.

ACKNOWLEDGMENT

We are pleased to acknowledge helpful discussions about the Consistent Comparison Problem with P. Ammann, P. Bishop, B. Chartres, J. Cordy, T. Hull, J. Ortega, B. Randell, and D. Richards. We are extremely grateful to J. Goldberg and the referees for their careful and constructive comments about earlier drafts of this paper.

REFERENCES

[1] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Dig. FTCS-8: 8th Annu. Int. Symp. Fault-Tolerant Comput.*, Toulouse, France, June 1978, pp. 3-9.

[2] J. C. Knight, N. G. Leveson, and L. D. St. Jean, "A large scale experiment in N-version programming," in *Dig. FTCS-15: 15th Annu. Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 1985, pp. 135-139.

[3] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Software Eng.*, pp. 96-109, Jan. 1986.

[4] ——, "An empirical study of failure probabilities in multi-version software," in *Dig. FTCS-16: 16th Annu. Int. Symp. Fault-Tolerant Comput.*, Vienna, Austria, July 1986, pp. 165-170.

[5] T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice.* Englewood Cliffs, NJ: Prentice-Hall International, 1981.

[6] D. E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.* Reading, MA: Addison-Wesley, 1969.

[7] P. Bishop, personal communication, June 1986.

[8] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," in *Proc. COMPSAC'77*, Nov. 1977, pp. 149-155.

[9] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals problem," *ACM Trans. Programming Languages Syst.*, pp. 382-401, July 1982.

[10] K. S. Tso and A. Avizienis, "Community error recovery in N-version software: A design study with experimentation," in *Dig. FTCS-17: 17th Annu. Int. Symp. Fault-Tolerant Comput.*, Pittsburgh, PA, July 1987.

Statistical Inference for General-Order-Statistics and Nonhomogeneous-Poisson-Process Software Reliability Models

HARRY JOE

***Abstract*—There are many software reliability models that are based on the times of occurrences of errors in the debugging of software. It is shown that it is possible to do (asymptotic) likelihood inference for software reliability models based on order statistics or nonhomogeneous Poisson processes, with (asymptotic) confidence levels for interval estimates of parameters. In particular, interval estimates from these models are obtained for the conditional failure rate of the software given the data from the debugging process. The data can be grouped or ungrouped. For someone making a decision about when to market a software, the conditional failure rate is an important parameter. The use of the interval estimates is demonstrated on two data sets that have appeared in the literature.**

Manuscript received July 15, 1988; revised April 18, 1989. Recommended by F. B. Bastani. This work was supported by a UBC Killam Research Fellowship and a grant from the National Science and Engineering Research Council of Canada.

The author was on leave at University College, London, England. He is with the Department of Statistics, University of British Columbia, Vancouver, B.C. V6T 1W5, Canada.

IEEE Log Number 8930509.

***Index Terms*—Failure rate of a process, maximum likelihood, nonhomogeneous Poisson process, order statistic, profile likelihood, software reliability model.**

I. INTRODUCTION

Many software reliability models for times to occurrences of bugs or errors during the debugging process have been proposed (see Miller [15] and Abdel-Ghaly *et al.* [1] and references therein), but there is comparatively less work on statistical inferences from these models. Some authors have mentioned the possibility of using maximum likelihood estimation and some have mentioned that maximum likelihood estimates behave poorly for some models, but this has not been qualified. One purpose of this paper is to make clear when maximum likelihood estimation can be good. Standard asymptotic likelihood theory cannot be applied to get approximate confidence limits of interval estimates of parameters because the setting here is not one of independently and identically distributed (i.i.d.) random variables. However, some authors, for example, Forman and Singpurwalla [4] and Yamada and Osaki [20], have misused this theory to get confidence intervals of estimates for software reliability models; while their interval estimates may be sensible in some cases, they have not justified when the confidence level is meaningful. This paper shows that there is an asymptotic sense (debugging process sufficiently long, number of original errors sufficiently large) in which asymptotic normality results (and hence asymptotic confidence levels) hold for the classes of software reliability models based on order statistics and on nonhomogeneous Poisson processes (NHPP). The asymptotics is similar to that of Lindsay and Roeder [13]. Both grouped and ungrouped data can be dealt with.

One important quantity, especially for someone who must decide on when to begin marketing a software, that one might want inferences on is the conditional failure rate of the software given the results of the debugging process so far. The use of software reliability models is a means of obtaining inferences on the conditional failure rate or other similar quantities.

In Section II, the conditional failure rate is derived for several software reliability models, including the general order statistics (GOS) models and the NHPP models. In Section III, asymptotic results are stated for these classes of models. These models essentially assume that all bugs are of a similar type, but there are data sets in the literature for which these models are applicable and the asymptotic results can be used. The methodology can be applied to more general situations such as described by Adams [2]. In Section IV, some computational details concerning likelihoods are discussed, and the results of a small simulation study are given to give an indication of what may be considered "asymptotic" in practice. In Section V, the results are applied to get interval estimates of the conditional failure rate for two data sets, one from Musa with ungrouped data and one from Forman and Singpurwalla [4] with grouped data.

II. CONDITIONAL FAILURE RATE

In this section, we define the conditional failure rate of a process and determine it for Ross' [18] model, GOS models, and NHPP models. Note that in this paper, time will usually mean (cumulative) execution time. Suppose that in the interval of time $[0, T]$, there have been n failures (or discovery of errors) in the software at the times $0 < Z_1 < \cdots < Z_n < T$. Then the conditional failure