Myths and Methods:
Is There a Scientific Basis for Y2K Inspections?

David Lorge Parnas, P.Eng.
NSERC/Bell Industrial Research Chair in Software Engineering
Director of the Software Engineering Programme
DEPARTMENT OF COMPUTING AND SOFTWARE
Faculty of Engineering
McMaster University, Hamilton, Ontario, Canada - L8S 4L7

ABSTRACT

Although it is possible to use scientifically based mathematical models in the analysis of software, most programmers rely on their intuitive understanding instead. With complex programs, our intuition is often inadequate and we overlook serious faults. Although most organisations have introduced a systematic management process for Y2K software inspection and repairs, the actual analysis of the software relies on intuitive "eyeballing" of the code. Intuition has given rise to some folklore about the Y2K problem that has no scientific basis. We need to base our analysis on sound science and mathematics. Moreover, sound models suggest procedures that are more systematic and more trustworthy than the intuitive ones. This paper will discuss both some of the unscientific myths and sound inspection methods. If we base our program analysis on unsound methods, we are "building on sand" and what we do will fail.

1 Introduction

Although the first reports of the Y2K problem in computer software were viewed as "alarmist", subsequent events have proven that the problems are quite real. Simple experiments, advancing the clock and letting a system run, have shown that many systems will halt or produce unexpected results. Today, governments and companies around the world are investing in a systematic examination of their software with the intent of finding and correcting the Y2K bugs.

2 Why did it happen? - the usual explanation

When most of these programs were first written, computer memories were much smaller, and much more expensive, than they are now. Programs were not "portable" and were discarded with each new machine. It seemed silly to waste memory on those redundant "19"s. Many of those old programs are still being used and the new ones had to be compatible with them. In other words, programmers were making a necessary compromise and did the best that could be done given the memory limitations.

3 Why did it happen? - the real explanation

The usual explanation of the problem does not hold up under scrutiny.

- Two digits representations do limit us to a century, but that century need not begin with the year 1900. Most of this code was written well have 1950. With very little effort, and no extra memory, the programs could have been written to work from 1950 to 2049 or 1960 to 2059.
- Most computers store digits in 8 bit bytes. A single 8 bit byte can be used to provide a range of 256 years. With a little bit of effort, but no extra memory, programs could have been written so that they would be useful until the year 2200.

- For nearly 3 decades we have known how to organise programs so that data representation decisions, such as the decision to store only 2 digits of a date, are easily revised. Techniques for doing this were discussed at conferences and published in textbooks and papers. They require no extra memory. Most programmers were either unaware of these techniques or did not choose to use them.

The real cause of the Y2K bug is that most programmers have not received an education appropriate to the work that they do. Bluntly put, many of the people who wrote those programs, and many who are still working on them, are not competent for the jobs they have.

4 The cause for concern

A great deal of effort is being invested in finding and correcting Y2K problems but many software experts remain very concerned. Their concern is based on simple facts and logic.

- Most software products are released to their users with many faults (euphemistically called "bugs").
- When a bug is reported and a "fix" or "patch" is sent to users it is very common to find that the program is still not correct. By some estimates, more than half of all "fixes" don't adequately correct the problem and another "fix" is needed.
- In a few studies done many years ago, it was found that sometimes each "fix" introduces more errors than it "repaired". In other words, as you "correct" some software systems they get worse, not better.
- We are now working with software that was written long ago by people who are no longer with the organisation. In some cases we are looking at software that is so old that the programming language is no longer being used by today's programmers.
- We have absolutely no reason to expect that our success rate with Y2K corrections will be higher than what we find when correcting current software. *Au contraire.* In repairing "legacy" software, we are working with a handicap - we are often working with software with which we are no longer familiar.
- There are some organisations that do not seem to be taking the problem seriously.

Given the above facts, any prudent person would expect that there will be problems as programs start to deal with dates for the year 2000. These problems will all happen around the same time and programmers will be very busy. Many programmers will not have access to some of the computer-based tools that they are used to using. It is unlikely that things will go smoothly.

5 Y2K Myths

In discussing this problem with a variety of programmers and engineers, I have heard a few statements that strike me as "urban folklore". These statements are false, but I still see them being used in arguments that justify decisions.

Myth 1: "Y2K" is a software problem. If the hardware is not programmable, there is no problem."

Any digital system, including those that are not programmable, can represent dates in a variety of ways and may use a representation or logic that does not work at the turn of a century. All digital hardware that may store dates must be reviewed and tested.

Myth 2: If the system does not have a real-time clock, there is no Y2K problem.

If a system maintains its own date and time information using an internal clock, it is an obvious candidate for a Y2K problem. However, systems that get date/time information from external sources can also have problems. It really doesn't matter whether the source of date data is internal or external; systems that process dates, even if they do nothing more than print out a date typed in by an operator, could have Y2K problems and must be inspected.

Myth 3: If the system does not have a battery to maintain date/time during a power outage there can be no Y2K problem.

Systems that are designed to retain dates even when power is off are also obvious candidates for a Y2K inspection, but a system that requires that the date be supplied again when it is restarted is also a potential problem. Moreover, some systems can retain dates if they do not have battery backup. For example, some pocket devices can retain data in volatile memory while batteries are being replaced (quickly). It is also possible that some systems were designed on the assumption of a highly reliable power supply - something that most of us would assume was available at a power generation station.

Myth 4: If the software does not process dates, there can be no Y2K problem.

Software systems that do not use dates may be dependent on other systems that do process dates. This may mean that a system that is itself Y2K compliant may fail because one of the systems with which it exchanges information fails. A system that sends data to a failed system may fail because the communications protocol calls for an acknowledgement of receipt.

Myth 5: Software that does not need to process dates is "immune" to Y2K problems.

Even software that does not need to process dates may actually do so. It is quite common to reuse software in order to save on development costs. The old software may include segments of code that are not needed for the new application. In today's networks, software may acquire information from other systems on the network and may fail when those systems fail. Software that does not need date information itself, may receive it and then relay it to other systems. We cannot conclude that a system is immune on the basis of requirements alone; one must examine the programs.

Myth 6: Only systems identified as safety systems need be of concern to safety authorities.

Generally, systems in nuclear power plants are identified as safety-critical if their failure may lead to difficulties in the short-term, i.e. they may lead to loss of regulation. However, in the case of Y2K we must also be prepared for long outages. Systems that are needed for maintenance or record-keeping can fail for an hour or two without serious effects, but if such a system is not available for several days or a week, safety hazards may go undetected and important safety procedures may be forgotten.

Myth 7: Systems can be tested one-at-a-time by specialised teams.

It is common to find heterogeneous networks, i.e. networks that contain computers of various types from a variety of manufacturers. Programmers and engineers tend to be experts in one system or type of system and not know much about others. This leads managers to assemble Y2K

teams that specialise in individual computer systems. Unfortunately, it is possible to repair Y2K problems on two communicating computer systems in such a way that each one works on its own but they will fail when used together in the year 2000. Any set of communicating systems must be analysed as a whole (cf. section 6.1) with teams that include experts in each of the systems.

Myth 8: If no date dependant data flows in or out of a system while it is running, there is no problem.

There are systems that have no internal clock and no interface that allows real-time date-dependant information to be exchanged while they are running. It seems obvious that such systems are not going to have date-dependant behaviour. However, date-dependant information may be supplied to the software by an operator carrying an “eprom”, disk, or some other data carrier. All information sources and paths must be taken into account, not just the run-time information paths.

Myth 9: Date stamps in files don’t matter.

Many software development organisations follow the (good) practice of including the date of last change in a file. When inspectors find these “date stamps” while looking for Y2K problems, they tend to ignore them. In most cases, this is justified but we must remember that the whole file is available to the software that processes it and a subtle bug may occur if an unexpected date such as “00” is present. For example, the date stamp may be used by archiving software. It is necessary to show that this information is not used; we cannot assume that to be the case.

Myth 10:Planned testing, using “critical dates” is adequate.

In the Y2K literature, we find lists of “critical” dates, dates where system failure is most likely. These are dates like 1.1.2000, 29.2.2000 or 9.9.1999. There are many known programming errors that will lead to errors on these dates. Unfortunately, we cannot be sure that a program that does not fail on these special dates will not fail on other dates. Prudence requires extensive additional testing using a statistically valid random selection of dates.

Myth 11:You can rely on keyword scan lists

Many organisations have assembled a list of keywords that are related to dates and may be present at date-sensitive parts of the software. These lists are exchanged and keep growing. Some organisations seem to think that they can find all potential points of failure by scanning for words like “date”, “year”, or “time”. Unfortunately, (1) it is well known that the behaviour of a program will not change if one identifier is systematically replaced by another that was not previously used in the program, and (2) programmers are notoriously capricious in their choice of identifiers and their choice of in-line comments. A programmer may decide that it would be fun to use “jaar” instead of “year” or even believe that the latter should be spelled “yir”. Scanning for key words is a useful supplementary confidence-building technique but we cannot assume that all date processing parts in the program will be found.

Myth 12:The original safety cases remain valid.

When a new plant is given a license or a modification is approved, a “safety case” is presented to show that the risk of failure is acceptably low. Usually the “safety case” is based on specific

assumptions about the likelihood of simultaneous failures. Unfortunately, the Y2K failures are very likely to occur simultaneously or in a short period of time. The likelihood of simultaneous Y2K failures is much higher than the likelihood of simultaneous failures of other types. All safety cases for existing equipment should be re-examined to consider the effects of simultaneous failures that were considered unlikely in when the analysis was done.

6 Systematic Methods

In this section, I will mention 3 methods of analysis that have been developed in the Computer Science community over the last decades. All three have been proven practical in practice and should be applied in the analysis of critical programs.

6.1 Decomposition based on data flow diagrams

Plants and businesses usually use many computers, often communicating with each other in a variety of ways. If we wish to do a thorough job of analysis, we must use the "divide and conquer" philosophy, i.e., divide the whole enterprise into a number of smaller subsystems and analyse each separately. However, this cannot be done arbitrarily; decomposition will only be helpful if the subsystems that are identified are relatively independent.

It is impossible to confirm the adequacy of an analysis, or the correctness of a proposed "fix", in a system unless we know the information-flow between the system in question and other systems. For example, the decision to correct a problem by shifting the internal date to an earlier date is sound if, but only if, there is no communication of date dependent information between that system and other systems.

A technique that has been used successfully for many years in the information systems industry is the data-flow diagram. Overview data-flow diagrams show all sources and recipients of information, all places where information is stored temporarily, output to a user/operator as well as all reports and forms that are produced. More detailed diagrams will include specific information about the information flowing between systems.

Even in the crude form that is used in the data processing industry, these diagrams provide a lot of insight into a system. The diagrams can be made more precise by providing further details about the information that "flows along the arrows" using mathematical notation. In the case of Y2K analysis, one must know whether or not the information flowing is date dependant. It is essential to include all information-flow, not just that information flowing along wires while the system is running. Information provided by an operator, carried in on storage media, and stored in a ROM must also be taken into account. Information-flow can be very indirect. For example, if system A controls the flow of a coolant and system B measures the flow of that coolant, information is flowing from A to B.

Information-flow diagram should be used when decomposing a set of computer systems into smaller systems that can be studied nearly independently. This is often done incorrectly, e.g. placing systems that are physically close in the same subsystem, grouping equipment by manufacturer, or grouping systems by the responsible department. For most safety analysis, it is the information-flow that matters, not the physical location, or the administrative responsibility.

In drawing information-flow diagrams, it is tempting to try to give only the "big picture", i.e. show only the high data rate flows or those that are considered most important. Unfortunately, for computer systems it is literally true that "every bit counts"; the whole system may fail because

one bit is wrong. Even small amounts of information that are transferred very infrequently must be shown.

A classic discussion of diagramming techniques is contained in [11].

6.2 Slicing

With large programs we have trouble understanding everything and so must search a program looking for the relevant sections. Unfortunately, it is very easy to miss a relevant section during this search because we are not able to spend much time on the parts that we consider irrelevant. It would be nice if there were a way to reduce a program to a smaller program that contained only the relevant sections.

Slicing is a technique that was introduced in to the Computer Science literature in 1981 by Dr. Mark Weiser. In slicing, one identifies certain variables of interest and then finds the parts of a program that use or modify those variables. The variables used or modified in those statements are then added to the set of variables of interest and the search continues. Eventually, the search terminates and one can identify all lines in the program that have any possible effect on the variable of interest. The set of such lines is called a “slice” and is often much smaller than the original program.

There are slicing tools available for specific languages. Even when you have no tool, slicing provides a systematic and trustworthy method that can be applied by hand.

There are programming techniques that make “slicing” much more difficult. It is nearly impossible in machine-language programs that use indexing or indirect referencing. The use of pointers, or the extensive use of arrays in “clever” ways can also make slicing ineffective.

There is an extensive literature about “slicing” but a good start can be obtained by going back to the older papers [8, 9, 10].

6.3 Program function table analysis

When analysing a complex program, it is very helpful to summarise the behaviour of sections of code so that one can more easily understand the behaviour of the enclosing or invoking sections. Any executable program can be represented by a mathematical function. However, the functions are more difficult to describe than those encountered in analog systems. We have found a tabular format allows these functions to be written and understood by the average engineer. This section gives an informal introduction to program-function tables based on the example in Figure 1.

	Array B contains x [a].	Array B does not contain x [b]	
j’ |	B[j’] = x	***true***	
present’=	**true**	**false**	$\wedge$ NC(x, B)

Figure 1: Specification of a search program.

a. “Array B contains x” is be defined as $(\exists\ i, B[i] = x)$

b. “Array B does not contain x” is defined as $(\forall\ i, ((1 \le i \le N) \Rightarrow B[i] \ne x))$

The table in Figure 1 describes a program that must search an array, B, to find an element whose value is that of the variable x. The program is to determine the value of two program variables called “j” and “present”. The variable j, (presumed to be an integer variable), is to record the index of one element of A whose value is the value of the variable x (if one exists). The variable called “present”, presumed to be a boolean variable, is to indicate whether or not the desired value could be found in B. If B does not contain the value sought, the value of j is allowed to be any integer value.

The header at the top of the table in Figure 1 shows that two situations must be distinguished. The first column of the main grid (under the first element of the top header) describes the case where the value sought can be found in the array. The second column describes what the program must do if the value cannot be found.

Each row in the main grid of a program-function table corresponds to a program variable and describes the value that this variable must have upon termination. The header to the left of the table identifies the variable whose value is described in each row and also indicates how that variable will be described. A “|” in the vertical header indicates that the variable’s value must satisfy a condition given in the appropriate cell in the main grid. When “=” appears instead of “|”, the grid elements in that row must be expressions that will evaluate to the value of the variable. The values of predicates are represented by “***true***” and “***false***”. The symbols “**true** and “**false**” represent the possible values of the boolean variable.

The condition “NC(x, B)” is ***true*** if x and B are not changed by the program.

There is nothing that can be said with such a table that cannot be said using equivalent conventional boolean expressions. However, using the table, one can select the row and column of interest and need not understand the whole expression in order to find out what must happen in a specific case. The number of characters that appear in the tabular expression is usually smaller than the number of characters in the equivalent conventional expression because in the latter some of the expressions that appear once in one of the headers would have to be repeated several times in the conventional expression. On larger tables involving many cases, many variables, and longer identifiers, the advantages of the table format are more dramatic. More extensive discussions of these tables can be found in [4,7].

Tables of this sort were used in the inspection of safety-critical software for the Darlington Nuclear Power Generation Station as described in [2, 6, 1]. The theory behind their use is given in [3, 5]. They are useful whenever a very careful and disciplined inspection of software is required.

7 Conclusions

Classically educated engineers know that they must apply science and mathematics in their work. Unfortunately, most programmers have not received an appropriate professional education and build and inspect programs in a purely intuitive manner. This has resulted in a never ending “software crisis” and the Y2K problem that we have today. If we want to solve that problem, we must move away from myths and folklore and apply scientifically sound software engineering techniques. A sound Y2K analysis process will include (1) accurate data flow diagrams, (2) program slicing for long programs that are not well structured, and (3) program function tables where precise descriptions of program behaviour are needed.

8 Acknowledgements

The support of the Atomic Energy Control Board of Canada and helpful discussions with Namir Anani are gratefully acknowledged.

9 References

1. Archinoff, G.H., Hohendorf, R.J., Wassyng A., Quigley, B., Borsch, M.R., "Verification of the Shutdown System Software at the Darlington Nuclear Generating Station", International Conference on Control & Instrumentation in Nuclear Installations, Glasgow, May 1990.
2. Parnas, D.L., Asmis, G. J. K., Madey, J., "Assessment of Safety-Critical Software in Nuclear Power Plants", *Nuclear Safety*, vol. 32, no. 2, April-June 1991, pp. 189-198.
3. Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering" published in *Science of Computer Programming* (Elsevier) vol. 25, number 1, October 1995, pp 41-61.
4. Parnas, D.L., "Tabular Representation of Relations", CRL Report 260, Communications Research Laboratory, McMaster University, October 1992, 17 pgs.
5. Parnas, D.L. "Mathematical Descriptions and Specification of Software", *Proceedings of IFIP World Congress 1994, Volume I"* August 1994, pp. 354 - 359.
6. Parnas, D.L. "Inspection of Safety Critical Software using Function Tables", *Proceedings of IFIP World Congress 1994, Volume III"* August 1994, pp. 270 - 277.
7. Parnas, D.L., Madey, J., Iglewski, M., "Precise Documentation of Well-Structured Programs", *IEEE Transactions on Software Engineering*, Vol. 20, No.12, December 1994, pp. 948 - 976.
8. Weiser, M., "Program Slicing", Proceedings of the 5th International Conference on Software Engineering, March 1981.
9. Weiser, M., "Programmers Use Slices When Debugging", Communications of the ACM, 25(7), pp. 446-52, July 1982.
10. Weiser, M., Lyle, J., "Experiments on Slicing-Based Debugging Aids", Empirical Studies of Programmers, pp. 187-197, 1986.
11. Yourden, E., Constantine, L., "Structured Design: Fundamentals of a Discipline of Computer Program and System Design - 2nd edition, September 1986, Prentice Hall.