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 - L85 4L

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 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.

Introduction

First reports of the Y2K problem viewed as alarmist".

We know now that the problems are quite real.

Governments and companies are all investing in finding and correcting the Y2K bugs.

I want to discuss the following issues:

- Why did it happen?
- Why should we worry?
- What myths are common about this problem?
- What can we do to make the inspection/repair process more effective?

Why did it happen? - the usual explanation

Programs were written when computer memories were much smaller.

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 New ones must to be compatible with old.

Programmers were making a necessary compromise and did the best that could be done given the memory limitations.

Why did it happen? - the real explanation

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

- With no extra memory, the programs could have been written to work from 1960 to 2059.
- A single ~ bit byte can be used to provide a range of 256 years. With no extra memory. programs could have been 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. Programmers were either unaware of these techniques or did not chose 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.

Many of the people who wrote those programs, and many who are still working on them. are not competent for the jobs they have.

The Cause For Concern

Many software experts are concerned because:

•Most software products are released with "bugs".

•By some estimates, more than half of all "fixes" don't adequately correct the problem and another "fix" is needed.

•Sometimes each "fix" introduces more errors than it "repaired"; as you 'correct" some software, it gets worse, not better.

•We are now working with software that was written long ago by people who are gone.

•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 no reason to expect that our success rate with Y2K corrections will be higher than usual. Au *contraire*.

•There are some organisations that do not seem to be taking the problem seriously.

Any Prudent person would expect problems.

These problems will all happen at the same time.

It is unlikely that things will go smoothly.

Y2K Myths

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, 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.

Y2K Myths

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

- Systems with an internal clock are obviously problem.
- Systems that get date/time information from external sources can also have problems.
- It doesn't matter whether the source of date data is internal or external;
- Systems that do nothing more than process a date typed in by an operator, could have Y2K problems and must be inspected.

Y2K Myths

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

- A system that requires that the date be supplied again when it is restarted is potential problem.

- Moreover, some systems can retain dates for limited periods of time if they do not have battery backup.

- Some systems were designed on the assumption of a highly reliable power supply. They may still have Y)K related errors.

Y2K Myths

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

- A system that is itself Y2K compliant may fail because one of the systems with which it exchanges information fails.

- A system that sends system to a failed system may fail because the communications protocol calls for an acknowledgement of receipt.

Y2K Myths

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

- 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 relay it to other systems.

We cannot conclude that a system is immune on the basis of requirements alone: one must examine the programs.

Y2K Myths

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

- Systems are identified as safety-critical on the basis of short-term behaviour.
- For Y2K we must be prepared for long outages.
- Systems that are needed for maintenance or record-keeping can fail for short periods without problems.
- If maintenance systems are not available for several days or a week, safety hazards may go undetected and important safety procedures may be forgotten.

Y2K Myths

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

- Heterogeneous networks, are common.
- Programmers tend to be experts in one system or type of system and not know much about others.
- Managers to assemble Y2K teams that specialise in individual computer systems.
- 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 with teams that include experts in each system.

Y2K Myths

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

- It is often assumed that systems that have no internal clock and no interface that allows real-time date-dependent information to be exchanged while they are running are not subject to Y2K problems.

- 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.

Y2K Myths

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

- *Date of last change* may be encoded in a file.
- When inspectors find these "date stamps" while looking for Y2K problems. they tend to ignore the.
- This is usually justified but the whole file is available to the software that processes it.
- Date stamps may be used by support software in archiving.
- Use of this data is unlikely, but ... for example ...
- A subtle bug may occur if an unexpected date such as "00" is present.

It is necessary to show that date stamp information is not used: we cannot assume that to be the case.

Y2K Myths

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

- Lists of critical" dates, 1.1.2000, 29.2.2000 or 9.9.1999 are very popular.

- There are many known programming errors that will lead to errors on these dates.

- We cannot be sure that a program that does not fail on these special dates will not fail on other dates. Prudence requires extensive testing using a statistically valid random selection of dates.

Y2K Myths

Myth 11: You can rely on keyword scan lists

- There are many lists of keywords that may present at date-sensitive parts of the software.Some organisations seem to think that they can find all potential points of failure by scanning for words like "date", "year", or “time".
- 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
- Programmers are notoriously capricious in their choice of identifiers and their choice of in- line comments.
- A programmer may use "jaar" instead of "year"Programmer may believe that "year" is 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.

Y2K Myths

Myth 12: The original safety cases remain valid.

- A "safety case" shows that the risk of failure is acceptably low.
- Usually the "safety case" is based on 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.

Systematic methods

Three methods that have been proven practical in practice and should be applied

- Data Flow Diagrams
- Slicing
- Inspection by creating Program function tables

Decomposition based on data flow diagrams

- We are often looking at complex networks or complex software systems.
- e must use the "divide and conquer" philosophy,
- Division cannot be done arbitrarily.
- Decomposition will only be helpful if the subsystems that are identified are relatively independent.
- We must know the information-flow between the system in question and other systems.
- We may "fix" systems so they work - but not together.
- 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.
- Data-flow diagrams provide a lot of insight.

Decomposition based on data flow diagrams

- Diagrams can be made more precise by providing using mathematical notation.
- In the case of Y2K analysis, one must know whether or not the information flowing is date dependent.
- Include all information-flow, not just that information flowing along wires while the system is running.
- 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.
- Physical placement, administrative placement, etc. are not critical issues. It is the information- flow that matters.
- "Every bit counts".

Even small amounts of information that are transferred very infrequently must be shown.

Slicing

- We must search a program for relevant sections.
- It is very easy to miss a relevant section
- Slicing is a way to reduce a program to a smaller program that contained only the relevant sections
- In slicing identifies variables of interest, then finds the pans of a program that touch those variables.
- Variables used or modified in those statements are added to the set and the search repeated.
- Eventually, the search terminates having identified all lines in the program that may affect the variables.
- 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.
- Slicing is a systematic and trustworthy method that can be applied by hand.
- Some programming techniques make "slicing" more difficult, e.g. machine- language, using indexing or indirect referencing, using pointers, using arrays in 'clever" ways.

Program function table analysis

- When analysing a complex we must 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 mathematical function.

- The functions are more difficult to describe than those encountered in analogue systems. A tabular format allows these functions to be written and understood by the average engineer.

An Informal Introduction To Program-function Tables.

	Array B contains x [1]	Array B does not contain x [2]
j'1	B[j'] = x	***true***
Present'=	**true**	**false**

NC (x, B)

1. "Array B contains x" is defined as ($\exists$ I, B[I] = x)
2. "Array B does not contain x" is defined as ($\forall$ I, (($1 \leq I \leq N$) $\Rightarrow$ $B(I) \neq x$))

- Program must search an array, B, for an element whose value is that of the variable x.
- Program must determine the value of two program variables called "j" and "present".
- The variable j records the index of one element of A whose value is the value of the variable. The variable called "present" 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.

More on Tables

	Array B contains x [1]	Array B does not contain x [2]
j'1	B[j'] = x	***true***
Present'=	**true**	**false**

NC (x, B)

- Header shows two situations must be distinguished.
- Each row corresponds to a program variable
- Entry describes the final value of this variable.
- Left header identifies the variable whose value is described in each row and how value is described.
- A "1" in the vertical header indicates that the variable's value must satisfy a condition given in the cells in the main grid.
- When "=" appears instead of "1", the expressions must evaluate to the value of the variable.
- The condition "NC(x, B)" is true if x and B are not changed by the program.

Advantages of Program Function Tables

- Nothing that can be said with such a table that cannot be said using equivalent conventional Boolean expressions.
- Using the table, one can select the row and column of interest and need not understand the whole expression. The number of characters that appear in the tabular expression is usually reduced 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.

Tables of this sort were used in the inspection of safety-critical software for the Darlington Nuclear Power Generation Station.

Program Function Tables are useful whenever a very careful and disciplined inspection of software is required. They "divide and conquer"

Conclusions

Classically educated engineers know that they must apply science and mathematics in their work.

Most programmers have not received an appropriate professional education and build and inspect programs in a purely intuitive manner.

The consequence is 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.

References

1. Archinoff, G.H., Hohendorf, RJ, Wassyag 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, 1., "Assessment of Safety-Critical Software in Nuclear Power Plants", Nuclear *Safety.* vol. 32, no. 2, April-June 1991, 189-198.

3. Parnas, D.L., Maday, J., "Functional Documentation for Computer Systems Engineering" published in Science *of Computer Programming (Elsevier)* vol. 25, number I, October 1995, pp 41-61.

4. Parnas, D.L., "Tabular Representation of Relations", CRL Report 260, Communications Research Laboratory, McMaster University, October 1992.

5. Parnas, D.L. "Mathematical Descriptions and Specification of Software", *Proceedings of IFIP World Congress 1994. Volume 1* 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., Madoy, 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 on Conference on Software Engineering, March 1981.

9. Weisa, M., ’Programmers Use Slices When Debugging", Communications of the ACM. 25(7), pp. 446-452, 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 bed.. September 1986. Prentice Hall.