

# Certifying Y2K “Fixes”

Jeffrey Voas

Reliable Software Technologies

**ABSTRACT** *There is much less talk about certifying the correctness of Y2K conversions than there is talk about how to make the conversions. Certifying that year Y2K “fixes” were appropriate can be done rather easily by using a combination of different software testing techniques. Here, we describe these different testing techniques, and why they should be considered as essential processes in any Y2K conversion solution.*

Maintenance is a software life-cycle phase whose processes can cost more than the total costs to develop the code. This is particularly true for software systems that are expected to continue in service for 20-30 years. Here, maintenance can account for 50-70% of the total life-cycle costs for a software system. And of these maintenance costs, testing can account for 50% or more [3]. Thus the cost to test modified code can be a substantial portion of the total costs of keeping legacy code alive.

The maintenance phase is responsible for overseeing two key activities: (1) fixing faults in existing code, and (2) adding new functionality to existing code. Y2K conversions can be placed into both categories depending on perspective and age of system. If you take the position that Y2K problems are programmer mistakes, then Y2K conversions can be viewed a fault eradication. (This position makes sense if the systems were fairly recently built.) If you take the position that the Y2K problem is a situation where software outlived its intended life-span, then Y2K conversions are simply adding functionality to aged systems. (This position makes sense for systems built many years ago.)

Regardless of your personal bias on this issue, the fact is that modified code should be tested and unmodified parts of the system should be re-tested. Testing and re-testing must occur if you want to have confidence that your converted system is now Y2K immune. And as already stated, these testing expenses can be pricy. One prominent reason for this is that few testing tools exist that are smart enough to automatically know how to minimize testing costs for modified code. To us, a “smart” tool would be one that could determine exactly what code needed retesting. Wouldn’t it be great if an automated tool could determine this in a “optimized” mode, *i.e.*, determine the least amount of code that needs retesting in order to demonstrate that a code conversion was correct.

Consider these facts with respect to the Y2K problem. It has been speculated that the total cost of “fixing” this problem globally is \$600B [1]. In fact, it has been suggested that the legal liability costs of this problem could top out at around \$1,000,000,000,000 [2]. To eliminate the Y2K problem for a particular system requires (1) identifying what parts of the

software need modification (and making the conversions), and (2) testing the conversions. According to [5], more than 60% of all Y2K costs will go to testing. And Robert Scheier from Computerworld states that this figure can be as high as 70% for some projects [4].

All of this suggests that there is a serious need for tools that seamlessly integrate with Y2K conversion tools and test Y2K conversions. If such existed, the total cost of the Y2K problem (globally) could be reduced (while still providing sufficient confidence that Y2K conversions were correct). We will now highlight what we believe these tools need to accomplish in order to provide the appropriate levels of confidence. In short, this will serve as our checklist of testing processes for certifying Y2K compliance.

First, the absolute minimal requirement is that modified code gets tested (or what is commonly referred to as “exercised”). If modified code is not exercised, it is not possible to know what its behavior will be. To exercise code, you need to generate test cases that execute the conversions and then employ simple *coverage* analysis to analyze whether modified statements are hit. This can be done easily if: (1) the conversion tool places comments in the converted code, (2) the coverage tool’s parser looks for those comments, and (3) the coverage tool then places instrumentation to record when those statements are exercised. Once coverage testing is successfully completed, we know that all code modifications have been executed at least once.

But from a quality perspective, recognize that simply reaching statements is barely sufficient. The reason we say “barely sufficient” is because there are other forms of coverage testing (e.g., *dataflow*) that are better at fault detection than is statement testing. For example, dataflow testing would allow you to test “all uses” of the year fields that were modified. Or if you attacked your Y2K problem by adding complex conditions (e.g., like changing

```
if y1 < y2 then
    years_apart = |y2 - y1|
```

to:

```
if (((y1 < y2) and (y2 < 00)) or ((y1 >= 00) and (y2 >= 00))) then
    years_apart = |y2 - y1|
else if (y1 <= 99) and (y2 >= 00) then
    years_apart = (99 - y1 + 1) + (y2 - 00)
```

to avoid increasing the size of year fields), then you should use a coverage testing approach like multiple-condition coverage (MCC) or condition/decision coverage (C/DC). Note that dataflow, MCC, and C/DC coverage testing are more thorough than statement.

But coverage testing is only one ingredient in Y2K certification. After all, even complete coverage testing of all code modifications does not imply that all Y2K conversions are correct. Recognize that all “fixes” could be correct, but the fixes may have broken system functionality. That is, all of the year fields may now work properly, but other functionality that did work now does not. (Such a situation could occur because of a lurking fault that could not be triggered until a year greater than 1999 is used.)

This potential problem needs to be mitigated via *retesting* of existing functionality. To do so, regression testing can be employed, and thus that is the next ingredient that we need for Y2K certification. *Regression* testing uses a suite of test cases (usually with respect to the

requirements/specification) to ensure that the outputs from the original code and converted code are identical for each member of the suite. Note here that it is assumed that for each member of the suite, we want identical behavior for both versions of the code. By doing this, we have evidence that the conversions have not affected functionality that they should not have affected.

But there will also be inputs to the original version that we want dissimilar outputs for from the converted version. After all, if there are not, then why did we convert the software? To determine that the new version is doing what we want for these inputs, *system-level* testing should be employed. Here, system level testing will employ test cases that correlate to events that represent events beyond the year 1999. System-level testing will serve as our last ingredient in Y2K certification.

It is important to note that system-level testing is neither a substitute for coverage testing nor *vice versa*. It is possible to exercise all code conversions and not discover that a conversion fails in the context of a test that represents an event after 1999. Likewise, it is possible to system-level test with a wide variety of post-1999 scenarios that do not exercise all modifications. Thus both forms of testing are needed for Y2K certification.

In summary, to certify that code is Y2K compliant, three different forms of testing should be employed:

1. Coverage testing to exercise fixes,
2. Regression testing to see whether new code breaks other system requirements that are not related to calendar dates, and
3. System-level test to see how the new system handles events past 1999.

These techniques do not guarantee that the conversions will integrate into your system and work correctly under all scenarios. Except for exhaustive testing, testing can never make such guarantees. Instead, testing provides confidence. And for legacy Y2K systems, that is what is needed after these “tried and true” systems are upgraded to handle events after 1999.

Since testing can account for 50% of maintenance costs, if you were planning to spend  $X$  dollars on conversion, the additional certification costs could double your cost to  $2X$ . But without taking these defensive measures, you could get fooled into thinking that your Y2K problem is behind you when it is not. Given that you have taken the proactive measures to solve your Y2K problem, this is a situation you’ll want to avoid.

In this article, I have deliberately simplified the Y2K certification process down to a handful of traditional testing approaches. Admittedly, there are more advanced certification processes that could be employed which provide similar results. Given that much Y2K conversion is ongoing without assurances that the conversions are correct, I felt that laying out the basic needs for Y2K certification would be more beneficial to the practitioner than laying out a Utopian “pipe dream” (such as proving the legacy system is correct!).

## References

- [1] COMPUTER NEWS DAILY. "Year 2000 Prophet Preaches \$600 Billion Digital Fix", October 1, 1997.
- [2] D. HASSETT. "Frequently Asked Questions about the Year 2000 Problem", available at: <http://www.y2k.com/legalfaq.htm>.
- [3] G. MYERS. *The Art of Software Testing*. Wiley, 1979.
- [4] R. L. SCHEIER. "Year 2000: Testing Can't Wait", Computerworld, October 20, 1997, available at: <http://www2.computerworld.com/home/online9697.nsf/All/971020test>.
- [5] TECH-BEAMERS. "White Paper Year 2000 Focus On Testing", May, 10, 1996, available at: <http://www1.mhv.net/~techbmrs/tstgwp.htm>.