Running Head: SOFTWARE TESTING

Software Testing as Applied to the Year 2000 Computer Problem

Daniel McCollum

A Senior Thesis submitted in partial fulfillment
of the requirement for graduation
in the Honors Program
Liberty University
Spring, 2000

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial fulfillment of the requirements for graduation in the Honors Program of Liberty University

_____

Terry Metzgar, Ph.D.
Chairman of Thesis

_____

Monty Kester, Ph.D.
Committee Member

_____

Carl Merat, M.S.C.S
Committee Member

_____

James Nutter, D.A.
Honors Program Director

_____

Date

Abstract

Maintenance is an important step in the development of software and one in which deserves to be examined with rigor. Because testing is also important to software development, and a majority of development cost is spent in the maintenance phase, then it is quite apparent that testing software in the maintenance step is crucial to the development of software. The principles of testing computer software are examined and specifically its implementation in the maintenance phase. From here, these principles will be applied to the Year 2000 problem, which consumed much of the computer industry's attention for several years around the turn of the century, to demonstrate how testing took place to validify the compliancy of software. Since the Year 2000 glitch is a prime example of how software maintenance is carried out, it will be used as an illustration of testing during software maintenance. To apply this even farther, an case study is done on the Year 2000 that was carried out on the Reactivity Measurement Analysis Software from Framatome Technologies. The testing principles discussed will be seen clearly in the application of this case study.

Software Testing as Applied to the Year 2000 Computer Problem

Introduction

The improvement of the software development cycle has been a topic of much interest throughout the computer industry. The main reason for this is the enormous size of software projects undertaken by companies as they strive to produce bigger and better software. Much research and study has gone into improving the software development practice, which has resulted in a universal acceptance of the need for some process to produce software. The results of this can be seen in the plethora of software life-cycle models presently in use by software firms. Though the software life cycles used have changed, and continue to change, the foundation to producing good software continues to be in testing the software throughout each phase.

The software life cycle is used to provide some kind of structured repeatable process for producing software. In general, the software life cycle can be divided into seven phases. Each phase is listed below and will be briefly discussed.

(1) Requirements Phase. During this part of the process, the feasibility of attempting the project being proposed is explored. The requirements to meet the needs of the customer are determined.

(2) Specification Phase. The customer's exact requirements have been determined and are put into a working document. It is at this point when specific requirements of what the software is supposed to do are laid out.

(3) Planning Phase. The timelines, cost, personnel, and other managerial decisions pertaining to the project are decided upon.

(4). Design Phase. The design phase is "sometimes split into two subphases: architectural or high level design and detailed design" (Ghezzi, Jazayeri, and Mandrioli, 1991, p. 6). The high level design breaks the whole project down into modules while the detailed design determines how the modules will work.

(5) Implementation Phase. The modules are coded and tested individually.

(6) Integration Phase. The modules are brought together and tested as a whole. This is the first time when all the components are brought together for a total trial run.

(7) Maintenance Phase. Once a product is determined to meet all specifications, then it is released to the customer. It is at this point when software enters the maintenance phase.

It is important to remember that these phases may not be completely followed in this order. If at some point in the cycle it is determined that an earlier phase was not done properly or the specification from the customer changes, then it is possible to go back and update phases. These phases are not set in stone and often times it is necessary to go back and forth between the various phases. Figure A shows the relative cost of the phases of the software life cycle.

As software life cycles are examined there is no disputing that "maintenance is an extremely time-consuming and expensive phase of the software life cycle" (Schach, 1996, p. 11). Maintenance is thus an important step in the development of software and one in which deserves to be examined with rigor.

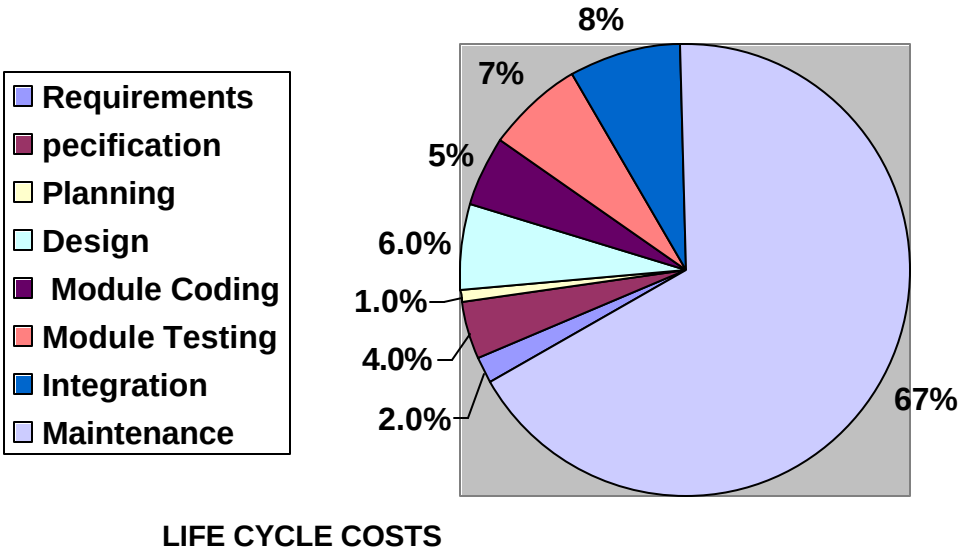

Figure A. Approximate relative costs of the phases of the software life cycle (Schach, 1996, p. 10).

Because testing is also important to software development, and a majority of development cost is spent in the maintenance phase, then it is quite apparent that testing software in the maintenance step is crucial to the development of software. The principles of testing computer software will be examined and specifically its implementation in the maintenance phase. From here, these principles will be applied to the Year 2000 problem, which consumed much of the computer industry's attention for several years around the turn of the century, to demonstrate how testing took place to validify the compliancy of software. Since the Year 2000 glitch is a prime example of how software maintenance is carried out, it will be used as an illustration of testing during software maintenance.

Testing Software

Testing software is a crucial part in delivering good software and its proper implementation is critical for software companies to achieve. Software testing manifests itself in a wide variety of ways and is not merely testing, which is the input and output of some set of data with test cases, at the end of the software cycle. On the contrary, testing should be present throughout the entire software life cycle in the form of reviews, inspections, audits, design walk-through, code walk-through, group code reads, and desk checks (Behforooz and Hudson, 1996). Each of these forms of examining software is known as static testing, which if carried on throughout the entire software development model, can help alleviate many of the errors that would show up in the dynamic stage of software testing. Dynamic testing in the maintenance phase of software development will be the primary focus of the following discussion.

Testing objective.

Software engineers often misunderstand the main objective of the software development process. Many developers believe that the purpose of software testing is to find all bugs and prove that the software works perfectly, but this is impossible. There is no way that software can be completely tested. Three reasons for this are: (1) the domain of possible inputs is too large to test, (2) there are too many possible paths through the program to test, and (3) the user interface issues are too complex to completely test (Kaner, Falk, and Nguyen, 1993). The complexity inherent to software proves to be too big a hurdle for testers to cross. To somehow design test cases that check all possibilities is impossible and for someone to think otherwise would be naïve. This is not to say that designing good test cases should not be a priority, but rather one must understand what the main testing objectives are in order to design proper tests.

In order to fully understand the main objective of software testing, several terms will be explained to lay a foundation. These terms are validation and verification. Software validation can be defined as "all actions taken at the end of the development cycle to confirm that the software product as built correctly reflects the SRS [software requirements specification] or its equivalent" (Behforooz et al., 1996, p. 302). Verification, on the other hand, can be characterized as "all actions taken at the end of a given development phase to confirm that the software product as being built correctly satisfies the conditions imposed at the start of the development phase" (Behforooz et al., 1996, p. 302). With this type of examination, it is important that static testing does take place to ensure that the requirement specification is accurate to the needs of the customer.

Once the specification is deemed correct and changes are made where necessary throughout the development cycle, then the ongoing development of the software can be verified during each step. If the requirement specification is not reviewed and checked for mistakes, then any mistakes present in the requirements will produce software that is tested against the incorrect specifications. This means that the level at which the problem resides is deeper than the code itself and correcting it is much more cost consuming than if the mistake would have been found at the beginning stages of the cycle. A significantly less amount of time is required to correct a problem when it is found during the requirements or design stages of a product before the implementation phase has been started. To fix an error in the requirements of a product which is found during implementation, or even later, requires that the design be changed and then implementation be adjusted accordingly. It is quite apparent that the sooner an error is found in the software life cycle, such as the requirements or design phase, then the development cost for a piece of software will be much cheaper. Mistakes found during maintenance can be the most costly, for a product may have to be redesigned or much regression work must take place to fix a problem. These types of problems may have resulted in minimal cost impart if they would have been detected during the design or implementation of the software.

This leads to the definition of the main objective of software testing which is "to prove (or when such proof is not possible, to show with a high level of confidence) that the software product as a minimum meets a set of preestablished acceptance criteria under a prescribed set of environmental circumstances" (Behforooz et al., 1996, p. 300).

This involves testing the requirement specification for its correctness as well as testing the software to those requirements. Testing in this manner is impossible because the software can never fully be proven to be correct. As discussed earlier, software is so complex that all of the program states cannot be checked. This would take an extraordinary amount of time, which software companies do not have. Because of this, testing is not only trying to show that the program works correctly, but is trying to determine the errors that reside within the requirements and the code. This is important to have as a basis, since test cases will be designed to accommodate these objectives.

Developing Tests

The testing technique that will be examined for the purposes of this thesis will be dynamic testing. Dynamic testing is an extremely important part of software testing, especially during the maintenance of software. Software maintenance, which will be discussed later, relies heavily on testing the execution of software and through regression testing and testing verification.

Before test cases can be determined, a preface must be given as to what constitutes proper test cases. A test case determines the proper input and output data that should be tested against the system in a methodical fashion. Test cases must be systematic to ensure that a system is tested correctly and functions as determined by its specification. Data to be used for input testing must consist of data that is good, or expected, as well as unexpected errant data. Not only must the system handle correct input data, it must be able handle this unforeseen erroneous data. A good test case will specify the proper output in accordance to the input that is given to the system, whether it

is good or bad input data. The compliance of the system to this determined acceptance criteria, as found in the specification requirements, is what is used to establish the validity of the system.

Systematic testing is crucial in determining the correctness of a piece of software, especially during maintenance testing. Once a system has met the specification requirements, which is determined by test cases, then the base test cases can be determined. It is important to have this prior acceptance criteria to form the basis of the test cases to be run. If the system has not been previously methodically tested, then regression testing, which is crucial for proper maintenance testing, cannot be properly done. Though the system may appear to be functioning properly, it still needs to be compared to the original test case results, which were accepted as correct, to ensure full functionality according to the specification.

Dynamic software testing can be accomplished in two main fashions: (1) black box testing and (2) white box testing. Each one of these will be explored thoroughly to discover how they determine the correctness of software.

Black box testing.

The first technique to be examined is the black box module testing technique. Black box testing is a way to test such that the "tester is not interested in the interior make-up of the box, but just its functional performance as it converts input to output" (Behforooz et al., 1996, p. 331). Essentially, black box testing is testing in accordance with the specification as opposed to testing to the code. Each module or program is seen as a whole or complete object carrying out some specific purpose as defined by the

requirements. Thus the testing of the program is not concerned with how the program does what it does, but rather with the correctness of what it does. Hence, designing test cases can be extremely difficult because it is obvious that all possible inputs cannot be checked. This leads to the foundation for designing black box test cases, which is "to devise a small, manageable set of test cases so as to maximize the chances of detecting a fault while minimizing the chances of wasting a test case by having the same fault detected by more that one test case" (Schach, 1996, p. 408). Several techniques are used to maximize the design efficiency of test cases.

Equivalence classes along with boundary analysis are two such techniques that are used. An equivalence class is "a set of test cases such that any one member of the class is as good as any other member of the class" (Schach, 1996, p. 409). For example, a valid integer value for a field is between 1 and 999. The equivalence class would be less than 1, from 1 to 999, and greater than 999. Equivalence class testing is based upon the assumption that if any value that falls in one of these ranges is tested, then the outcome of that test is the same for all of the other values in the range. This is a critical theory because it can eliminate much unnecessary testing as well as assisting in defining what exactly needs to be tested.

Another technique that takes equivalence classes to the next level is boundary analysis. Boundary analysis is accomplished by checking the values on and just on each side of each equivalence class. It has been shown that many of the errors associated with programs can be located within this category of analysis. Many times boundary analysis can be used to show that a program handles incorrect data along with correct data. So

from the example above, the boundary analysis equivalence cases would be to check for a negative number, 0, 1, an integer value in between 1 and 999, 999, 1000, and then a very large integer value. By checking these values you can eliminate the majority of faults that may occur if an error is present.

White box testing.

Another way to test software is white box testing. With this technique, "test cases are selected on the basis of examination of the code, rather than the specifications" (Schach, 1996, p. 411). This type of testing is especially applicable during the implementation phase of software as programmers are coding modules. Since testing is based upon the code itself, this form of testing is accomplished through examination of the code. From this, the tests can be formed using three main techniques: (1) statement coverage, (2) branch coverage, and (3) path coverage. Statement coverage is the simplest form of white box testing and consists of executing every statement at least once. This ensures that every line of code is executed at least once and that any dead code can be fixed or removed. The draw back to this type of test execution is that it does not ensure that all the branches that are executed have the proper outcome. An improvement to this is using a technique known as branch coverage. This is accomplished by "running a series of tests to ensure that all branches are tested at least once" (Schach, 1996, p. 412). This is a good way to overcome the problems associated with statement coverage testing by executing all the branches, thus verifying their entry and exit. The last technique used for white box testing is the most powerful one that can be used. It is known as path coverage. Path coverage tests all possible paths of a program by running through the

code to ensure that each path is valid. This can be very time consuming and resource consuming, so many different implementations of this exist. It is worth noting that many tools do exist that assist in running these white box tests. Since it would take an enormous amount of resources to actually carry these tests out individually, these tools can be quite helpful and cost efficient.

Maintenance

The maintenance phase is by far the most important part of any life cycle model. In fact, “about two-thirds of total software costs are devoted to maintenance“ (Schach, 1996, p. 11), which shows what an extremely crucial part of the life of software. Since the life of a software product can be consumed with maintenance, it is a common misconception that the more time spent on software maintenance is a reflection of a poor piece of software. On the contrary, maintenance of software in not merely fixing programming flaws, but rather most of the time is “spent on enhancing the product with features that were not in the original specifications or were stated incorrectly there” (Ghezzi et al, 1991, p. 24). Though some maintenance can be performed on poorly designed software, it is much easier to maintain when it has been designed properly. Thus, only software that is well designed and properly implemented will have the capability of being maintained over many years.

Three different types of maintenance can be performed on a delivered piece of software: perfective, adaptive, and corrective maintenance. As Figure B shows, more time is spent on perfective maintenance than any other type. Because the various maintenance types are frequently misunderstood, each one of them will be briefly discussed in order for the reader to fully understand the forms in which software maintenance can transpire.

Three types of maintenance.

The first category of maintenance that will be discussed is perfective maintenance. This occurs when software is changed to improve the use and functionality of features. These types of "changes are due to the need to modify the function offered by the application, add new functions, improve the performance of the application, make it easier to use, etc" (Ghezzi et al., 1991, p. 26). Perfective maintenance is more of an extension rather than a correction of some software flaw.

The second form of maintenance to be examined is adaptive maintenance. This kind of maintenance takes place when the environment in which the product is used is altered, resulting in the need for the software to be changed. This can occur in many various ways, several of which are: porting to a new compiler, operating system, or hardware, the changing of some predefined process in which the product had a role, altering the representation of some piece of data, and any other environmental transformations that may take place.

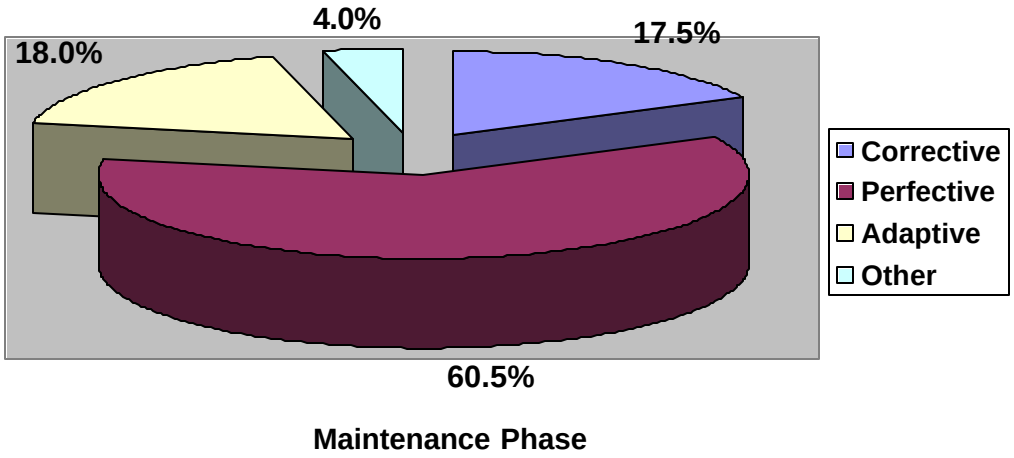

Figure B. Percentage of time devoted to each of the types of maintenance (Schach, 1996, p. 463).

An example of how this may occur can be seen when the U.S. Postal Service introduced the nine-digit zip code in 1981. All products that allowed only five-digit zip codes had to change to allow for nine-digit zip codes (Schach, 1996). Thus the software had to adapt to this alteration brought forth by the environment in which it existed.

The last type of maintenance that will be discussed is corrective maintenance, which is the primary focus of the last study for this thesis. Corrective maintenance deals with "the removal of residual errors present in the product when it is delivered as well as errors introduced into the software during its maintenance" (Ghezzi et al., 1991, p. 26). When most people hear about software maintenance, this is the type of maintenance that comes to their mind. Though corrective does occur, only about eighteen percent of maintenance appears in this form (Schach, 1996). Although it does not occur as frequently, it is vital to the survival of a product that these errors are fixed within a timely manner and without introducing new faults.

Three stages of testing during maintenance.

Within the maintenance phase there are three stages of testing that must be done to ensure that the changes were made correctly. These three stages will be briefly described, but specific testing principles will not be discussed here. The first step is unique in that it is used mainly for corrective maintenance. This stage of testing is concerned with verifying that a problem does exist and determining exactly what the symptoms of the problem are in the software. In general, this step is not used to narrow down where exactly in the code the problem may reside, but rather it determines as precisely as possible what the problem is with the product. The second step in testing

during the maintenance phase is “checking that the required changes have been correctly implemented” (Schach, 1996, p. 462). This type of testing must be done to ensure the design and implementation of the new modifications have been carried out accurately. The last step involves verifying that the alterations made to the software did not produce in any inadvertent changes to the correctness of the product. This usually means that the new modified software must be retested and its results compared to those which were previously accepted as correct. This type of testing is called regression testing. If the system has not been previously methodically tested, then regression testing, which is crucial in maintenance testing, cannot be properly carried out. Though the system may appear to be functioning properly, it still needs to be compared to the original test case results, which were accepted as correct, to ensure full functionality according to the specification.

Case Study: Year 2000

The following section will be used to examine the Year 2000 problem and show how the principles for software testing can be applied. The turn of the century presented many obstacles for the computer industry to overcome concerning the correctness of computer systems. Much time and many resources went into the testing and correcting of computer systems over several years right around the year 2000. In fact, over $180 billion was spent in these areas as companies attempted to become Year 2000 compliant. Because this endeavor was so encompassing, it represents a good case to examine in order to illustrate how testing within software maintenance can be applied.

The Year 2000 problem existed in many various forms, but in general they all dealt with computer systems that could not handle dates in the 21st century. Dates such as January 1, 2000, February 29, 2000, or other various dates were not correctly calculated by the hardware, operating system or software. To determine what systems would handle the change of the century correctly, much testing was completed before the year 2000. The principles discussed earlier will be applied where applicable in explaining how testing was accomplished to determine Year 2000 compliancy.

Year 2000 Compliancy Basis

To fully understand the basis on which testing was accomplished, it is crucial to realize the foundation for testing. As Microsoft leads the industry in the production of computer software, they have presented a list of criteria that Microsoft products must pass to be year 2000 compliant (Microsoft: Year 2000 Test Criteria, 1998). These criteria can be used in the evaluation of not only Microsoft software but also for the evaluation of the compliancy of any hardware, operating system, or software. Though further criteria will be presented for certain types of software, the following criteria serves as basis for Year 2000 testing. These criteria are:

- The product stores and calculates dates consistent with a 4-digit format throughout its operational range.
- If the product allows the user to enter a 2-digit short cut for the year, the product recognizes the year consistent with a 4-digit format
- The product will correctly execute leap year calculations.

- The product does not use special values for dates within its operational range for data.
- The product will function into the 21st century, through the end of year 2035.

Compliancy Classification

The level of compliancy of any hardware, software, or operating system can be classified in the following several categories depending upon whether it meets the above criteria for compliancy (Microsoft: Year 2000 Product Guide, 1998). These classifications of Microsoft products will be extended to form a basis for the formulation of Year 2000 testing guidelines, especially for operating systems.

Compliant	The product fully meets Microsoft's standard of compliance. May have prerequisite patch or service pack for compliance
Compliant with minor issues	The product meets Microsoft's standard of compliance with some disclosed exceptions that constitute minor date issues
Not Compliant	The product does not meet Microsoft's standard of compliance
Testing yet to be completed	Product test is not yet complete or has not been started but will be tested
Will not test	The product will not be tested for compliance

Parts of the Computer System to Test

In order for an entire computer system to be Year 2000 compliant, the hardware, software, operating system, and all other devices or programs that use clock functions must be individually compliant. One of the main obstacles to overcome with this is the fact that computer technology uses dates in a wide variety of ways. Because of this, if the date is not Year 2000 compliant in one area of the computer, it can cause havoc

throughout the whole computer. If the motherboard's BIOS does not handle the date correctly, then a program or operating system that relies on that date will not operate properly. The same is true for a program that is dependent upon the operating system's clock. If it is wrong, then it will cause the program to falter. As software testing is examined, it is crucial to understand that software is dependent upon the system on which it runs. Especially in the case of testing for Year 2000 compliancy, the platform on which the software resides is a critical part of determining its compliancy. For this reason, the Year 2000 issues within hardware and operating systems will be discussed in the following section.

Hardware Issues

The CMOS, BIOS (Basic Input/Output System), and Real Time Clock (RTC) offer a pervasive problems in the Year 2000 predicament and will be investigated. Since the majority of personal computers each have one of these components present, their non-compliancy to Year 2000 standards can be of much concern. Furthermore, since most computers utilize the CMOS, BIOS, and RTC at the beginning of every startup, its compliancy, or lack thereof, can affect every area of a computer system. Because of their relevancy and implementation scheme, an examination of how their function is achieved will be explored.

Understanding the design and function of the BIOS and RTC is crucial to the overall comprehension of the Year 2000 problems they may contain. Since the BIOS and RTC are foundational to many of the compliancy glitches found in computer systems,

this more in-depth knowledge serves as a basis for total understanding of how software testing for Year 2000 is dependent upon hardware.

The BIOS was originally implemented in 1977 by Gary Kildall in order to standardize hardware control data and provide a "separate set of configuration information versus the custom configuration information that had to be in each computer's operating system" (MITRE Y2K Team, 1999). Until this point, each computer model had to contain an operating system that was specifically implemented for its make. The general function of the BIOS is to "provide the basic instructions for controlling system hardware" (Newice: Year 2000, 1999), which "the operating system and application programs both directly access to provide better compatibility" (Newice: Year 2000, 1999). The BIOS is thus a low-level driver used for interaction between software and hardware. These instructions are usually stored in ROM and then loaded from ROM to "start up the hard disk so that the operating system can be loaded" (Newice: Year 2000, 1999).

Before 1984, the user of a computer had to manually set the system clock every time the computer was started by a cold boot. As personal computers (PC's) became user-friendlier and their use extended to the general public, a method of time keeping while the computer was turned off was needed. This development was introduced in the form of a Real Time Clock and CMOS memory to store the time while the computer is off.

As noted above, the Real Time Clock and BIOS play an integral part of the time keeping capabilities of computer systems. A general functional overview of how the two

work together to maintain the proper time will be examined. When the computer is shut down, the Real Time Clock keeps the current time. The Real Time Clock "is maintained in a battery-backed computer chip. The RTC functions as a running clock and a keeper of two digit year values (the '97' in '1997')" (Kaplan, 1997). The Real Time Clock can be thought of as just a set of counters that increment to keep the current date and time. The first counter increments from 0-9 and upon rolling over to 0 the second place counter increments by one. This second counter is the ten's place counter for counting seconds and is likewise incremented when it rolls over to 0. This process continues to track minutes, hours, and days. The number of days incremented varies from month to month according to the corresponding number of days for each month. The months and years are recorded when the "counter proceeds to go from 1 to 12, and then of course the year counter starts its journey with the good old 0 to 9, and finally we have the year 10's counter again with values going from 0 to 9" (Real-Time Clock and CMOS, 1999). But this is where much of the Year 2000 problem exists in computer systems. Since the date is usually kept as a two-digit year, then the century is not kept properly and any software or hardware that directly accesses it will be given an incorrect date.

When a computer is booted up, the BIOS is run to load the operating system and the current settings for the computer. One of the many configuration details that must be set is the current time and date. The BIOS completes this function by reading in the current date and time as found in the Real Time Clock. Since the Real Time Clock stores the century date in a two-digit format, then it is the responsibility of the BIOS to set the appropriate century date.

Most systems BIOS chips are configured to assume the century to be 1900, so when the date is read from the Real Time Clock, the two-digit year is appended to the century date 19. Because of this, when a computer is booted after being turned off for the change of the century, then "the BIOS will be told by the RTC that the year is 1900… [while] some BIOS will convert this to 1980" (AMI, 1999). This date is then loaded into the operating system as the correct date for year 2000 dates.

Since some operating systems have techniques to monitor the date and correct it when the BIOS is incorrect, one would think that the BIOS error could be ignored. But this assumption is false for several reasons. Software applications can be written to "request the date and time from the OS [Operating System], the BIOS, or the RTC." Though most applications retrieve the date from the operating system, some solicit the information from the BIOS, and even fewer ask the Real Time Clock. This presents quite an opportunity for error within applications that access the date in a way other than calling the operating system.

The problems associated with the BIOS assuming the century date to be 1900 when read in from the Real Time Clock can be numerous. Several major ones are listed and briefly discussed below.

Impact of BIOS Problems
1. Files that are created and /or modified will be date-stamped with the wrong date. This impacts versioning and sorting the files in a list. Additionally, if the files are used for legal purpose, incorrect dates may have a more serious impact.

2. Erroneous processing. Many PC applications use the system date for processing. For applications that do more than print the current date on report, that is, applications that perform calculations using the current system date or perform sort functions using the current system date, erroneous system dates pose a serious problem.
3. Operating systems must be designed to monitor for errors in the turn of the century. If operating systems wish to oversee the correct time and date it is given, then some mechanism must be implemented to do so.

(Commonwealth of Massachusetts, IT Division, 1998)

In conclusion, many of the problems in computer systems because of the turn of the century can be attributed to the improper assignment of the century by the BIOS. Because of this error, when the date is sent from the Real Time Clock to the BIOS, the BIOS assumes the century date to be 1900. Thus, when a computer is booted and the date and time is read in by the operating system then the incorrect date retrieved. Though patches within operating systems can monitor and correct faulty dates, they cannot monitor against software that directly accesses the BIOS or Real Time Clock. Though this does not occur frequently, some programs have been written in this format, which inevitably results in the wrong date.

Operating System Issues

The operating system is a vital part to the performance of the computer, and its reliability is a must. With a vast array of operating systems on the market today, each one has its own way of handling the year 2000 problem. The most widely used operating systems will be examined to determine their level of compliancy in regards to their

operating status during the 21st century. Three main operating systems will be briefly discussed to reveal some of the issues that exist when testing on specific platforms. These are Microsoft, Unix/Linux, and Macintosh.

Microsoft.

Microsoft operating systems can be divided into two categories, DOS and non-DOS based. Windows 3.1 and 95 have as their underlying basis MS-DOS. Each platform sits upon DOS and uses it as its main interaction with the hardware. Thus a summary of how the MS-DOS operating systems handle the year 2000 problem would be appropriate for a general overview of how the DOS based operating systems handle the Year 2000.

In general, MS-DOS is characterized in that it "recognizes dates beyond the year 2000 … [but] does not display the full year, but will sort files correctly" (Microsoft Year 2000: MS-DOS, 1998). DOS handles dates entered from 1980-1999 correctly when entered from the command *Date* using a 2-digit format (i.e., 05/15/88). If a date which falls in the 21st century is attempted to be entered in a 2-digit format, the operating system will return an error, Invalid Date. The operating system, however, will handle the setting of 21st century dates in a 4-digit format (i.e., 8/23/2006). Whenever a program receives the date from an API program, the program must add 1980 to the date. This is caused because "MS-DOS file system APIs use a year offset from 1980 to store dates (Microsoft Year 2000: MS-DOS, 1998).

There are several other compliance issues that exist in MS-DOS versions 5.0 through 6.22. The first one is that DOS cannot display a 4-digit date when the command

DIR is used. The date shown is displayed in a mm/dd/yy format and does not allow for dates following December 31, 1999. Another compliance problem comes about when external programs wish to modify the date internally; they must enter dates from the 21st century in a 4-digit format. This is crucial to programs that work directly with the utilization and enhancement of MS-DOS as an operating system. They must be configured to update the year in the proper way or many problems can result due to this errant coding. Lastly, *MSBACKUP* has several problems with the use of dates following the year 2000 (MS-DOS 5.0 does not have these problems because the tool *MSBACKUP* is not a part of the operating system). The errors present come in two forms, the first is that *"MSBACKUP*: naming conventions do not recognize the 'tens' place" (Microsoft Year 2000: MS-DOS, 1998). "When a backup is made with the same number in the 'ones' place and a different number in the 'tens' place (i.e., 1996 and 2006), *MSBACKUP* treats them as being made on the same date" (Microsoft Year 2000: MS-DOS, 1998). A letter is added to the date to show that the files are different. The second compliancy problem is that *MSBACKUP* does not store the dates after 1999 correctly. When a backup file posted 20th century is to be loaded to overwrite an existing file, it displays a warning that the existing file is newer than the backup file. The option to overwrite this warning is given and the 21st century files will be loaded properly if the warning is bypassed.

Windows 3.1 as an operating system is reliant upon MS-DOS for much of its functionality including clock operations. This means Windows 3.1 uses the underlying MS-DOS operating system to accept dates. Just as noted above, the *Date* command used

by MS-DOS will not accept 2-digit date changes for the year 2000 and beyond. Windows 3.1 uses a graphical interface DATE/TIME property to communicate with the DOS *Date* command; thus the date must be entered in the same 4-digit year format as with the *Date command*.

Since Windows 95 also uses MS-DOS as its basis of operation, it handles the use of dates much the same, though several variations are present. All dates are stored in a 4-digit format except for those that are MS-DOS API's. Unlike Windows 3.1, the DOS version in Windows 95 accepts dates into MS-DOS *Date* command in 2-digit and 4-digit format (Microsoft Year 2000: Windows 95, 1998). Windows 95 programs must also add 1980 to DOS API's to get the appropriate date needed. Win32 API's are an exception to this rule. They are not affected by the 1980 offset.

Since Windows 98, 2000, and NT are not dependent upon MS-DOS, they have been designed so that they are Year 2000 compliant. Thus, Microsoft classifies these operating systems as being "compliant" to the year 2000 criteria as presented earlier. Windows 98 follows the forerunner Windows 95 by storing the dates in 4-digit format except for API's (they are stored as previously noted). Also, Explorer displays dates in 2-digit format unless selected otherwise. Other minor issues exist within these operating systems, but are insignificant in regards to testing on the various platforms. It is important to realize that when testing on a platform, you need to research the platform's exact known Year 2000 issues. From this, an analysis can be made to determine if the specific software utilizes the functions from the operating system that may cause compliancy issues.

Unix and Linux.

The versions of Unix and Linux distributed by most manufacturers (i.e. Debian and Red Hat) in general are similar and are year 2000 compliant. Linux and Unix both store dates in a 4-digit format, but allow programs to store dates in a 2-digit format if requested. The reason they are compliant is that "They store dates as a count of seconds since New Year's Day 1970 [and] this counter will overflow about 40 years from now, in early 2038, not 2000" (YEAR 2000 Compliance Statements of Some Distributions, 1998). The way they access the date is through calling a function Time_t. Currently this function is a 32-bit variable and merely needs to be changed to 64-bit by the year 2038; it should then be fine for millenniums to come. Most current versions of Unix and Linux do read "the time at boot up from the CMOS clock chips of the machine" (Linux, 1998). This implies that each individual system on which Unix or Linux runs must use a compliant CMOS clock chip in order to read the clock properly.

Mac OS and Apple Macintosh.

Apple Macintosh and Mac OS were designed from its beginnings in 1984 to be free of any turn of the century bugs. The date is handled by Macintosh operating systems in an error-free structure in regards to the storage of the date. These operating systems use a "32-bit value to store seconds, starting at 12:00:00 a.m., January 1, 1904 and ending with 6:28:15 a.m. on February 6, 2040" (Macintosh, 1998). The only known concern with the Mac operating systems is within the Date & Time control panel. This program "constrains user input to dates between January1, 1920, and December 31, 2019" (Bechtel, 1998). The date can be set up to 2040 using the function *SetDateTime* found in

the Macintosh Toolbox. Macintosh also handles leap year and recognizes February 29, 2000. It uses a unique way of keeping up with the date by utilizing the Gregorian calendar system for keeping track of days. The way this is accomplished is that the "Gregorian calendar is a solar calendar that measures time from the year of the birth of Jesus Christ" (Bechtel, 1998). It tracks years by having 11 months with fixed periods of 30 or 31 days while the 12th month contains 28 days except every 4th year. The Gregorian calendar is structured so that all years divisible by 100 must also be divisible by 400. Thus, 2000 is figured as a leap year.

In conclusion, the operating system of a computer is very vital to the proper handling of the Year 2000 bug. Though not the only factor in its compliancy, the operating system must handle the turn of the century to have an error free machine. With the various operating systems on the market, a wide variety of problems exist and vary from operating system to operating system. Most of the operating systems examined have minor compliancy issues, which can be taken care of with some sort of upgrade, or have been designed from their beginnings to run through the turn of the century.

Building Year 2000 Test Cases

Now that a foundation has been laid for testing software for Year 2000 compliancy, testing guidelines can now be applied to the Year 2000 problem. Just as testing any type of software or hardware, a plan for testing must be put into place to track down errors that would cause failure at the turn of the century. This test plan contains many important segments that make the process achievable, but determining the correct test cases is crucial in testing for Year 2000 compliancy. The next several sections will

cover two main principles when dealing with test cases to check for Year 2000 compliancy. These concerns cover what system components should be tested and what dates should be checked for proper validation of Year 2000 compliancy.

System components.

The system components must be fully test where they apply to ensure Year 2000 compliancy. The specific components to be tested will vary according to the type of system and the way it is used. Each component must be considered carefully so as not to miss any components that needed to be tested. Combinations of the various components need to be considered when developing test cases. Each component will be described and discussed briefly in the following table.

System Element	System elements must be tested before and after remediation. This can include but is not limited to software, hardware, and firmware. Test cases should validate that now desired pre-existing functionality has been lost (regression testing) and that new 'Year 2000' functionality works as expected (compliance testing).
System Data	Systems should be tested using data having current dates and dates that have been advanced beyond the Year 2000. Date-data that have been advanced beyond the Year 2000 may be referred to as aged data. Existing data can be aged or Year 2000 data set can be created to meet testing requirements. Systems should also be tested with the system date in or post Year 2000 with data from before Year 2000.

System Time	Systems must be tested using current system time and with the time set beyond the Year 2000. Setting a system time to beyond 2000 may allow the testing of many system factors not normally observable using current time. These factors include the operating system utilities that use internal time stamps, system archiving or backup utilities and other aspects of the internal operation of a system.
Data and Time Combinations	Test cases should represent all combinations of system time and date-data that are possible during the system's transition into the Year 2000. • System time prior to Year 2000 with data before Year 2000 • System time prior to Year 2000 with data after Year 2000 • System time post Year 2000 with data after Year 2000 • System time post Year 2000 with data before Year 2000

(The above was taken from IEEE P2000.2 Draft Recommended Practice for Information Technology: Year 2000 Test Methods, 1998, p. 35)

These elements should be considered when putting together test cases to check compliancy. It is important to verify that the any external programs or platforms that software may use or run on are compliant. Also the hardware on which a software product may reside must be confirmed to be compliant. These especially come into play when test cases are taking place, as variations may produce wayward results.

Dates to consider.

The dates to test may seem obvious at first, but in reality a broad base of dates must be tested. Once again, the testing of certain dates may vary from system to system and the test cases should be formed around the specific environment to be checked. A list of the dates to be considered when developing test cases will be presented below.

Date	**Reason**
1999-09-01 (Wednesday)	The four digit date format (YY-MM) is sometimes used, with the three digit input 99-9 as a representation of an unknown of 'out of range' date.
1999-09-09 (Thursday)	This date is commonly used to indicate an unknown date in 6-character (i.e. 99-9-9) data entry fields that don't require a leading zero.
1999-09-10 (Friday)	In systems that have used 9-9-99 as a never expire date, logic allowing deletion of data after a specified date may fail to protect data that should be maintained forever.
1999-10-01 (Friday)	This is the first day of the U.S. Government Fiscal Year 2000.
1999-12-31 (Friday)	The last day that can be represented in standard 6-digit date format without Year 2000 rollover risk. This date is sometimes used to trigger special logic. It must be established that the system is able to distinguish between a regular end-of-year 1999 date a special meaning date.

2000-01-01 (Saturday)	The first day of the Year 2000, several issues are related to this date: • A system with a day-of-week function based on 6 digit dates may change from Friday 1999-12-31 to Monday 2000-01-01 at Year 2000 rollover. 1900-01-01 was a Monday. • There is a possibility that the date will be misinterpreted as 1900-01-01. • System date counters may increment to erroneous dates like 19100-01-01. • Parsing functions may misinterpret dates entered with one or both leading zeroes omitted.
2000-01-03 (Monday)	This may be the first business day of the Year 2000. Certain business software calculates using proper business dates and days.
2000-01-07 (Friday)	This is the first Friday of the Year 2000. Paycheck calculations may be affected.
2000-01-17 (Monday)	This is the first Monday holiday in the Year 2000. Since this holiday is always on a Monday, a day of the week calculation may be required to identify this date as a holiday.
2000-02-28 (Monday)	This date is not expected to cause any specific Year 2000 errors. Its relevance to testing is that it should be used as a start date in testing the system's ability to increment to 2000-02-29.
2000-02-29 (Tuesday)	The Year 2000 is a leap year. Program logic used to identify leap years may be incomplete. This could cause date processing errors for the remainder of the year.
2000-02-30 (Non-existent)	This day does not exist. Date functions should continue to recognize this as an invalid date.
2000-03-01 (Wednesday)	This is the first day after leap year day. The date calculations that transition from the last day of leap year February to the first day of March could fail.
2000-03-31 (Friday)	This is the last day of the last month in the first quarter of the first year in the Year 2000. Quarter-end dates are significant in business and financial applications.
2000-04-17 (Monday)	Primary U.S. Income Tax due date in the Year 2000.
2000-04-30 (Sunday)	This is the first month-end that coincides with a weekend in the Year 2000.

2000-09-29 (Friday)	Last business day of the third quarter in the Year 2000.
2000-09-30 (Saturday)	This is the last day of the government fiscal year and last day of the third quarter of the Year 2000.
2000-10-01 (Sunday)	This is the first 7-digit date with a 2-digit month values. Parsing functions may need to be modified to allow for new date formats and wider range of date-data during the remediation process.
2000-10-10 (Tuesday)	This is the first date, after rollover that must be represented as an 8-digit date. Parsing function may fail when the numbers of digits changes.
2000-12-31 (Sunday)	The last day of the Second Millennium of the Gregorian calendar. The ordinal date 1900-365 was the last day of 1900. Since 2000 is a leap year, its last day is 2000-366.
2001-01-01 (Monday)	This is the first day of the third Millennium on the Gregorian Calendar. There is a possibility of errors in computing the day of the week.
2004-02-29 (Sunday)	First leap day after Year 2000 rollover not affected by a century or millennium transition.
2004-12-31 (Friday)	This date can be used to determine if normal leap years are recognized by an ordinal date system.

(The above was taken from IEEE P2000.2 Draft Recommended Practice for Information Technology: Year 2000 Test Methods, 1998, p. 28-31)

The dates listed above cover a wide array of possible mishaps when using date functions. The reason for each date's importance is not discussed due to space constraints, but when testing, it is important to check the software against all applicable dates. Test cases should include dates that are relevant to the type of system being tested. In most cases the majority of these dates will not need to be tested, as they are irrelevant system to functionality.

Case Study: Reactivity Measurement Analysis System

The following section will show how the preceding principles for Year 2000 testing are applied to a specific piece of software. The RMAS software from Framatome Technologies is a data acquisition system used for nuclear power plant startup. This software can log up to 16 inputs, which can then be viewed or analyzed. The RMAS software works in conjunction with a Reactimeter, which contains the Programmable Logic code, to read in the inputs.

RMAS system components.

It was crucial to analyze each component that made up the full RMAS system. Since the software is a commercial product, accompanying software (i.e. Windows) and hardware had to first be evaluated "to assure that potential influences on RMAS had been identified" (RMAS Year 2000 Compliance Test Procedure, 1998, p. 6). This was done by obtaining Year 2000 compliancy statements from vendors, which was usually done via the Internet. Once this was accomplished, the software was ready to be tested on a platform that was Year 2000 compliant.

The RMAS software was analyzed to determine what exactly needed to be tested to ensure Year 2000 compliancy. The RMAS software package consists of two main parts that were tested for Year 2000 compliancy:

1. Reactimeter Programmable Logic Control (PLC) code.

2. RMAS-4 software, including:

   - Reactimeter Interface (RI) software.
   - Operator Interface (OI) view node software.
   - RMAS Analysis programs. (RMAS Year 2000 Test Report, 1999, p. 4)

Since the RMAS software is conventionally systematically checked and verified for full functionality at each release, the test results from Year 2000 testing could be compared to expected values. The determined acceptance criteria for the above components were that it should "provide the correct time and date functions and calculations, data display, and data storage and retrieval for the time period of the test" (RMAS Year 2000 Compliance Test Procedure, 1998, p. 5). There should "be no time or date related difference between the results obtained during the baseline time period and the results obtained during the rest of the time periods" (RMAS Year 2000 Compliance Test Procedure, 1998, p. 5).

Dates to test.

Once it was determined what parts of the software were to be verified, then the specific dates to be used were determined. The specific use of the system was taken into consideration to decide which dates should be tested. All dates that were deemed critical to the correct operation of RMAS were tested, producing the following list of dates.

Pre-Millennium Periods:
From 02/28/1996 at 23:48:00 to 02/29/1996 at 00:18:00 (This time period is a leap day).
From 09/08/1999 at 23:48:00 to 09/09/1999 at 00:18:00 (This time period is

defined as the baseline time period).
From 09/09/1999 at 11:48:00 to 09/09/1999 at 12:18:00.
Millennium and Post-Millennium Time Periods:
From 12/31/1999 at 23:48:00 to 01/01/2000 at 00:18:00
From 09/08/2001 at 23:48:00 to 09/09/20001 at 00:18:00.
From 02/28/2000 at 23:48:00 to 02/29/2000 at 00:18:00.
From 02/29/2000 at 23:48:00 to 03/01/2000 at 00:18:00 (This is a leap day check).
From 02/28/2004 at 23:48:00 to 02/29/2004 at 00:18:00.
Non-existent Dates for File Imports
From 02/28/2000 at 23:48:00 to 02/29/2000 at 00:18:00.
From 02/29/2000 at 23:48:00 to 02/30/2000 at 00:18:00
From 04/30/2000 at 23:48:00 to 04/31/2000 at 00:18:00.

(RMAS Year 2000 Test Summary Report, 1999, p. 4-5)

Test conditions.

The test configuration consisted of three computers connected to the test Reactimeter with all sixteen analog inputs connected to voltage sources. The test computers used Windows 95 and each had a color printer connected. The PLC clock and the three computers had their clocks set to the beginning of the time period to be tested. The times were synchronized as closely as possible and the time periods were tested chronologically. The voltage supplies were initiated and run for a thirty-minute time period. This defined length of timed allowed the RMAS software to collect a sufficient

amount of data to analyze. All the data received during these time periods were archived for future reference and use in documentation.

Test data sheets.

The following three data sheets were used to carry out the testing that was involved for the various aspects of the testing process. Each sheet was used to verify different characteristics of the software. Each data sheet was completed for each of the dates tested, as mentioned previously.

YEAR 2000 TEST DATA SHEET – PRE TEST

CSV File: _____ Date: _____

Software Version: _____ Power Source: _____

Operating System: _____ Wave Generator: _____

Computer: _____ Reviewed by:

Test Date __/__/____ __:__ to __/__/____ __:__

		Date:	Verified By:
4.1.1	Hardware		
4.1.2	Operating System		
4.1.3	InTouch Software		
4.1.4	RMAS Application Software		
4.1.5	Inputs		
4.1.6	PLC		

YEAR 2000 TEST DATA SHEET – LIVE INPUTS

Reactimeter: _____ Date: _____

Software Version: _____ Power Source: _____
Operating System: _____ Wave Generator: _____
Computer: _____

Reviewed by:

Test Date

__/__/____ __:__ to __/__/____ __:__

Date:	Verified By:

INTOUCH TESTING

RMAS Runtime Window

Chart		

Reactimeter Interface

Flux & Reactivity 1		
Flux & Reactivity 2		
Process		
I/O Review		
Internal Test		
Historian		
Historian Download		
Historian Statistics		
Log File Dialog		
HDMerge		

	Date:	Filename:	Verified By:
RMAS Application Software			
Physics Test Manual			
Sensible Heat			
Chart			
Analysis			
Reactimeter Checkout			
Chart			
Analysis			
AROCB			
Chart			
Analysis			
Temperature Coefficient			
Chart			
Analysis			

YEAR 2000 TEST DATA SHEET – SIMULATED INPUTS

CSV File: _____ Date: _____

Software Version: _____ Power Source: _____

Operating System: _____ Wave Generator: _____

Computer: _____ Reviewed by:

Test Date __/__/____ __:__ to __/__/____ __:__

		Date:	Verified By:
Sensible Heat	Chart		
	Analysis		
Reactimeter Checkout	Chart		
	Analysis		
All Rods Out	Chart		
	Analysis		
Temperature Coefficient	Chart		
	Analysis		
Rod Worth	Chart		
	Analysis		

Conclusion

As software life cycles are examined there is no disputing that maintenance is an important part of the software life cycle. Because testing is also important to software development, and a majority of development cost is spent in the maintenance phase, then it is quite apparent that testing software in the maintenance step is crucial to the development of software. The principles of testing computer software were examined and specifically its implementation in the maintenance phase. These principles were then applied to the Year 2000 problem to demonstrate how testing took place to verify the compliancy of software. Since the Year 2000 problem was a good example of testing software in the maintenance phase, a case study was done on the RMAS software from Framatome Technologies. The procedure used to verify this software was examined to show how the testing principles discussed earlier were specifically applied to this software.

References

AMI. (1999). Desktop pc's and the year 2000 problem: Frequently asked questions. http://www.amibios.com/y2k/y2k_faq.html.

Behforooz, B., & Hudson, H., J. (1996). Software Engineering Fundamentals, 1st ed., 300-331.

Bechtel, B., (1998). The year 2000: No big deal, or apocalypse never. http://devworld.apple.com/mkt/informed/appledirections/Sep96/year2000.html.

Commonwealth of Massachusetts, Information Technology Division. (1998). PC bios and the year 2000. http://www.state.ma.us/y2k/pccompliance/whtppr-bios.html.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (1991) Fundamentals of Software Engineering, 1st ed., 24-26.

IEEE. (1998). IEEE draft recommended practice for information technology: year 2000 test methods, Draft 12.

Kaner, C., Falk, J., & Nguyen, H., Q. (1993). Testing computer software.

Kaplan, A., (1997). Explaining the year 2000 problem in personal computers. http://www.y2ktimebomb.com/computech/issues/pcs9736.htm.

Linux. (1998). Does Linux OS suffer? http://www.linux.org/ help/beginner/year2000.html.

Macintosh. (1998). The Mac OS and the Year 2000: Approaching the New Millennium. http://product.info.apple.com/pr/letters/1997/961210. pr.ltrs.macros2000.html.

Microsoft. (1998). Microsoft year 2000: MS-DOS. http://www.microsoft.

com/technet/topics/year2k/product/user_view6839en.htm.

Microsoft. (1998). Microsoft year 2000: Product guide. http://www.microsoft.com/technet/topics/year2k/y2kcomply/y2kcomply.htm.

Microsoft. (1998). Microsoft year 2000: Test criteria. http://www.microsoft.com/technet/topics/year2k/product/criteria.htm.

Microsoft. (1998). Microsoft year 2000: Windows 95. http://www.microsoft.com/technet/topics/year2k/product/user_view7450en.htm.

MITRE Y2K Team. (1999). Real-time clock and CMOS. http://www.mitre.org/tehcnology/cots/rtc.html.

MITRE Y2K Team. (1999). Y2K: The basic input/output system (BIOS). http://www.mitre.org/tehcnology/cots/pcbios.html.

Newice. (1999). Newice: Year 2000. http://www.pvn.com/newice/year2000d.html.

RMAS Year 2000 compliance test procedure. (1998). FTI Document 51-5001859-001.

RMAS year 2000 Test report. (1999). FTI Document 51-5002496-00.

RMAS year 2000 Test summary report. (1999). FTI Document 51-5003215-00.

Schach, S., R. (1996). Classical and Object-Oriented Software Engineering, 1st ed., 408-463.

Year 2000 compliance statement of some distributions. (1998). Http://li.luga.or.at/resources/year2000.html.