Fault Tolerance and Reliability in Software

M. Rzucidlo
Penn State Great Valley
School of Graduate Professional Studies
30 East Swedesford Road, Malvern PA 19355
mikerz@yahoo.com

Abstract

Over the past few years, the consumer electronics industry has grown considerably and consumer's expectations have grown with it. It is assumed that cell phones, PDAs, DVD players, etc will not fail and are fault tolerant. These expectations are unlike that of the PC where a few hours of downtime a month is normal. For over 30 years, research has been performed on reliability computing, but has moved very slowly to mature. Now, due to a grant, NASA and Carnegie Mellon University have formed the High Dependability Computing Consortium (HDCC) along with 12 information technology companies to eliminate computing failures critical to the welfare of society. Companies such as Microsoft have recently faced a series of highly publicized incidents caused by typographical errors to major design defects like buffer overflows in their software. Errors like these could have been avoided had there been a focus to remove faults in the design at the beginning of the development cycle and not after shipping a product.

In this paper, a software black box design is discussed as a way to uncover details about a software crash. Additionally, various fault-tolerant methods are introduced such as N-Version Programming (NVP), comparison approach, pipeline approach, software rejuvenation software reconfiguration and a new method called fault-injection software test (FIST). Experiments were performed to validate the software black box technique discussed in this paper.

Keywords

Software black box, software fault tolerance, N-Version Programming, Comparison approach, Pipeline Approach, Software Rejuvenation, Software Reconfiguration, Fault-injection software test

1 INTRODUCTION

The popularity of consumer electronics has grown considerably automating more of the consumers' life everyday. Additionally, the consumer has much higher expectations in these consumer electronics than that of the PC. It is expected that they won't fail and are reliable. Two paradigms to allow software to become more reliable are through the use of a software black box and software fault-tolerance.

A software black box can record the events leading up to the software crash for analysis to reconstruct later. The black box in aviation has been an important factor that has given the FAA data to improve safety on flights. Following the tragic terrorist attacks on September 11, 2001 that killed over 3000 people from different countries all over the world, the black box became one of the only pieces of information that the NTSB could use to recreate the events leading up to the four different tragic and fatal crashes. With the exception of the cell phone calls of passengers to their loved ones, there is no way to put the pieces of the puzzle together that occurred on the flights that sunny Monday morning without the help of black boxes.

As consumer electronics become more popular in everyday life, reliability is no longer a luxury feature. The software engineering discipline has not matured quickly enough lacking the emphasis of improving the software life cycle standards. Important questions where the use of a black box could have been useful are:

- What if you were held accountable according to the law because your application crashed as a result of a malicious hacker? Laws for this type of an issue are being recommended by the National Academy of Sciences.
- What if the black box mechanism became able to detect problems based on past data being able to prevent future disasters?

- What if Microsoft had an internal software black box mechanism to record the user's session to analyze what actually happened? Could it be possible to release a service pack for Microsoft products in a more time critical manner? Or would this be a violation in the consumers' privacy? Could law enforcement gain access to this software black box under any circumstances?

Currently, NASA and Carnegie Mellon University have teamed up with at least a dozen information technology companies to form the High Dependability Computing Consortium. The mission of this group is to eliminate computing failures critical to the welfare of society and to create software reliability standards so that software is designed such that it never fails according to the HDCC. Critical applications that would utilize these concepts are air-traffic control, health systems and space exploration. However while this is a good idea, a concept like this will take time to adapt once in place, as the architecture of the software itself would need to be redesigned in many cases.

Software black boxes on the other hand can be developed and integrated into products today. They can be used to diagnose and fix problems much more quickly than spending many hours trying to reproduce the problem and determining the state of the software at the time of the crash in a lab. According to Pogue (2002), the increase in the sale of DVD players in the market today demonstrates that consumers will return their player based on one factor alone: Reliability. As prices drop for these electronics, the market becomes even more competitive and manufacturers rush units out of the factory quicker than ever with as many features possible with less quality assurance.

However through fault tolerance, many of the design and implementation flaws that were introduced in the original software could have been removed. According to Boehm and Basili (2001), one of the goals of developing any software is to remove as many defects as possible although this is difficult. Fault tolerant methods such as n-version programming (NVP), comparison approach, pipeline approach, software rejuvenation, software reconfiguration and a new method called fault-injection software test (FIST) are used for this purpose. Also, important questions where the use of fault tolerance could have been useful are:

- Had fault tolerance been introduced into Windows NT, could the widely publicized software failure causing the jet propulsion system to fail on the USS Yorktown requiring it to be towed back to shore not have happened or had been less severe?
- If Nokia had designed fault tolerance into their software, could their product line of phones have been protected against a security glitch that allows a hacker to send a text message with an incomplete header rendering the phone unusable afterwards?
- Would all the java virtual machines run everywhere still have had the flaw that made them vulnerable to hacker attacks if fault-tolerance methods were used when it was designed? How costly would the damages have been had it not been caught before hackers found it?

Software applications such as the ones mentioned above require that even during stressful conditions or improper usage that the performance is not degraded and is robust. Problems will continue to occur until companies are held liable for the software they produce and held liable for the damages caused such as security holes according to a group of U.S. scientists (Bowman, 2002).

According to Richard and Tu (1998), ideally fault-tolerant software is transparent, portable to any platform and easy to use. However, this is not the case. The reality is that with the research performed developing theories to support fault tolerance, many solutions are impractical.

If fault tolerance was introduced at the OS level, applications could take advantage of this feature naturally. Many OSs such as Windows (any version) behave in ways that were not envisioned by the architects. This has created obstacles for implementing fault-tolerant methods at the application level. By developing a stable product, it not only cuts costs to fix bugs, but can also have the potential to increase market share as consumer confidence is high wanting to buy that product.
Fault tolerant methods such as n-version programming, comparison approach and pipeline are introduced in Section 3 later in this paper. The strengths about these methods are that they are used throughout the software development life cycle filtering out the bugs with each step in the process. Software rejuvenation and reconfiguration are more recent methods that are also introduced. Their strengths are that they are always present protecting against faults and allowing the software to be efficient and have a more flexible

recovery respectively. Finally the last method discussed is the fault-injection software test that executes test patterns at initialization for previously known detrimental faults.

2 SOFTWARE BLACK BOX

The software black box is an important model in software development to recover data to prevent future disasters. First the discussion begins with the software black box discussed by Munson and Ebaum (1998). The behavior model explains how the software behaves and how it will be modeled. Next the description of how the recorder and decoder work will follow. Finally, some experiments were performed using a simulation testing the black box technique.

2.1 BEHAVIOR MODEL

When software fails, it is usually because a module within the software system has failed. The objective of a software black box is to recreate the sequence of events that lead up to this failure. A module consists of one or more functionalities that it provides. When one of the functionalities fails, this makes the software behave in such a way that it normally would not to so such as crashing. Based on a particular design, not all modules will execute with the same probability when a user exercises a specific functionality. This software behavior describes the basis of the software black box and the correlation between modules and functionalities.

Based on the requirements to fulfill the needs of the customer, functionalities can be derived from them. In a software system **S**, functionalities in **F** can be assigned to specific modules m which is an element of the set of program modules. In Table 1, function f_2 has the modules $\{m_1, m_2, m_3, m_7\}$ implemented. If functionality f is expressed in module m, then it can be said that for a set of relations, ASSIGNS $\subseteq$ **F** x **M** such that if true, ASSIGNS(f,m) $\rightarrow$ 1. Otherwise ASSIGNS(f,m) $\rightarrow$ 0 if false. When a specific functionality is executed, the control is passed to a different module with the assumption that after some number of software calls that it will return to the original module. The behavior can be modeled by a call graph displaying the transitions from one module to another that can be referred to as an epoch. Each epoch has an epoch number that is incrementally changed when one module transfers control to another.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
f_1	T	T		T			
f_2	T	T	T		T		T
f_3	T		T			T	

Table 1: Functionalities and Modules

To gain a further understanding, a software system **S** has a set of all modules **M** and for each functionality $f \in$ **F**. There is a relation of c over **F** x **M** that defines the number of functionalities that may execute in a module $c(f,m)$. This can be defined into two different and distinct sets that are either unique (indispensable) or shared (potential). The unique set of modules $\mathbf{M}_u = \{m{:}\mathbf{M} \mid f \in \mathbf{F}, c(f,m) = 1\}$. The shared set of modules $\mathbf{M}_s = \{m{:}\mathbf{M} \mid f \in \mathbf{F}, c(f,m) > 1\}$. Another way of representing functionalities and modules that they might cause to be executed where $f \in$ **F**, there is a relation p over **F** x **M** that defines the proportion of execution events of module m when the system is executing the functionality $p(f,m)$. When $p(f,m) < 1$, then a module m may not execute when functionality f is invoked. The set of potentially involved modules is $\mathbf{M}_p^{(f)} = \{m{:}\mathbf{M}_F \mid \exists\, f \in \mathbf{F}, \text{ASSIGNS}(f,m) = 0 < p(f,m) < 1\}$.

Other modules might be indispensable where each time a function f is called, a distinct set of modules is called. The set of indispensably involved modules for a particular functionality f pertain to the set of those modules with the following property is $\mathbf{M}_i^{(f)} = \{m{:}\mathbf{M} \mid \forall f \in \mathbf{F}, \text{ASSIGNS}(f,m) \Rightarrow p(f,m) = 1\}$. For functionalities that have at least two modules or more where one of them is a element of the unique set of modules and not of the potentially involved modules can be defined as $f_i \in$ **F** if $f_i = \{ \forall\, m \in \mathbf{M}^{f_i}, \exists\, m_j \in \mathbf{M}_u{}^{mj} \in \mathbf{M}_i, \| \mathbf{M}^{f_i} \| > 1 \}$.

2.2 SOFTWARE BLACK BOX RECORDER

The software black box recorder stores system behavioral data throughout the use of the software providing useful information when a failure occurs. It stores the essential characteristics attempting to minimize perturbation while recording this information.

With the use of the software black box recorder, transitions of modules can be measured and observed. The execution of program functionalities result in the sequence of module transitions where the system can be described as modules and their interaction. When software is running, it passes from one module to the next passing the control of execution to the called module. Passing from one module to the next is considered a transition. Call graphs can be developed from these transitions graphically using an N x N matrix, where N represents the number of modules in a system. In Figure 1, an example of a call graph is shown and the adjacency matrix in Table 2 shows another representation of call graph. As can be seen, it can be derived that for each transition from m_i to m_j, the value in matrix a_{ij} is true (T).

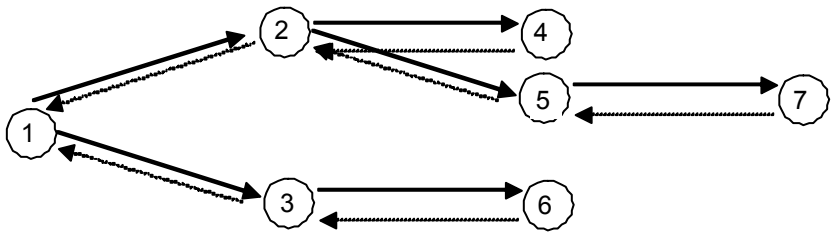

Figure 1: Call Graph Diagram

Modules	1	2	3	4	5	6	7
1		T	T				
2	T			T	T		
3	T					T	
4		T					
5		T					T
6			T				
7					T		

Table 2: Call Graph

Adjacency graphs account for adjacency and associated frequency too. When each module called, each transition is recorded in the matrix, incrementing that element in a transition frequency matrix. From this, another matrix can be derived. The transition probability matrix where the probability p_{ij} is represented from m_i to m_j. The marginal is the total number of observed transitions shown in Table 3. The probability is the likeliness that a transition will occur as shown in Table 4. The transition frequency and transition probability matrices indicate the number of observed transitions and probability, the sequence is missing in this data.

Modules	1	2	3	4	5	6	7	Marginals
1		2	4					6
2	2			6	6			14
3	4					3		7
4		6						6
5		6					1	7
6			3					3
7					1			1

Table 3: Transition Frequency Matrix

Modules	1	2	3	4	5	6	7	Probability
1		0.33	0.67					1
2	0.14			0.43	0.43			1
3	0.57					0.43		1
4		1.00						1
5		0.86					0.14	1
6			1.00					1
7					1.00			1

Table 4: Transition Probability Matrix

A sequence that could be generated during the execution of software can be represented by (m_1, m_2, m_3, m_4, m_5, … m_i) where m_i is the execution of the most recent module, m_{i+1} the next most recent and so on. From this, a transition can be represented by a sequential pair of elements t_{23} = (m_2, m_3) and is called a system epoch. An example of a system of epochs from the call graph in Figure 1 could be <4,2,1,2,5,2,1,2>. Each time a new module is entered, a new element is added to the sequence.

The architecture of software black box recorder has a front end, kernel and back end. The front end gets input from target system and validates the input data. The format of input data is

`SBBR(module-ID, operation).`

In the SBBR function, the module identifier (`module-ID`) is a long unsigned integer representing the operation (`operation`) as an enter (1) or an exit (0). The kernel derives and stores all information and derives the transition that occurred with an internal call stack. Each time a new module is entered by system, the following occurs:

- New transition *t* is derived
- Frequency matrix is updated
- Top of stack is incremented by one
- The module identifier is added to top of call stack information is added to the tail of the transition sequence.

Each time some module is exited, that module at stack [top-1] is added at the tail of transition and top of stack is decremented by one. The back end provides a mechanism to operate the software black box recorder.

The recorder contains three primary data structures, transition matrix, frequency matrix, call stack used to assist in updates of the first two data structures. The transition sequence is stored in a queue containing the last *n* transitions in sequence. Using this method, the prior transitions disappear and the latest software behavior is reflected. The transition frequency matrix complements the transition sequence by reflecting the behavior of the system since the beginning of execution. The transition frequency matrix and transition probability matrix can be derived which indicates the probability the different paths the system may have taken. The back end of the software black box decoder dispatches the data stored in the recorder to safety in the event of a failure and allows control and operation of the software black box recorder to perform activities such as activation/deactivation, download data, restart and resize the sequence size.

2.3 SOFTWARE BLACK BOX DECODER

Recovery begins after the system has failed and the software black box has been recovered. The software black box decoder generates possible functional scenarios, quantifies the normality of data and allows further analysis. The first stage is to generate functional scenarios, but many scenarios may be produced due to noise making this process non-deterministic. The first source of noise is due to the difficulty in mapping functionalities to modules. Many modules are common to much functionality which makes the decoding process more difficult because a module cannot be mapped to one functionality. The second source of noise is due to failures occurring during the execution of a shared module, in which case it is not obvious where the functionality was currently being executed.

The data used to generate functional scenarios are from the recorder and the mapping between the modules and functionalities. The generation process attempts to map the modules in the transition or modular

sequence to functionalities. Mapping of each module found in the sequence to the functionalities produces a sequence of functionalities. Unique modules present no difficulty and can first be mapped and fill in some of the functional sequence cells. For example, consider the following problem below using the information from Figure 1 and Table 2.

Module Sequence	4	2	1	2	5	2	1	2
Intermediate	u	s	s	s	u	u	s	s
Functionality Sequence	1	?	?	?	2	?	?	?

Where the empty cells are located are the gaps in the sequence. These gaps are broken into three categories. Group one are the gaps where there are known functionalities on both sides. Group two are the gaps where the last known functionality is on left. Group three are the gaps where the last know functionality is on the right.

1	-	-	-	2	-	-	-

Referring to Table 2, the specific functionalities of the last two modules run before the crash are shown below.

Module Sequence	4	2	1	2	7	2	1	2
Functionality Sequence	2	-	-	-	5	-	▼	▼
Candidate Functionalities							1	1
							2	
							3	

The next step is to generate Table 5 with the possible scenarios. Some scenarios can be easily ruled out since the modules introduce different functionalities therefore contradicting the last identified functionality. PS 1, PS 3 and PS 4 demonstrate this case as shown in Table 5.

Corresponding Modules	M1	M3
~~PS 1~~	~~1~~	~~1~~
PS 2	2	2
~~PS 3~~	~~3~~	~~1~~
~~PS 4~~	~~1~~	~~2~~
PS 5	2	1
PS 6	3	2

Table 5: Possible Scenarios

Now that the scenarios have been reduced to only the final scenarios by eliminating the cases that do not exist, duplicate scenarios can also be refined. For example PS 2 and PS 6 are equivalent when consecutive instances of the same functionality are joined.

Common	Variants	Complete	Collapsed
	PS 2: 2-2	1-2-2-2	1-2
1-2	PS 5: 2-1	1-2-2-1	1-2-1
	PS 6: 3-2	1-2-3-1	1-2-3-1

Table 6: Final Scenarios

The software black box recorder helps to determine whether the last epochs were abnormal by using the transition sequence that was observed. The software black box decoder helps determine whether the steady state of the system is equal to this transition sequence that was recorded before the failure. If it does not equal the steady state, then it can be said that the behavior was abnormal. The functional scenarios, transition and probability matrices assist in defining the steady state. The transition sequence has the last *n* epochs before the failure which can be used to derive a new frequency matrix. Now this frequency matrix

can then be compared to the same population as the steady state data. It can be validated using a chi-square distribution with *n*-1 degrees of freedom as shown:

$$\sum_{i=1}^{n} \frac{(f_i^{seq} - m\_p_i)^2}{m\_p_i} < \chi_\gamma^2$$

where χ_γ^2 represent the 100% point, f_i^{seq} is the sequence executed during the last n epochs and m_p_i is the transition probability matrix during the last *n* epochs.

2.4 SOFTWARE BLACK BOX RESULTS USING A SIMULATION

A simulation was developed using C++ to demonstrate the simple concepts of a software black box recorder. A very simple "Hello World"-type program was simulated passing control of execution to other modules as described to utilize this black box. The same sequence <4,2,1,2,5,2,1,2> was used to validate the approach.

The biggest problem that hindered this simulation was that there was no instrumentation tool to insert the hooks into the program using the black box. The line of code that needed to be inserted in every module was `SBBR(module-ID, operation)`. Although, the program was very small in size, if it had been a larger program in lines of code, then manually inserting hooks would not be a viable solution.

Once past this obstacle, the black box ran smoothly. It recorded the transitions, entering and exiting and updating the matrices. It proved to be a useful concept in this example for recording. However due to time constraints, developing a software black box decoder was not possible. Manually, analysis was done to carry out the decoding of the recorded information. Again, the size of the program was small enough that it was possible to decode in a short amount of time.

3 SOFTWARE FAULT TOLERANCE

According to Webopedia, software fault tolerance can be defined as the ability of a software system responding to an unexpected failure gracefully. Software fault tolerance can be divided into two categories: fault-tolerant methods used throughout the software development life cycle and fault-tolerant methods used to refresh software internals during operation. A fault is considered any defect in the program that executed during a particular set of circumstances resulting in a failure according to Yurcik and Doss (2001). Unlike a fault, a bug is a euphemism for an error in a program according to Schach (1997) which is an unacceptable term in modern software engineering because the responsibility for the error was transferred to the bug and not the programmer who made the error. A software failure is an event that occurred while the software was subjected to an input condition due to the presence of one or more faults which resulted in the output to be different than the required output according the to design specifications.

Faults can be classified as two different types: independent and related according to Grnarov (1997). Independent faults are faults that occur as a result of unclear requirements used to generate the common specification. Independent faults are also very distinct errors where a specific set of circumstances or sequence of steps will allow a fault to occur every time. Related faults are faults that have dependencies in separate designs and implementations. Related faults can be subdivided among their origin: among several variants or among one or several variants and the decider (if NVP is used). Related faults are similar errors where different sets of circumstances may allow a fault to occur.

Faults can be further classified into a number of different categories as to why they occurred such as design faults, compiler faults, language complexity faults, implementation faults, hardware faults, and so on. A design fault would be a result of a different design used for each version. A compiler fault would be a result of using a different compiler for each version. A language complexity fault would be a result of using different languages. An implementation fault would be a result of the programmer interpreting the specification differently than originally intended. Most faults are a result of the common specification being based on the requirements or dependencies in separate designs and implementations.

3.1 N-VERSION PROGRAMMING

N-version programming (NVP) is the idea of having many different versions of software developed based on the same set of requirements. In using this method, the different versions of software feed their output to a decider with the goal of fault tolerance. NVP has been criticized for where its faults are created according to *Software Fault Tolerance* (ed. Lyu, 1995). Related faults can be introduced between the specification of NVP and the common specification for the versions, between any of the two versions, and between any of the specifications including the specification used for the decider algorithm. Independent faults can be introduced between the common specification and any of the versions and between the specification of the decider and the decider. In Figure 2 from Hilford, Lyu, Cukic, Jamoussi, and Bastani (1997), the example shows NVP with three different versions of software based on one common specification. Based on this common specification, the decider determines the faults. The associated costs of using this method are high along with the risk of delivering software on time being high too according to Kovalev and Grosspietsch (2000).

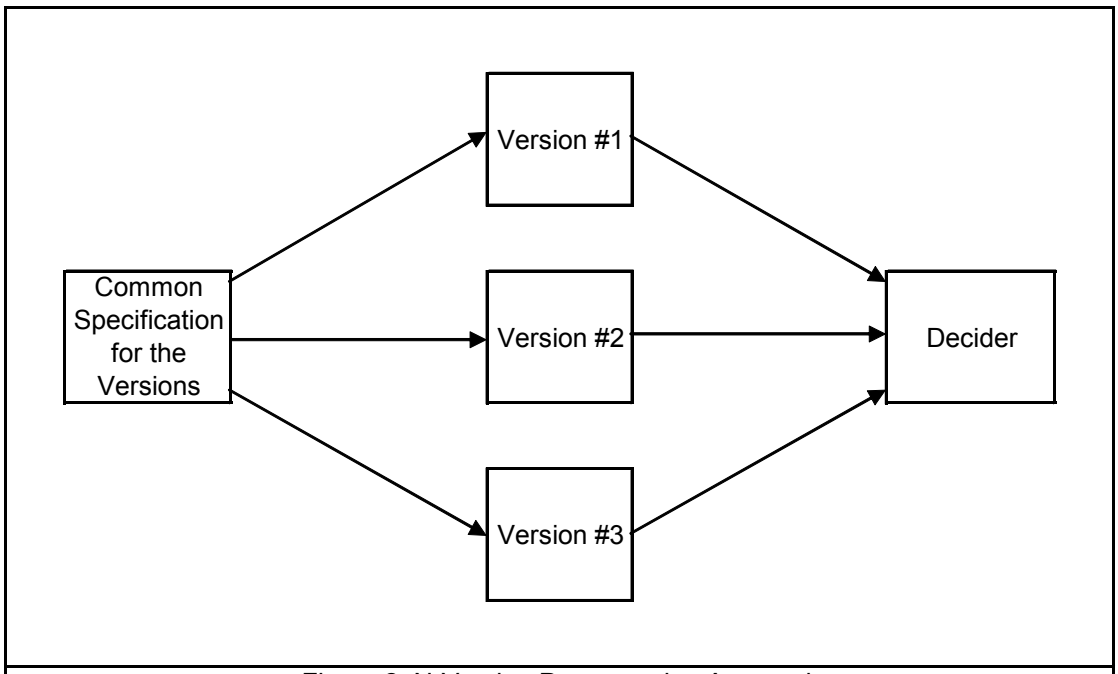

Figure 2: N-Version Programming Approach

3.2 COMPARISON APPROACH

Developing with the comparison approach consists of N-teams of one or more people developing software in parallel (resembling NVP). Each team goes through software life cycle and at the end, the results are compared. All versions are compared resulting in the production of only one final version. With different versions being compared, this exposes many independent faults to build a very reliable final version. However, the cost associated with this method is high and putting together the final version incurs large costs and possible schedule overruns according to Hilford, Lyu, Cukic, Jamoussi, and Bastani (1997). In Figure 3. the example shows the comparison approach with three different version of software from a common specification. At the end of the software being developed, the software is only then compared producing a final version.

Figure 3: Comparison Approach

3.3 PIPELINE APPROACH

Developing with the pipeline approach according to Hilford, Lyu, Cukic, Jamoussi, and Bastani (1997) there is a trade-off between high cost and level of diversity. With this level of diversity, the cost associated with this method is probably the lowest and has the best results. All members of a team concentrate on producing single reliable version of software. During the design and coding phase, there is one team of programmers where each are assigned different parts of the specification with clearly defined interfaces. During the code review phase, N-review teams with two people each review the code. The final program is submitted for acceptance testing and debugged by someone who was on a review team. One fault of this method is that each time the code is reviewed that the code should be reviewed again verifying that the changes in the code were integrated. In Figure 4, the example shows five teams that are in constant communication with each other throughout the process. After the design and coding phases, three teams review each other's code verifying that it is compliant to the specification.

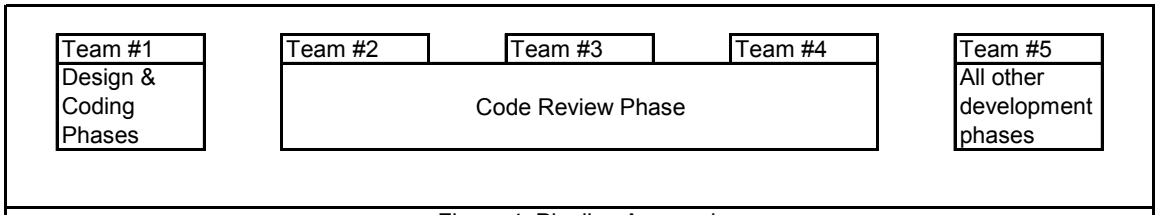

Figure 4: Pipeline Approach

3.4 SOFTWARE REJUVENATION

Incorporating software rejuvenation techniques into software helps protect against transient faults, environmental disruptions, human errors, and malicious attacks that could occur. These could be caused by memory leaks, memory fragmentation, memory bloating, missing scheduling deadlines, broken pointers, poor register use, build-up of numerical round-off errors, buffer overflows. System failures happen due to deteriorating operating system resources, unreleased file locks and data corruption. According to Yurcik and Doss (July-August 2001), Yennun Huang and colleagues introduced the concept of software rejuvenation in 1995. Software rejuvenation is a proactive approach to stop executing software periodically, cleaning internal states and then restarting software.

Garbage collection, memory defragmentation, flushing operating system kernel tables and reinitializing internal data structures could clean up internal states. However, it does not remove bugs resulting from software aging, but rather prevents them from manifesting themselves into unpredictable whole system failures. Periodic software rejuvenation limits the state space in the execution domain and transforms a nonstationary random process into a stationary process that can be predicted and avoided according to Doshi, Goseva-Popstojanova and Trivedi (2000).

Rejuvenation incurs immediate overhead in terms of some services being temporarily unavailable. The idea is to prevent more lengthy failures from occurring in the future. While software decay occurs in incremental steps through change processes that humans initiate, software aging occurs through underlying operating system resource management in response to dynamic events and a varying load over time according to Trivedi, Vaidyanathan and Goseva-Popstojanova (2000). Two techniques for optimal rejuvenation are measurement-based and modeling-based. The measurement based technique estimates rejuvenation timing based on system resource control metrics. The modeling based uses mathematical models and simulation to estimate rejuvenation timing based on predicted performance.

3.5 SOFTWARE RECONFIGURATION

Incorporating software reconfiguration techniques into software provides more flexible and efficient recovery, independent of knowledge about the underlying failure. Software reconfiguration is a reactive approach to achieving fault-tolerant software. The system is reconfigured after detecting a failure where redundancy is the primary tool used. For software, redundancy is in three different dimensions: software redundancy, time redundancy and information redundancy. Software redundancy manifests as independently written programs performing the same task executing in parallel. Time redundancy appears as repetitively executing the same program to check consistent outputs. Information redundancy has

redundancy bits that help detect and correct errors in messages and outputs. The fault tolerance challenge is to minimize redundancy, which reduces cost and complexity. Software reconfiguration can use redundant resources for real-time recovery while dynamically considering a large number of factors like services and load. So the system does not have to crash to be gracefully recovered.

3.6 PROPOSED NEW METHOD – FAULT-INJECTED SOFTWARE TEST (FIST)

As more consumer electronics become internet-enabled, the need for updating software dynamically without the users' effort will become necessary. However, there is a risk involved with updating the software. The problem could be that the downloaded software was not tested on the specific hardware. There is a need to have a fault injection software test after restarting the device to introduce previously known faults in the software running that was just downloaded on the consumer electronic. This method can be considered analogous to LFSR as a method for hardware fault tolerance. An LFSR is set to cycle rapidly through a large number of states. These states, whose choice and order depend on the design parameters to define the test patterns as mentioned by Al-Asaad, Murray and Hayes (Oct.-Dec. 1998). The test patterns used for the FIST technique could be pseudorandom patterns or previously known faults. Similarly after downloading the new software, the software cycles though many states testing for bad outputs. Once the software has been acknowledged to not encounter any faults, a confirmation bit can be set high into its nonvolatile memory until the next update or restart.

This method could be proven to be useful especially when only a block of memory for the main software starts at a specific address where it is written. This test is also good to test for transient faults, environmental disruptions, human errors, and malicious attacks like software rejuvenation which are likely not to have been encountered in the lab. To the best of the authors' knowledge, this approach has not been proposed before. It is similar to software reconfiguration where information redundancy uses redundancy bits to help detect and correct errors in messages and outputs. This proposed method does not correct errors in messages or outputs, but rather confirms that a version of software will run on a hardware platform pending any faults that were not encountered during initial tests in the lab. If the software has been verified not to run successfully on the hardware platform where the confirmation bit is set low, then the desired result would be to revert to the previous running version of software.

The associated cost of using this method is considered low assuming that the new version of software has been thoroughly tested. However if the new version of software has not been tested well, then the associated cost is extremely high because not only is there a risk that infinitely many consumer electronics could be rendered useless until taken to a shop for repair, the user's data could potentially be lost, network bandwidth is wasted and the new version of software was not useful in any respect. Other aspects which may deter the use of this method is that cycling through the test patterns may add too much time at power-up of the device or that the test patterns are not sufficient. This is something that will need to be addressed in future experiments as discussed later in the Future Work and Conclusion section.

4 PREVIOUS WORK

Previously, this paper focused only on the software black box method. While this method is sufficient for existing software systems mainly, there was a need of methods for newly developed software systems. Fault-tolerant methods handle these scenarios. Additionally, MicroFIPA-OS was the lightweight agent platform chosen to carry out the software black box experiments, however it was found to be insufficient because it did not have any features for fault tolerance in its current implementation. JADE 2.5 was not chosen either for the same reasons.

5 FUTURE WORK AND CONCLUSION

This paper presents many different methods for increasing the reliability of software. Some fault-tolerant methods are more practical than others in implementation details, but it is important to consider the trade-off of cost versus diversity. It is important to consider all factors and requirements of the software before beginning the software life cycle. To gain further understanding and refine current methods, four groups of experiments could be performed: implementing a black box, fault tolerant methods used during the life cycle (NVP, comparison approach, and pipeline approach), fault-tolerant methods used to refresh the software internals (software rejuvenation and reconfiguration) and fault-tolerant methods to validate new software on existing hardware such as FIST.

Using a simple microcontroller such as an 8051 or a lightweight agent platform such as JADE or MicroFIPA-OS, the software black box could be implemented and further experiments could be done. This would validate the work of Munson and become a basis for more tests such as how filtering or higher-order statistical methods would be an option for analyzing the time series data.

Including fault-tolerant methodology designed for the use during the software life cycle into the new software engineering curriculum at Great Valley to stress the importance of these concepts would become a valuable resource to the student in their future classes and software careers. This information could be incorporated into SWENG 597: Advanced Software Engineering, IE 513: Real-Time Microcomputers or even CSE 411: Operating Systems. A more thorough understanding of fault-tolerant methodology also could be presented in its own course assuming the student has an understanding of the fundamentals of software engineering. This would allow students to work in small groups to carry out a small software project experiencing a full software life cycle (specification, design, implementation, testing and maintenance) based on the same set of requirements given out in the beginning. This could help refine existing fault-tolerant methods in practice and develop new ones. Through these efforts, the culmination of data gathered could be presented into another paper discussing the results of what problems occurred, where they occurred and why they occurred recommending a future path of fault tolerance and to help refine software engineering methods to produce more reliable results.

Software rejuvenation and reconfiguration could be incorporated into an 8051, JADE, MicroFIPA-OS or a small software project in a software engineering class. Key concepts for software rejuvenation to experiment with could be garbage collection, memory defragmentation, flushing operating system kernel tables and reinitializing internal data structures. Key concepts for software reconfiguration to experiment with could be software redundancy, time redundancy and information redundancy.

Investigating the proposed new method FIST would verify whether or not this technique proves to be effective. As a first step, it could be incorporated into software running on an 8051 or into JADE or MicroFIPA-OS. The initial test patterns could be pseudorandom and a small finite number of them. An algorithm could be defined more clearly optimizing it by the number of test patterns versus time. The shortest amount of time possible cycling through the most patterns would demonstrate how valid this theory is. As a second step, deterministic test patterns versus time could be a second step. Finally, the results could be presented into another paper demonstrating the strengths and weaknesses of this technique.

As software systems grow in complexity, reliability will become more of an issue. Software will continue to fail more often if the emphasis of a bug-free design in not stressed. With the current lack of software life cycle standards and immature software engineering discipline, it is certain nothing will change quickly. As an old software product has new features added, a software black box should be considered. As a new software product is designed, software fault-tolerant methods should be considered. Using these methods presented will help ensure reliable software.

6 REFERENCES

Elbaum, S.; Munson, J.C. (2000) “Investigating software failures with a software black box”, Aerospace Conference, Proceedings, IEEE, Volume: 4, Pages: 547-566 vol.4.

Hilford, V.; Lyu, M.R.; Cukic, B.; Jamoussi, A.; Bastani, F.B. (1997) “Diversity in the software development process”, Object-Oriented Real-Time Dependable Systems, Proceedings., Third International Workshop on, Pages: 129-136.

Yurcik, W.; Doss, D. (July-Aug. 2001) “Achieving fault-tolerant software with rejuvenation and reconfiguration”, IEEE Software, Volume: 18 Issue: 4, Pages: 48-52.

Gokhale, S.S.; Lyu, M.R.; Trivedi, K.S. (1997) “Reliability simulation of fault-tolerant software and systems”, Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on, Pages: 167-173.

Goseva-Popstojanova, K.; Grnarov, A. (1997) “Performability and reliability modeling of N version fault tolerant software in real time systems”, EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference, Pages: 532-539.

Al-Asaad, H.; Murray, B.T.; Hayes, J.P. (Oct.-Dec. 1998) “Online BIST for embedded systems”, IEEE Design & Test of Computers, Volume: 15 Issue: 4, Pages: 17-24.

HDCC, High Dependability Computing Consortium, www.hdcc.org

Pogue, D. (2/7/2002) **“**STATE OF THE ART; DVD Players Under $100: What Price A Bargain?” http://story.news.yahoo.com/news?tmpl=story&u=/nyt/20020207/tc_nyt/dvd_players_under__100__what_price_a_bargain_

Basili, V.R.; Boehm, B. (May 2001) “COTS-based systems top 10 list” Computer, Volume: 34 Issue: 5**,** Pages: 91-95.

Richard, G.G., III; Shengru Tu (1998) “On patterns for practical fault tolerant software in Java**”** Reliable Distributed Systems, Proceedings. Seventeenth IEEE Symposium on , 1998, Pages: 144-150.

Bowman, L. (1/23/2002) “Scientists: Fight flaws with laws”, ZDNN, http://techupdate.zdnet.com/techupdate/stories/main/0,14179,5102131,00.html

Schach, S. (1997) Software Engineering with Java. WCB/McGraw-Hill, USA.

Webopedia, http://webopedia.lycos.com/TERM/F/fault_tolerance.html

Dohi, T.; Goseva-Popstojanova, K.; Trivedi, K.S. (2000) “Analysis of Software Cost Models with Rejuvenation” High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000, Pages: 25-34.

Trivedi, K.; Vaidyanathan, K.; Goseva-Popstojanova, K. (2000) “Modelling and Analysis of Software Aging and Rejuvenation” Simulation Symposium, 2000. (SS 2000) Proceedings, Pages: 270-279.

Various authors (1995) Software Fault Tolerance (ed. Lyu, M.), John Wiley & Sons, Chichester, England.

7 DEDICATION

I want to dedicate this paper to my grandfather who died this past December (1925-2001) of Alzheimer’s disease.

8 BIOGRAPHY

Mike Rzucidlo received his Bachelor of Science in Electrical Engineering from Drexel University in 1997 and will receive his Master of Engineering in Engineering Science from Penn State University in May 2002. Upon graduating with his bachelors’ degree, he joined Unisys Corporation as a software engineer. There he worked with large-scale voice messaging systems developing support applications for speech data analysis as well as working in a team integrating natural language technology into telephony platforms. After working at Unisys for three years, he joined Ravisent Technologies, then ST Microelectronics in March 2001. Here he has been developing and testing software for consumer DVD players and next generation set-top boxes.