1
Introduction

When examining processes and software development environments currently being used, one may observe that, although there is a large number of tools and techniques available to develop software, most of the effort (specially code writing) is still essentially manual. Hence, there is a great chance of development errors due to human mistakes. Furthermore, even perfect software might fail due to hardware or platform failures or due to human usage errors [Brown et al, 2005]. Software failures are particularly critical in cases where a system requires a high level of dependability, as is the case of non-internet based applications like embedded systems, supervisory systems, process control systems, and also internet-based applications like e-commerce or e-banking systems.

Recovery oriented systems are built with the perspective that hardware failure, software defects and operation mistakes are facts to be coped with, not problems that could be completely prevented at development time [Fox, 2002; Brown et al, 2002]. The recovery oriented software axioms are the following:

1. It is impossible to build defect-free software, and if we would succeed in doing so we would not be able to know it.
2. It is not allowed to assume that software, even if perfect, will not be affected by external issues, such as hardware or platform failure, user interaction errors, or even environmental issues.
3. It is not allowed to assume that one can foresee all possible failures that software might display.
4. Some failures can be tolerated to some extent, as long as their consequences are assuredly below an acceptable threshold.

For the purpose of this work, a software defect, or simply *defect*, is a code fragment that, when exercised in a certain way, generates an error. An error is an unacceptable state with respect to the specification or the real world with which the software interacts or which it replicates or simulates. Errors may also be due to external sources, such as machine failure, operating system failure, or usage failures among many others. A failure is an error that has been observed by some means [Avizienis, 2004]. A defect or external error source, i.e. an exogenous error, is said to be encapsulated if it still remains in the artifact but is controlled in such a way that its consequences are acceptable. Examples are controls that observe and adequately handle transient failures.

It may be argued that exogenous errors that go undetected are also a form of software defect. As a simple example, take an input buffer. During normal use the buffer is never overrun, hence no error occurs. However, if due to some human error or intention an excessively long string is inserted, buffer overrun may occur, unless the software contains some form of overrun control. Not containing this control is clearly a defect.

It follows that the objective of software development must not just be to assure that it is free from defects (absolute defect prevention), but it should also encompass developing a system for which the risk of run-time failures and their consequences are acceptable. It is important to notice that the causes of the failures that might happen (either during development time or during usage) are unknown; otherwise the defects could have been removed or at least encapsulated. In response to a failure, it must be possible for a user to quickly resume his/her work [Fox, 2002; Brown et al, 2002]. This means that a recovery oriented system must minimize downtime time intervals that are due to failures. This is particularly important in systems for which unavailability or malfunction might represent an unacceptable potential damage, like embedded systems, e-commerce systems and e-banking systems, just to mention a few.

The risk and the nature of acceptable failures are a function of the requirements and the application domain of the system under development. Among others, factors that affect this identification are: risk of loss of life, serious damage to equipment, or to the business. Other factors are loss of work and time to restore the state of the system to a correct state with minimal loss of performed work.

However, in addition to restoring the system to a valid state as quickly as possible, it is necessary to properly identify and remove or encapsulate the causes of the failure, i.e. it is necessary to perform corrective maintenance. This will reduce the risk of the same cause provoking a failure in the future. However, a failure might have a number of causes, such as coding or design mistakes, software misuse or transient hardware malfunction. Failures can also be caused by accidental situations, without a specific location; for example, magnetic fields or radiation may induce hardware malfunction. This shows the importance of creating mechanisms to quickly detect and assure that the damage will remain below acceptable limits as is usual in defect tolerant systems [Avizienis et al, 2004] [Pullum, 2001]. Different from these, however, is the need to quickly identify and remove the defect, and redeploy the corrected system.

Recovery oriented software must focus on the following issues [Fox, 2002; Brown et al, 2002]:

1. Minimize the risk of the software containing defects, i.e. endogenous error sources;
2. Minimize the impact of exogenous error events;
3. Reduce the mean time to repair and redeploy (MTRP);
4. Reduce the mean time to recover (MTTR);
5. Minimize the consequences of failures.

By MTRP we mean the time elapsed since identifying the failure until its complete removal from the software, with a new deployed version. By MTTR we mean the time elapsed since the moment when the failure occurred until the moment the service is restored (completely or partially) in a dependable way.

Another important consideration is the mean time to fail of an application (MTTF). However, [Fox and Patterson, 2002] show that, from a user's point of view, it is better to reduce the MTTR than to enhance the MTTF.

1.1. Main Goals

This work focuses on preventing defects in a system, controlling external defects, reducing the MTTR and enabling fault tolerance in a system. The existing proposed solutions are quite complex and have a high cost of implementation. We look for a simpler alternative that would be easier and cheaper to implement.

We have two major goals in mind:

- Goal 1: should the software fail, it should be possible to put it back into dependable operation in a very short amount of time, even if the functionality has to be slightly reduced (graceful degradation).
- Goal 2: should the software fail, it should take a short time to diagnose the cause and it should also be possible to correctly remove or encapsulate the defect and redeploy the software in an acceptable short amount of time.
- Goal 3: use existing software development techniques. We wish to show enough evidence that there are sufficient available technologies to build software that meet goals one and two.

We propose the systematic and combined use of well-known software development techniques and tools, like software components, formal methods, mock components and existing design patterns. By doing this, we are complementing the problem of developing defect tolerant systems with the requirement of generating recovery oriented software. However, it is important to bear in mind that we do not intend to propose a new, or the modification of an existing, software development process.

The key idea is to show that development of recovery oriented software does not impose a prohibitive effort overhead, with the benefit of generating software that can co-exist with a set of failures, i.e., software that is more reliable. By co-existing, we mean reducing the damage and consequences of a failure below acceptable levels, in such a way that a user can continue to use the software with little effort after a failure has been identified and the system recovered.

The first step towards recovering from a set of failures is to create means to properly:

- Identify a failure that displays specific characteristics;
- Isolate the failure within a limited set of components of the system so that, if possible, they can be recovered;
- Identify potential damages due to the failure;
- Generate useful debugging information (that will be used to locate and fix the defect(s) that originated the failure).

Only then it is possible to run recovery code. Recovery code must:

- Tell the user the nature of the failure (this may not happen synchronously, for example, a log may be generated to be examined later);
- Take the system to a valid state of execution (termination of the software can be considered a valid state of execution and may be considered a recovery code, depending on the system).

Recovery code may:

- Terminate the set of components found defective (in order to continue system execution in a degraded way);
- Restore the set of components found defective (in order to continue system execution smoothly);
- Restore as much as possible damages caused by the failure;
- Isolate the damages caused by the failure, so that the system does not use such pieces of information.

Our work intends to study and develop techniques to achieve all these mentioned requirements.

1.2.
Evaluation Methods

We shall apply the ideas and concepts presented in this work to the development process of five different real world systems from various domains, and by measuring the overall performance of the development teams, the total effort spent, and the results for the generated software, we will be able to infer the effectiveness of the proposed techniques. All of the systems must have at least the following characteristics:

- Cannot be a simple information system; we are interested in active systems that control, monitor or interact directly with other systems or hardware. This does not mean that an observed system cannot have some part purely dedicated to storing and organizing information, but this part cannot be the main purpose of it;

- Teams must have different programmers; However, this restriction shall now be applied to software engineers to whom we expect to spread the culture of using the techniques;

Every system will be developed using a subset of tools, processes and techniques explained in previous sections of this document. The languages used for the system must be C, C++ or Java. The development of each system will be monitored, and the following metrics must be measured:

- Time spent for modeling the whole system; this includes the architectural and project phases, but does not include the requirements definition phase;
- Time spent for coding the whole system;
- Number of lines of code; number of lines of code dedicated to failure detection (assertions); number of lines of code dedicated to failure recovery: for these metrics to be reliable, every project will use the same code conventions;
- Number of failures detected by assertions in simulated production environment during the test phase; time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures not detected by assertions in simulated production environment during the test phase; time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures detected by assertions in the acceptance test phase (controlled production environment); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures not detected by assertions in the acceptance test phase (controlled production environment); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;

- Number of failures reported while in production considered light (i.e., no loss of work, recovery limited just to running the system again); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures reported while in production considered serious (i.e., loss of work, recovery not limited just to running the system again); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Time for "system stabilization" (i.e., time for the number of failures reported coming to acceptable levels);
- Total development time;

These metrics shall be measured with the help of tools, like time trackers and issue trackers. The obtained results will be compared to those present in the literature, as there are no available resources to adopt different development strategies for the same system.

We intend to collect enough evidence to show that our approach successfully generates software that can recover from a previously defined set of failures, with the drawback of a controlled development overhead, what makes a feasible adoption for the industry, which means keeping development costs, diagnosis costs, correction costs, restarting after failure costs, restarting after correction costs, damages etc. all under control. One interesting consequence of such an approach is kind of a "risk based development technique", in which the risks of a given set of failures to be observed can be estimated and controlled.

1.3. Document Structure

The next chapters are the following:

2) *Concepts and Technologies*: details the concepts and technologies used in the work.
3) *Combining Technologies to Develop Reliable Software*: presents the key ideas of the thesis proposal.

4) *Recovery Techniques*: discusses some ideas and techniques used to recover from failures detected during runtime.

5) *Experiments and results*: presents results from the application of the ideas to real world software.

6) *Discussion*: comparison with some related work found in the state-of-the-art literature and the work presented in this document.

7) *Conclusions, contributions and future work*: discusses the conclusions and some contributions for this work, and proposes some possible future work derived from the thesis results.

8) *References*: presents the references used in this work.