Safety-Critical Software Failure Prevention Using Defence-in-Depth Approach

Abdalla M. Khattab[1], Hany Sallam[2], Ehab Shafie[3]

[1]Department of Computer Science, Faculty of Computing and Information, Al-Baha University, Kingdom of Saudia Arabia.
[2,3]Nuclear and Radiological Regulatory Authority, EGYPT

Abstract
In the nuclear field, software-based systems are of increasing importance to safety for both research reactors and Nuclear Power Plants (NPPs) as their use is increasing in both newly installed and refurbished old facilities. These software-based systems are used in safety systems, such as the reactor protection systems, and safety-related applications, such as some functions of the process control systems and the monitoring systems. Taking into account the criticality and severity of such systems, these systems are known as safety-critical software systems.
The reliability of safety-critical software is crucial for ensuring nuclear safety. Reliability is one of the most important requirements of software-based systems. For safety-critical software systems, it is not enough to depend on testing to ensure that the system will not fail and if it failed, it will fail-safe. Although defence-in-depth (DiD) strategy is used in designing and developing many systems in the nuclear field to ensure the fail-safe of these systems, this strategy is still not used in developing safety-critical software systems used in NPPs and research reactors. This paper proposes a new software fault-tolerant methodology based on using the DiD strategy. The proposed methodology is a novel technique to ensure software safety.

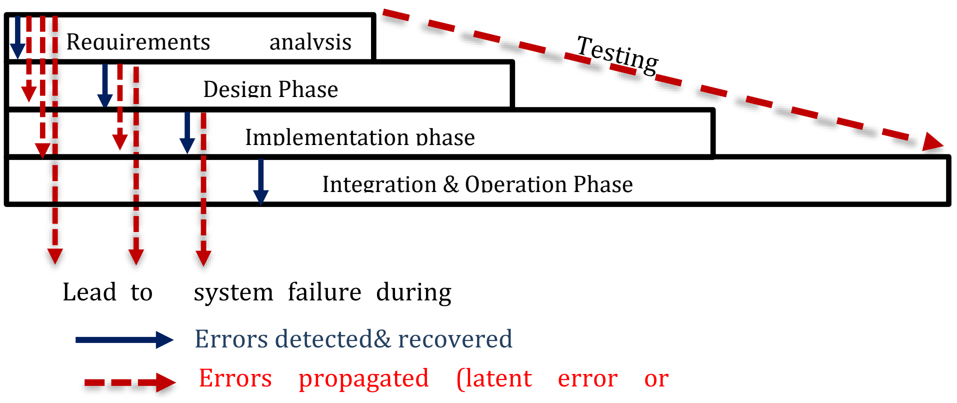
Keywords Safety-critical software systems; defense-in-depth; fail-safe; fault tolerant .

1. Introduction

The development life cycle of software is essentially a series of the following phases, planning phase, requirements analysis phase, design phase, implementation phase, integration phase, deploying and maintenance phase [1, 2]. During different phases of the safety-critical software development lifecycle, some errors may remain hidden although of testing process for each software development phase [3]. These errors will remain hidden during normal operation and will announce about themselves at a given time. Software errors, which represent the seed for fault and then the failure of the system [4]. Human is the source of software errors at different phases [5]. These errors are waiting specific conditions to appear as a failure of the system.
Errors in a given phase can be initiated from errors in that phase processes or referenced to errors in processes of previous phases as an example: error in implementation phase may result from errors in implementation processes such as the using of an adequate control structure, or errors in processes of previous phases such as design phase or requirements analysis phase as shown in Figure 1.

Errors in a given phase may be discovered by testing and resolved during this phase otherwise, these errors may be remained hidden and propagated to the next phases and give arise to faults during run time. These faults will cause system failure and consequently give arise to incidents as shown in Figure 2. These errors may be classified into errors of commission or errors of omission. Errors of commission involve implementing code that is not part of the specification or design. Errors of omission involve lapses wherein a behavior specified in the design was not implemented.

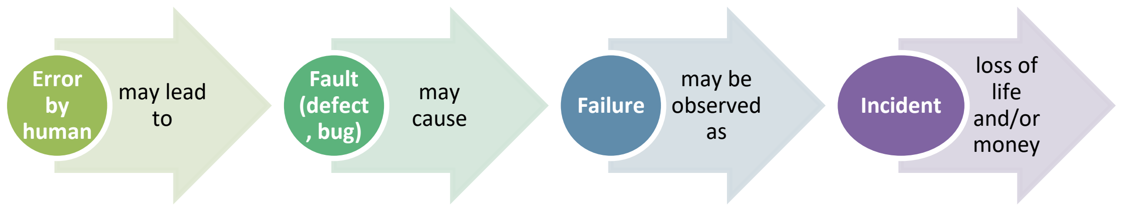
Fig.1 Errors propagation


Requirements analysis
Design Phase
Implementation phase
Integration & Operation Phase
Testing
Lead to system failure during
Errors detected& recovered
Errors propagated (latent error or

A software error is a discrepancy in code that causes a fault, it is incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner. Fault is caused by an error in the source code that was compiled into an executable program. Error is a human mistake that is manifested by fault [2, 4]. A failure is an incorrect external behavior, in other words, *failure* is the condition or state of not being able to meet an intended objective according to its requirements [2, 4]. A failure is diagnosed when a system or a component failed to perform the required function according to its designed requirements. Examples of such faults are out of memory error which occurs when a program consumes more memory than the hosting hardware and segmentation fault in which a programmer points out to a memory location which is out of available space or reserved address.

The majority of software faults that are diagnosed as code faults are referenced to design faults. Some software faults are basically faults in the requirements analysis. The design implements the requirements and the code implements the design. The software design may not implement the software requirements specification [6]. Whatever requirements analysis or design faults are initially referenced to human errors in these phases.

Fig. 2 Relation between Fault, Error, and Failure



2. Defence-in Depth (DiD)

Software fault tolerant is the designed ability of software to detect and recover from a fault that is happening or has already happened in the system in which the software is running in order to provide service in accordance with the specification [7].

The goal of using DiD is to include safety features in the software design to ensure that the software will perform correctly in case of error, and prevent system failures. The need for fault tolerant performance should be early engineered during the system requirements analysis phase followed by design satisfying these requirements correctly. The application of DiD strategy is well known in physical systems whatever for ensuring safety or security, where physical and/or logical barriers can be used. But using DiD strategy for developing fault tolerant software is not known up to the date of writing this paper. The idea is novel and represents the main contribution of this paper.

Software DiD strategy must be engineered in phases that proceed implementation phase. Applying DiD strategy should begin with planning phase followed by requirements analysis and refinement phase, extend to architectural definition, and move through design, implementation, and test.

3. Difference between DiD for Physical and Software systems for Software

Software systems unlike physical systems, the success or failure of software systems primarily depends on the input, which may activate hidden fault in the software. On the other side physical systems success or failure depends on many parameters other than the input. If a given software system is tested against a given input data and detected faults (bugs) are removed, the system will remain functioning correctly with similar input data. The question that should be asked in this case, could the software be tested and validated against all possible input data, of course this process will be time and cost consuming. It can be concluded that the number of hidden faults in software is inversely proportional to the input space coverage which is used for testing. Explaining the difference between DiD in physical systems and DiD in software systems, software DiD is an imitation to physical DiD. DiD in Software levels or layers are inspired on providing similar defense layers in software to prevent error propagation. While DiD in physical is developed to prevent adversaries from attacking assets. For security of software [8] it is the same as safety [9] and security of physical systems. In this case there is a real attacker tries to threaten the security (confidentiality, integrity, and availability) of the of software system.

In case of safety critical software systems, there is no external attacker, there is a hidden error or in other words undiscovered error during testing. Which might be remained inactive for a long time. It becomes active under given circumstances in terms of input data. With a given input data, this error becomes active and threaten the safety of the system by providing error output or incorrect action although of testing of the software with different input data ranges represent different operation states.

4. Software DiD

To prevent loss of life, damage of the systems and components, both faults and failures must be contained, in other words the failure effect propagation paths, must be stopped or contained to prevent system failure. This failure effect containment is achieved by DiD layers which represent failure containment zone boundaries. If failure succeeded in crossing out one of these boundaries, the other boundaries will stop and limit its propagation to other components of the system.

So, the target of software DiD firstly is to detect errors in software development processes, secondly to prevent these errors to propagate to other decedent processes, thirdly to mitigate the consequences of this

error, i.e. the software fails safely. This will require hierarchically modularized the software and define the function of each module also the input and output, limiting the capabilities of each module to its designed function. With clear modularized, software errors can be traced and isolated. Advantages of using modularization in software development include simplified testing, easier maintenance, and lower propagation of errors.
This can be achieved by following general strategies in designing the safety-critical software systems such as redundancy to avoid single failure criteria, diversity to prevent common cause failure, as a result of an internal design flaw or programming bug which could cause all identical redundant modules to fail in the same way at the same time [10], and voting, if any one of the modules fails, it is outvoted by the others, so that the output of the vote is always the correct set of information [11]. With diverse redundant modules
if one version fails on a particular input, the alternate versions should be able to provide an appropriate output. Software is subjected to implementation errors caused by shortcomings in planning phase, requirements analysis phase, coding and testing phase, resulting in in-service problems that can cause damage to the system which based on that software.

5. Fault Tolerant by Design

Fault tolerant is achieved through the use of redundancy in the hardware, software, information, or time domain [6]. For fault tolerant design, the designer needs to anticipate the fault or deviation from expected behavior, and can develop necessary measures to cope with the situation [12]. The selection of the fault tolerant techniques used in a system depends on the requirements of the application such as reliability and safety. Reliability is determined by the probability of failure per demand, and safety is determined by the consequences of these failures. So, safety is achieved through the use of reliable structures, components, systems and procedures [13]. For critical applications such as nuclear, it is required to have a high degree of confidence on the correct and safe operation of the computer systems to assure their ability to prevent loss of life or damage to components. Another requirement for critical application is the availability of the system which means the system will be ready to provide the intended service when so requested with a very high probability. The design of systems should be fault tolerant by design capabilities to satisfy particular application requirement. Fault tolerant by design is a complex process undergo to theoretical and experimental analysis in order to find the most appropriate tradeoffs within the design space.

5.1 Redundancy

Software fault tolerant is based on the use of redundancy to detect and recover from faults [14]. Redundancy is a fundamental aspect of fault management designs, as all fault management mechanisms rely on some form of redundancy [11]. In each case, the fault management mechanism operates only against certain classes of faults and failures. The classic example is that in hardware identical redundancy, the fault management mechanism mitigates against random part failure in any of the redundant strings, but cannot mitigate against a design flaw common among all of the redundant strings. Redundancy is fundamental to fault management design, verification and validation, and operations. The principle of redundancy always applies to fault management, and the principle can be used to understand, assess, and justify the design and the risks of not having fault management when those risks are acceptable. Functional redundancy is the use of dissimilar hardware, software, or operations to perform identical functions. Dissimilar redundancy is typically utilized for failure detection, but it can also be potentially utilized for failure response [11].

5.2 Fault localization

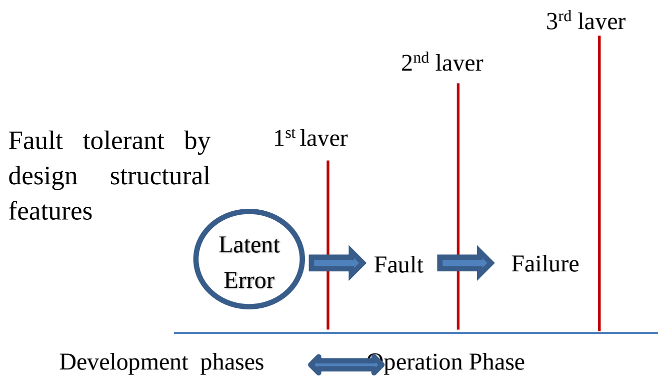
Failure masking, fault tolerant, and fault and failure containment are closely linked concepts [11], [15]. To prevent loss of life, loss of the system, or loss of mission, both faults and failures must be contained. The concept of failure containment is easy to understand: failure effects, as they spread along failure effect propagation paths, must be stopped or contained to prevent safety system failure. Most of the failure containment activities are associated with safety-critical systems, where the main concerns are for the system to be as safe or to be as accident free as possible [16]. The location at which a particular set of failure effects are stopped is called a failure containment zone boundary. The set of failure containment zone boundaries creates a failure containment region, in which certain classes of failure effects are contained. If the failure recovery function operates properly and successfully, then failure effects are generally contained, and, for this reason, failure containment is not considered a separate, independent fault management function. It is encompassed in the overall process and functions of failure detection, fault isolation, and failure recovery. However, fault containment is a separate issue that must be addressed separately from the containment of failure effects. The prevention of the spread of permanent physical (or logical damage, with software) must be addressed with means different from those of containing failure effects.

6. DiD-Based Fault Tolerant (3 Layers)

In this paper DiD strategy is proposed to get software fault tolerant. Software fault tolerant means that it has the capabilities to detect fault, prevent failure, and failure recover or handling. Fault tolerant techniques limit defect manifestation to a local area to avoid global failures, through the use of some redundancy designed into the software systems or their operations. To achieve these capabilities the software should be designed and structured according to basic techniques that help in building DiD layers. These techniques such as redundancy and modulization should be planned in early phases of developing the software.

The discussion in this paper concerned with software fault management during operation of software not during the development phases, where our focus is to handle errors that are not detected during the development life cycle, which are known as latent errors, and appears as faults during run time. The strategy here is build multi defensive layers around the error as shown in Figure 3, each defensive layer has defined function. The ultimate objective of software DiD design is to prevent error from developing into failure and jeopardize the system safety.

Fig.3. Software Fault Tolerant



6.1 Layer 1

Fault Detection: basic element of a fault tolerant design is error detection. Error detection is a critical prerequisite for all fault tolerant mechanisms. The first barrier will be error detection which is based on structuring the software design using modulization, redundancy, atomic structure, system closure, and portioning techniques. This barrier takes appropriate actions to prevent the propagation of detected errors to other components of software.

Graded approach should be used to determine the degree (and coverage) to which error detection mechanisms should be used in a design. Graded approach rationalizes the cost of the additional redundancy and the run-time overhead with the importance of a software module to safety. Fault tolerant redundancy is not software functional requirement but rather it is a non-functional requirement that contributes to the quality of software [17]. Actual usage of fault tolerant in a design is based on trade-offs of functionality, performance, complexity, and safety [18].

6.2 Layer 2

Diagnosis: After a fault is detected, the system must assess its health in order to decide how to proceed. If the containment boundary layers are highly secure, diagnosis is reduced to just identifying the enclosed components. If the established boundaries are not completely secure, then more involved diagnosis is required to identify which other areas are affected by propagated errors.

Containment: in order to be able to deal with the large number of possible effects of faults in a complex safety-critical software systems, it is necessary to define confinement to boundaries for the propagation of faults. DiD containment boundary layers are usually arranged hierarchically throughout the modular structure of the system. Each boundary layer protects the rest of the system from errors occurred within it and enable the designer to count on a certain number of correctly operating components by means of which the system can continue to perform its function.

6.3 Layer 3

Masking: The timely flow of information is a critical design issue for some applications such as reactor protection system in nuclear power plants. For reactor protection system it is not possible to just stop or restart the information processing system to deal with detected errors. Masking is the online correction of errors; fault masking prevents a fault from spreading beyond a certain location [11]. The fault **is** masked if the design specification **is** satisfied by the incorrect state [19]. In general, masking errors is difficult to perform in line with a complex component. Masking, however, is much simpler when redundant copies of the data in question are available.

Repair/reconfiguration: In general, systems do not actually try to repair component-level faults in order to continue operating. Because faults are either physical or design-related, repair techniques are based on finding ways to work around faults by either effectively removing from operation the affected components or by rearranging the activity within the system in order to prevent the activation of the faults[20].

Recovery and Continued Service: After a fault is detected, a system must be returned to proper service by ensuring an error-free state. This usually involves the restoration to a previous or predefined state, or **rebuilding the state by means of known-good external information.**

7. Conclusion

In this paper, a new software fault tolerant methodology based on using DiD strategy is proposed. The presented methodology is a novel technique to ensure software safety.

DiD strategy is used to develop fault tolerant software and avoid the impact of latent errors in safety-critical software systems. Latent errors represent the root cause of failure of critical software systems which have potential to human life and physical systems. Taking into account, not all software latent errors are discovered and corrected, fault tolerant development is crucial requirement for safety-critical software systems. This paper introduces a new application for DiD strategy in developing safety-critical software systems and by setting the base for processes to be engineered during software development. The proposed strategy is based on multi defensive layers around the error, each defensive layer has defined function. The first to fault detection, in the second layer, the fault is diagnosed and confined. The third layer performs fault masking, repairing, and recovery. Based on the presented defensive layers, the chance of developing the latent errors into system failure will be dimensioned, and accordingly the chance of critical software failure will be minimized.

References

1. IEEE Standard for Developing Software Life Cycle Processes," in IEEE Std 1074-1991 , vol., no., pp.1-112, 29 Jan. 1992, doi: 10.1109/IEEESTD.1992.101080.
2. ISO/IEC/IEEE International Standard - Systems and software engineering -- System life cycle processes," in ISO/IEC/IEEE 15288 First edition 2015-05-15 , vol., no., pp.1-118, 15 May 2015, doi: 10.1109/IEEESTD.2015.7106435.
3. Lincoln, John E. "Lifecycle considerations for device software." Journal of Validation Technology, vol. 18, no. 1, Winter 2012, p. 15+. Accessed 10 June 2020.
4. Zubrow, D. "Ieee standard classification for software anomalies." IEEE Computer Society (2009).
5. Lopez, Tamara (2016). Error Detection and Recovery in Software Development. PhD thesis The Open University.
6. ISO 12749-5. “Nuclear energy, nuclear technologies, and radiological protection – Vocabulary - Part 5: Nuclear reactors”. 2018.
7. ISO 1709. “Nuclear energy - Fissile materials - Principles of criticality safety in storing, handling and processing”. 2018.
8. Stytz, Martin R. "Considering defense in depth for software applications." IEEE Security & Privacy 2.1 (2004): 72-75.
9. Zhou, Jixiang, Pan Zhu, and Peng Xiao. "Defense-in-depth and diversity design of instrumentation and control system in nuclear power plant." Nuclear Power Engineering 35.1 (2014): 122-124.
10. Jones, Harry. "Common cause failures and ultra-reliability." 42nd International Conference on Environmental Systems. 2012.
11. Johnson, Stephen B., et al., eds. System health management: with aerospace applications. John Wiley & Sons, 2011.
12. Basu, Swapan. Plant hazard analysis and safety instrumentation systems. Academic Press, 2016.
13. Nowakowski, Tomasz, et al., eds. Safety and reliability: Methodology and applications. CRC Press, 2014.
14. Carzaniga, Antonio, Alessandra Gorla, and Mauro Pezzè. "Handling software faults with redundancy." Architecting Dependable Systems VI. Springer, Berlin, Heidelberg, 2009. 148-171.
15. Heimerdinger, W. L., and C. B. Weinstock. "A Conceptual Framework for System Fault Tolerant (CMU/SEI-92-TR-33, ADA264375). Pittsburgh, Pa.: Software Engineering Institute." (1992).

16. Tian, Jeff. Software quality engineering: testing, quality assurance, and quantifiable improvement. John Wiley & Sons, 2005.
17. Critchley, Terry. High-performance it services. CRC Press, 2016.
18. Torres-Pomales, Wilfredo. "Software fault tolerant: A tutorial." (2010).
19. Laski, Janusz, Wojciech Szermer, and Piotr Luczycki. "Error masking in computer programs." Software Testing, Verification and Reliability 5.2 (1995): 81-105.
20. Surbhi Bansal , Jon Tømmerås Selvik. vestigating the implementation of the safety-diagnosability principle to support defence-in-depth in the nuclear industry: A Fukushima Daiichi accident case study, Engineering Failure Analysis Volume 123, May 2021, 105315.