2009 IEEE International Advance Computing Conference (IACC 2009)
Patiala, India, 6–7 March 2009

# Operation Level Safety Analysis for Object Oriented Software Design Using SFMEA

**Pankaj Vyas**
*Lecturer, Computer Science and Information System*
*Birla Institute of Technology and Science, Pilani.*
pankajv@bits-pilani.ac.in

**R. K. Mittal**
*Professor, Mechanical Engineering and Computer Science*
*Birla Institute of Technology and Science, Pilani.*
rkm@bits-pilani.ac.in

## Abstract

Recent trends have indicated the increased use of object oriented technology not only for the design and development of traditional software but also for safety critical software. There has been an ongoing effort for the application of traditional well documented and well tested hardware safety and reliability analysis techniques to software. Software Failure Modes and Effects Analysis (SFMEA) is one such technique that has been adopted from its hardware counter part Failure Modes and Effects Analysis (FMEA). Despite differences in operational failure modes between hardware and software, the recent research has shown the usefulness of the technique in software development process. This paper aims to: (i) highlight the application of SFMEA in object oriented design process and (ii) use the results of analysis obtained from previous step at implementation phase for improving robustness of the code.

## 1. Introduction

Object oriented technology is being explored more and more for its applications in safety critical systems and Unified Modeling Language (UML) has become the universal standard for modeling the static as well as dynamic aspects of both traditional and safety critical software. In recent years, there has been an ongoing effort for improving and enhancing the quality of the safety critical software by applying traditional hardware safety analysis techniques like Failure Mode and Effects Analysis (FMEA) to software known as Software Failure Modes and Effects Analysis (SFMEA) [1]. SFMEA is bottom–up approach which starts from basic failure events and then investigates its effect on system. When the criticality of the effects is also considered, the technique is called a Software Failure Modes, Effects and Criticality Analysis (SFMECA). SFMEA and SFMECA are primarily used to discover software design defects during software development. SFMEA is an extension of FMEA developed in the 1970s to provide a systematic form of failure analysis that could improve reliability. The application of SFMEA or FMEA focuses on two important aspects: (i) How any hardware/software component can fail and (ii) What will be the consequences(s) or effects on the system as whole.

For hardware systems FMEA is well documented, automated and has been supported by rich body of research. But for software systems the technique is not yet standardized and no universally accepted standard exists for application of the process. This paper presents a novel approach for the application of SFMEA for object oriented based software design. The analysis process is applied at method(s) or operation(s) of classes identified during design phase of object oriented life cycle. The rest of the paper is organized as follows. Section 2 gives an overview of related and background of published works in the fields of SFMEA as well as its application in various phases of software development life cycle. In the Section 3, the reason for applying the software FMEA process at operation-level or method-level is discussed. In Section 4, the proposed approach for applying software FMEA at method level is described and rules for identifying various failure modes are discussed. The application of the proposed approach is explained in Section 5 by taking methods of a sample MessageQueue class as examples. Section 6 will conclude the paper by highlighting the importance of software FMEA process at software design phase.

## 2. Background and Related Work

The application of FMEA to software has so far remained a manual, time-consuming and labor intensive activity. Despite this weakness the research in the past has indicated the great importance of the process in every phase of software lifecycle. The work described in [1] has provided examples of various software failures. Lutz and Woodhouse [2,4] recommended use of SFMEA in conjunction with another well known backward safety analysis technique known Software as Fault Tree Analysis (SFTA) for analyzing the requirements of critical space craft software. Lutz [5] applied SFMECA and SFTA for the safety analysis of two space instruments Mars Microprobe Project and Earth Orbiting System's Microwave Limb Sounder. Chunping et el. [3] described an automated analysis method and software package tool for SFMEA and explains its use in digital fly control systems. Goddard [6] provided a summary overview of two types of SFMEA: system software FMEA and detailed software FMEA. Goddard also outlined the importance of applying system software FMEA in early phases of software life cycle most suitably at requirement analysis. The detailed examples of functional analysis, interface analysis and detailed analysis of software using SFMEA can be found in [7]. Detailed survey of various SFMEA techniques and standards can be traced to the works described by Haapanen and Helmunen [8]. Guichet and Baron [9] have applied FMEA to perform message level risk analysis for UML based systems and described eleven possible error or failure modes for messages. Ozarin [10] analyzed the assembly language code using SFMEA and outlined the procedure for performing

978-1-4244-2928-8/09/$25.00 © 

Software FMEA. For the smooth and easy use of the software FMEA process Ozarin also advocated to use of database tools. Use of FMEA for UML based applications can be found in [9, 11]. The earlier use of software FMEA is either manual [7] or computer aided [3, 11]. Recent research indicated the automated attempts for the code level analysis [13, 15 and 16]. Neal and Price [15] used fault propagation model for the automated analysis. The effort is going on for automating the Software FMEA process by using the dependence relations among software components.

For last two decades the focus of research for Software FMEA process has been for the code level analysis. Very little published work exists for the application of SFMEA for software design phase. Many researchers stressed upon the application of Software FMEA to earlier phases of software development life cycle [2, 6]. Moreover SFMEA process at design phase can help the developers to improve the robustness of the code written at implementation phase.

## 3. Operation level Safety Analysis

Why operation level or method level SFMEA? In object oriented systems classes are group of operations or methods. Hect [11] noted that when a method does not execute in accordance with its specification then software can fail under some condition. Methods have been considered equivalent to a part of hardware system where failure can occur under certain conditions. Ozarin [12] discussed the relative merits of performing analysis at four different levels: Method-level analysis, Class-level analysis, Module-level analysis and Package-level analysis and also stressed that for SFMEA process to be more accurate and fruitful then it should be performed at the lowest level of analysis i.e. Method-level. During object oriented design the methods are the more suitable candidates for the application of the process. The other reason for selecting methods is guided by 'how we test an application and fix the bugs'. While testing an application if error is encountered a search is carried out to locate the piece of code where error occurred or more specifically the instructions of the method. As SFMEA is to be applied in design phase, choice of methods seems to be the suitable option.

Classes have methods that provide implementations for their operational responsibilities. Whenever, objects communicate either by sending signals or by passing messages, it is internally linked to a method call. Methods that perform safety critical tasks are generally viewed as agents fulfilling a contract for performing its operational tasks safely. Methods may/or may not have formal parameters and/or return values. UML's Object Constrained Language (OCL) provides semantics for defining pre conditions, post conditions for methods, class invariants for classes and range and type constraints for formal parameters. For successful application of SFMEA for methods we have to identify failure modes for various types of methods and while identifying the failure modes we have to focus on how method is going or is to be implemented. So for this purpose we have used Pseudo code Description Language (PDL) which is independent of any high level language.

## 4. Proposed Approach

Our approach presented in this section is based upon the previous works described in [10, 11, 12, 14, and 15]. Our approach extends the idea of method-level analysis to object oriented software design phase. Our approach will also describe the details for the identification of various failure modes for the methods.

### 4.1 Assumptions for the Proposed Approach

The proposed approach is based on the following assumptions:

1. The operation or method under consideration has a well defined protocol for its correct usage consisting of the attributes: Precondition Invariants, Post condition Invariants, and Proper Signature containing an ordered formal list of parameters.
2. Operation's behavior is implemented in a pseudo code type description language as mentioned in the example of Section 5.
3. Pseudo code description of the method is virtually divided in two parts: constraint checking part and actual logic part. All method constraints are checked upon entry in the method and after that actual logic will execute.
4. Our approach is not considering variable failure modes as mentioned in [7, 10] rather it will only focus upon the functional aspects the method. Variable Failure modes can be considered when method code is implemented in any high level language i.e. code level analysis or post implementation analysis.
5. It is assumed that whenever invoked then method performs its task T in finite time, there is no time constraint.
6. Operation or method under consideration is neither handling any concurrent operation nor is being shared among various clients.

### 4.2 Identifying Method or Operation Failure Modes:

Based upon the various types of operations or methods that a class can have and constraints and parameters that an operation can have we have identified the four types of failure modes for a method: Precondition Violation Failure Modes, Parametric Failure Modes, Method Call or Invoke Failure Modes, Post Condition Violation Failure Modes. These are described below:

**A. Precondition Violation Failure Modes**: The guidelines mentioned in the following steps can be used to define pre condition failure modes for the methods

I(a) Check what will be the method response if invoked with precondition violated. If there is no built in check in the method pseudo code then make precondition violation as failure mode else if method raise the alarm in the form of exception say E then make precondition violation and !raise(E) (i.e. precondition is false but exception mechanism fails) as the failure mode. [ raise(E) is used for raise exception E and !raise(E) is used for not raising the exception E]

I(b) The other failure mode can be in the form of a situation when precondition is satisfied but its corresponding exception is raised.

**B. Parametric Failure Modes**: To identify parametric failure modes the following guidelines can be used:

II Check constraints on parameter values. Do not focus on their types at the moment as type checking is implementation level issue. Exclude any parameter constraint if it is covered by precondition of the method For any other parameter constraint check the method response. If no protection is provided in the method pseudo code then consider constraint itself as the failure mode else if the method raises any alarm in the form of exception then include two failure modes for two cases (In the first case constraint is false and its corresponding exception is not raised and the second is constraint is true but corresponding exception is raised)

**C. Method Call or Invoke Failure Modes**: To identify this type of failure modes consider the following two sub cases:

(a) A method m1 invokes method m2 of the same class or super class then include following failure modes in the list of failure modes of m1

III(a) m1 invokes m2 in wrong order (if the invocation of m2 is condition based)

III(b) if m2 is parameterized then make m1 invokes m2 by wrong parameters as the failure mode

(b) A method m1 of class A invokes method m2 of B then include following failure modes in the list of failure modes of m1

IV(a) m1 fails to invoke m2 (because of lack of instance of object of class B)

IV(b) m1 invokes m2 in wrong order (if the invocation of m2 is condition based)

IV(c) if m2 is parameterized then make m1 invokes m2 by wrong parameters as the failure mode

**D. Post Conditional Failure Modes**: These types of failure modes are additional failure modes. Use the following rule to identify the post conditional failure modes

V Check the various forms for post condition failure. Write post condition failures in to effects column of the respective Failure Modes Table ( as shown in Table 1, 2 or 3) and then do the back ward analysis in the pseudo code to find the error for it. If the error originated from the method in consideration then includes it in the failure mode list otherwise ignore it for the current method.

## 5. Example: MessageQueue class

The example considered in this section is converted from implementation to pseudo code form to describe the application of our approach. The example pseudo code is for the class named MessageQueue which represents the queue of messages and constraints are shown only for three methods: constructor of the class, removeFirst() and add() method. Table 1, 2 and 3 show the failure modes identified for constructor method, removeFirst() method and add() methods. The structure of the table is selected from [14]. The failure mode column defines the failure mode for the method, brief description of the failure modes is given in description column, causes column defines the reasons for the failure modes, effects column defines the effects on the system, Actions/response column indicates the response of the system if the corresponding failure occurs and the last column defines how the corresponding failure mode has been identified. Table 1 includes only one failure mode for the constructor method i.e. when cap parameter is 0 and the InvalidCapacityException is not raised. The opposite failure mode for constructor method i.e. when cap is not equal to 0 and InvalidCapacityException is raised is not considered because cap is local parameter and active inside only constructor method. Table 2 shows two failure modes for the removeFirst() method i.e. size() = 0 && !raise(*E2) and size() != 0 && raise(E2) where E2 is QueueEmptyException. First thing to be noted for removeFirst() method is that it is invoking size() method to determine the output of the condition. It may be possible that size() may return the wrong result then it may effect the outcome of removeFirst() method. The same approach is applied for the add() method also. The third additional failure mode which is msg = null is parametric failure mode. It is included here because this particular case is not the part of the precondition of the add() method. One important question still needs to be answered as why there is no failure mode indentified according to rules III(a), III(b), IV(a-c) and V. The answer for this is that rules from IV(a) to IV(c) are not applicable for add() and removeFirst() methods as they only call the methods isFull() and size() of the same class MessageQueue. Now suppose method removeFirst() fails to call size() method then there will be two cases: First, the queue may be empty which is same as first failure mode of the removeFirst() method. The second case may be that queue is not empty then it does not cause any failure in the system. So both rules III(a) and III(b) are not applicable for removeFirst() method. The same explanation holds for the add() method. The other entries in the table are entered manually and this depends upon the knowledge and understanding of analyst doing the analysis.

```
class MessageQueue
{
pivate Message[] elements;
private int head;
private int tail;
private int count;
private int capacity;
/** Description for Constructor MessageQueue
@Task constructs an empty message queue
@param cap: the maximum capacity of the queue
@preccondition cap > 0
@postcondition capacity! = 0
*/
public MessageQueue (int cap)
{
** Pseudocode Description
if (cap is less than or equal to 0)
{
raise the InvalidCapacityException;
exit;
}
set size of elements equal to capacity;
set head equal to zero;
set tail equal to zero;
set count equal to zero;
set capacity equal to cap;
}
/** Description for Method removeFirst()
```

```
@Task removes the first element from the queue and returns
it
@param: none
@preccondition size() > 0
@postcondition getsize_new() = getsize()_old - 1
*/
public Message removeFirst()
{
** Pseudocode Description
if(size() is less than or equal to 0)
{
raise QueueEmptyException;
exit;
}
Message msg;
set msg equal to elements[head];
set head equal to (head + 1) % capacity;
set count equal to count -1
return msg;
}
/** Description for Method add()
@Task adds msg at tail end of queue
@param msg: Message to be added in queue
@preccondition!isFull() i.e. queue should not be full
@postcondition getsize_new() = getsize()_old + 1
*/
public void add(Message msg)
{
** Pseudocode Description
if(isFull() equal to true)
{
raise QueueFullException;
exit
}
set queue[head] equal to msg;
set tail equal to (tail + 1) % capacity;
set count equal to count +1
return msg;
}
@Task returns true if queue is full otherwise false
public boolean isFull() { }
@Task returns the current count/size of the queue
public int size() { }
}
```

Table 1: Failure Modes for constructor MessageQueue()

| Failure Mode | Description | Causes | Effects | Action/Response | How Identified |
|---|---|---|---|---|---|
| cap = 0 && !raise(*E1) | Capacity is equal to 0 but invalid capacity exception is not raise | Failure of Exception Handling Mechanism | Capacity of the queue will be zero | Check for the capacity before exiting the method. | By applying rule I(a) |

**E1 → InvalidCapacityException, raise(E1) stands for raising exception E1 and !riase(E1) stands for not raising exception E1**

Table 2: Failure Modes for removeFirst() Method

| Failure Mode | Description | Causes | Effects | Action/Response | How Identified |
|---|---|---|---|---|---|
| size() = 0 && !raise(*E2) | Size of the queue is equal to 0 but QueueEmptyExceptio n is not raised | Failure of Exception Handling Mechanism or size of the queue is not properly updated | Count of the queue will become negative and any dummy value as message can be returned | Check for the count value each time before decrementing. | By applying rule I(a) |
| size() != 0 && raise(E2) | Size of the queue is not equal to 0 but QueueEmptyExceptio n is raised | Count not updated properly after last addition in the queue | Remove operation failure | Check the value of count after every successful add and make sure it is added properly | By applying rule I(b) |

**E2 → QueueEmptyException,**

Table 3: Failure Modes for add() Method

| Failure Mode | Description | Causes | Effects | Action/Response | How Identified |
|---|---|---|---|---|---|
| isFull() = true && !raise(*E3) | Queue is full and queue full exception is not raised | Failure of Exception Handling Mechanism | Msg will be added to head by overwriting the previous value | Check for count and capacity values if they are equal do not add | By applying rule I(a) |
| isFull() = false && raise(E3) | Queue is not full and queue full exception is raised | count not updated after last delete | Add failure msg will not be added in to the queue | Check the value of count after delete and make sure it is one less than the previous count | By applying rule I(b) |
| msg = null | Passed message is null | msg parameter may not be properly initialized by the caller | Null message added to queue | Check the value for message before adding OR include it in precondition of the message | By applying rule II |

**E3 → QueueFullException**

## 6. Conclusions

The application of our approach described in this paper has demonstrated the importance and usefulness of the technique for object oriented software design phase. In the implementation phase of the software life cycle, Action column of Tables 1, 2 and 3 can be used for improving the robustness of the code. There are two main weaknesses of the software FMEA process that have been observed. First, the process has so far been a manual, labor intensive and time consuming activity. Second, the process considers only a single fault at a time. Many researchers advocated the use of the process for software life cycle but no universally

accepted standard exists for applying the process at various phases of life cycle. So the future research directions should be directed to achieve the following:

- The application of the process should be made easy and less labor intensive by the development of an automated analysis tool.
- There is a need to define the procedures for the application of the process to various phases of software life cycle.

## 7. References


1. Reifer, D. J., "*Software Failure Modes and Effects Analysis*," IEEE Trans. Reliability, vol. 3, pp. 247-249, Aug, 1979.
2. Lutz, R. R., and Woodhouse, R.M., "*Experience Report: Contributions of SFMEA to Requirement Analysis,"* In Proc. 2nd International Conference on Requirements Engineering (ICRE'96), Colorado Springs, Colorado, USA, pp. 44-51, April 1996.
3. Chunping, H., Peiqiong, L. and Yiping, Y., "*The Application of Failure Mode and Effects Analysis For Software in Digital Fly Control System*," In Proc. 16th AIAA/IEEE Digital Avionics Systems Conference (DASC), Irvine, CA, USA, pp. 8-13, Oct. 1997.
4. Lutz, R.R. and Woodhouse, R.M., "*Requirement Analysis using Forward & Backward Search*," Annals of Software Engineering, vol. 3, pp. 459-475, 1997.
5. Lutz, R.R., "*Applying Adaptive Safety Analysis Techniques*," In Proc. 10th International Symposium on Software Reliability Engineering., Boca Raton, Florida, USA, pp. 42-56, Nov. 1999.
6. Goddard, P. L., "*Software FMEA Techniques*," In Proc. Annual Reliability and Maintainability Symposium (RAMS 2000), LA, USA, pp. 118-123, Jan. 2000.
7. Bowles, J. B. and Wan, C., "Software Failure Modes and Effects Analysis for a Small Embedded Control System," In. Proc. Annual Reliability and Maintainability Symposium (RAMS 2001), Philadelphia, Pennsylvania, USA, pp. 1-6, Jan. 22-25, 2001.
8. Haapanen, P., and Helmunen, A., "*Failure Mode and Effects Analysis of Software-Based Automation Systems*", *STUK-YTO-TR*, Aug. 2002.
9. Guiochet, J. and Baron, C., "*UML based FMECA in risk analysis*". In Proc. of the European Simulation and Modeling Conference ESMc2003, Naples, Italy, October 2003.
10. Ozarin, N. and Siracusa, M., "A Process for Failure Modes and Effects Analysis of Computer Software," In Proc. Annual Reliability and Maintainability Symposium (RAMS'03), Tampa, Florida, USA, pp. 365-370, Jan 2003
11. Hecht, H., and Hecht, M., "*Computer Aided Software FMEA for Unified Modeling Language Based Software*," In Proc. Annual Reliability and Maintainability Symposium (RAMS'04), LA, USA, pp. 243-248, Jan. 2004.
12. Ozarin, N., "*Failure Mode and Effects Analysis During Design of Computer Software*," In Proc. Annual Reliability and Maintainability Symposium (RAMS'04), LA, USA, pp. 201-206, Jan. 2004.
13. Snooke, N., "Model-based Failure Modes and Effects Analysis of Software," In Proc. 15th International Workshop on Principles of Diagnosis ,DX-2004, Carcassonne, France, pp. 221-226, June 23-25, 2004.
14. Lauritsen, T., and Stalhane, T., "Safety Methods in Software Process Improvement," In Proc. 12th European Conference on Software Process Improvement, EuroSPI 2005, Budapest, Hungary, pp. 95-105, Nov. 9-11,2005
15. Price, C. and Snooke, N., "An Automated Software FMEA," In Proc. International System Safety Regional Conference (ISSRC 2008), Singapore, April 2008
16. Dong, W., et al., "Automating Software FMEA via Formal Analysis of Dependence Relations," In Proc. 32nd Annual IEEE International Computer Software and Application Conference (COMPSAC 2008), Turku, Finland, pp. 490-491, July 28 – Aug 1, 2008.