Model-based Failure Modes and Effects Analysis of Software

Neal Snooke [1]

Abstract. Failure Mode and Effects Analysis is widely used in engineering hardware systems to help in understanding the effects of potential failures and the faults that cause them to occur. The analysis is iterative leading to improvements in the design until the risks associated with potential failure modes are reduced to an acceptable level. Interest in applying the technique to software has increased in recent years, particularly among embedded software developers who are already familiar with the benefits of FMEA analysis. Several authors discuss experiences of performing manual FMEA of software however there has been no attempt at automating the (very tedious) analysis. We describe a model based approach inspired by both work in diagnosis of software and the success of model based automated FMEA for hardware.

1 Introduction

The notion of Software Failure Modes and Effects (SFMEA) analysis has been proposed in the embedded systems community for at least two decades. The motivation has been both the success of FMEA as a standard (IEC 61508) technique to improve reliability and safety and the increasing contribution of software to many products. The inevitable consequence being the incresing proportion of safety and reliability problems attributable to software failure.

Traditional FMEA is a time consuming and tedious process that requires the effort of skilled engineers. Manual SFMEA promises to be even more expensive and has therefore only been documented industrially for systems such as aircraft [9], spacecraft [7], and military applications. Application of model-based reasoning (MBR) techniques has progressed automation of the analysis required to produce an FMEA to the point of deployment in the Automotive industry [12]. Recent research into MBR of software for diagnosis [16] provides the potential for the development of automated SFMEA methods.

2 SFMEA characteristics and benefits

FMEA assesses the risk of a significant failure of a system. For software there are three causes of failure:

1. Abnormal value input to the software from its environment.
2. Failure in the hardware upon which the software is executed.
3. Logical/algorithmic/semantic error in the implementation code (a bug).

The first cause is applicable to all software although it is important to note that a fault does not necessarily produce an out of range value. Input checking cannot therefore preclude FMEA, although it will help prevent certain faults causing catastrophic effects such as premature termination. This analysis has been attempted manually by tracing input variables through a system [5]. In another example some administrative automation is provided by a database to assist by extracting variable symbols from the code and providing support in producing the documentation [10].

The second cause is applicable to embedded microprocessors with limited memory error checking but does not apply to sophisticated architectures. Though important to some applications, these faults produce effects that are hard to predict and are not considered further in this work.

Code bugs are the third source of the failures that occur in well tested code, often after the system has been deployed for a long time. Such faults occur because a previously unused region of the behaviour of the software has been encountered. This may be due to some external change such as an upgrade to another system, or simply that the user has done something new given the viewpoint of the internal behaviour of the code. At a functional level the user may not even consider anything different has been attempted.

Given that formal methods are rarely applied, for a variety of reasons, and testing can never cover all states of non trivial software, much effort in S/W engineering is employed in developing sound architectures and design. As with other areas of engineering good basic design makes a successful product possible but does not guarentee it will work. As noted by [15] ‘Software architectures use abstractions which hide details that must be revealed for safety engineering’. For software the devil really is in the detail and while good design strategy such as high cohesion and low couplng will help to prevent small faults having serious consequences, it is the implementation details that determine the nature and consequences of a fault.

We can characterize a SFMEA as an analysis to determine the impact of hypothetical faults in the system. For software all faults may be represented as an abnormal value. The analysis is a worst case scenario and therefore should consider all possible effects of a faulty value. The main objective of SFMEA is to establish what the effects of a faulty value might be and not the programming error that lead to it. Once the risk associated with the fault has been assessed there are many tools and processes that may be used to modify the risk ranging from more testing to exteme measures like parallel execution of independently implemented systems or redesign using formal development methods. Generally the response might be to identify and remove unintended interactions and improve error checking for example.

This implies that the analysis should be at a level of abstration that does not care for example *when* a value might be used, how many times a loop may execute before an output is affected, which branch

[1] Computer Science Dept, University of Wales Aberystwyth ,UK email: nns@aber.ac.uk

of a conditional is taken, which particular instance of execution of a statement propagates a fault. SFMEA is not concerned with:

- Verification of nominal behaviour/functionality.
- Functional test generation or test execution.
- Dealing with dynamic behaviour or object instance details.

We propose that SFMEA will fill the gap between abstract metrics that assess overall design strategy, and testing that can only verify an small part of the system behaviour.

Several authors have documented the identification of input and output variables and the propagation of the source of each input to the destination of each output variable. Goddard [5] introduces the identification of software 'threads' as the data flow mechanism for tracing postulated failures to system effects. Lutz [7] uses a forward search to derive an effect for each potential faulty value and a backward search to examine the possibility of occurance of each failure mode. Bowles [1] also traces the effects of critical variable failures; however the manual analysis allows for a more detailed explanation of system level effect. For example 'if the set-point variable exceeds the allowed tolerance high the ball will stabilize at some point above the set point'. This level of reasoning considers the important thresholds in the system and links the system behaviour to each fault. This result would require knowledge of the software and of the external system to be controlled. Models of the system may be combined with an analysis of the software, but we leave this as future work. Finally Ozarin [10] uses a database to allow engineers to identify directly affected variables (ie local propagations), allowing the tracing of effects to be divided into small steps to avoid the analysis becoming too complex and effects being overlooked. A certain amount of automation is provided to extract variable names and modules from the code to allow easy selection from pull down lists. While this assists production the report, the analysis remains essentially a manual process.

3 Automated analysis

The Functional Dependency Model (FDM) is a result of recent work in software diagnosis [4, 8, 3, 16] and is suited to the task of propagating abstract faults in software. A model is generated from software source code (or possibly a graphical language) and represents statements as components linked by the variables forming the nodes of a dependency graph. Programming languages concentrate on the control flow and due to the reuse of memory (variable names) mask the data flow. The FDM provides each new assignment of a variable with a new index thus making the data flow explicit allowing propagation of faults. The result is a model that can predict which output variables are affected by a potentially faulty input or what the effect of a faulty internal variable might be.

A formal definition is provided in [8], here we suffice with an intuitive example produced from a code snippet provided by [10]. For the code in figure 1 the model shown in figure 2 is generated. Each successive asignment to a variable is provided with a unique subscript and the dependencies are tuples in the form (A, B) where B is the set of variables that A depends on. Generally the model is for internal processing and would not be presented in graphical form.

An engineer will use the results of fault propagation to determine the functional effects of the fault thus allowing an intelligible report to be produced. To achieve this a simple functional labeling system similar to the one presented in [11] can be adopted to interpret the behavioural outputs. Since we are not dealing with values it is simply necessary to identify the outputs that contribute to each function

```
    void sub1(void){
1.      E=0;
2.      A=B+C
3.      D=B*C
4.      J=1;
5.      if (A<0) {
6.          B=0
7.          C=0
8.          Call sub2()
            }
        else {
9.          E=(C*D)+F
        }
    }

10. void sub2(void){
11.     J=K-B
12.     if(B==0 || C==0){
13.         A=0
14.         F=2
        }
15.     G=J*F
    }
```

Figure 1. Example code

of the software. These outputs may be I/O ports, memory mapped devices, or calls to system functions that deal with input or output, for example.

4 Determining Risk Priority

Automated FMEA analysis is only useful if the results are at a level of abstraction that allows an engineer to easily assess each fault class and effect. Fault equivalence classes [13] allow condensed output by identification of all faults that lead to an identical effect. A Risk Priority Number (RPN) is assigned to each effect based on a product of the how **severe** the loss of functions is considered to be, how **detectable** the loss of function is (either to the end user or engineer), and an **occurence** value to indicate how likely the fault is to occur. Functional labeling not only allows abstraction of behaviour also provides a container for the Severity and Detectability knowledge of the effects to be asessed.

Internal software faults cannot be assigned failure probabilities as is done for hardware components. An analogy suggested by [14] is to use a complexity measure related to the code containing the cause of the fault, since complexity has been shown to be linearly correlated to defect rate. Part of the purpose of the analysis is to help determine where critical sections of code are located and such a complexity measure already attempts to determine this but without any real knowledge of the structure or dataflow. A preferred measure for occurence (that doesn't pre-empt the result) is suggested by [10] who uses the terms **possible** and **definite** to indicate the possibility that the fault *might* or *will* produce an observable effect at a threatened function output. This can be considered as a 3 level qualitative measure, and can be obtained by using a fault propagation model of each statement. There are 3 possibilities: $occ = 0$; $occ = 1$; $0 < occ < 1$. Situations where $occ = 0$ are not normally documented in an FMEA unless all outputs have zero likelihood of being affected by a fault (fault has no effect). In software this is likely to indicate that either dead or redundant[2] code has been detected or that faulty logic exists.

[2] code that is executed but whose results cannot be used to contribute to an output

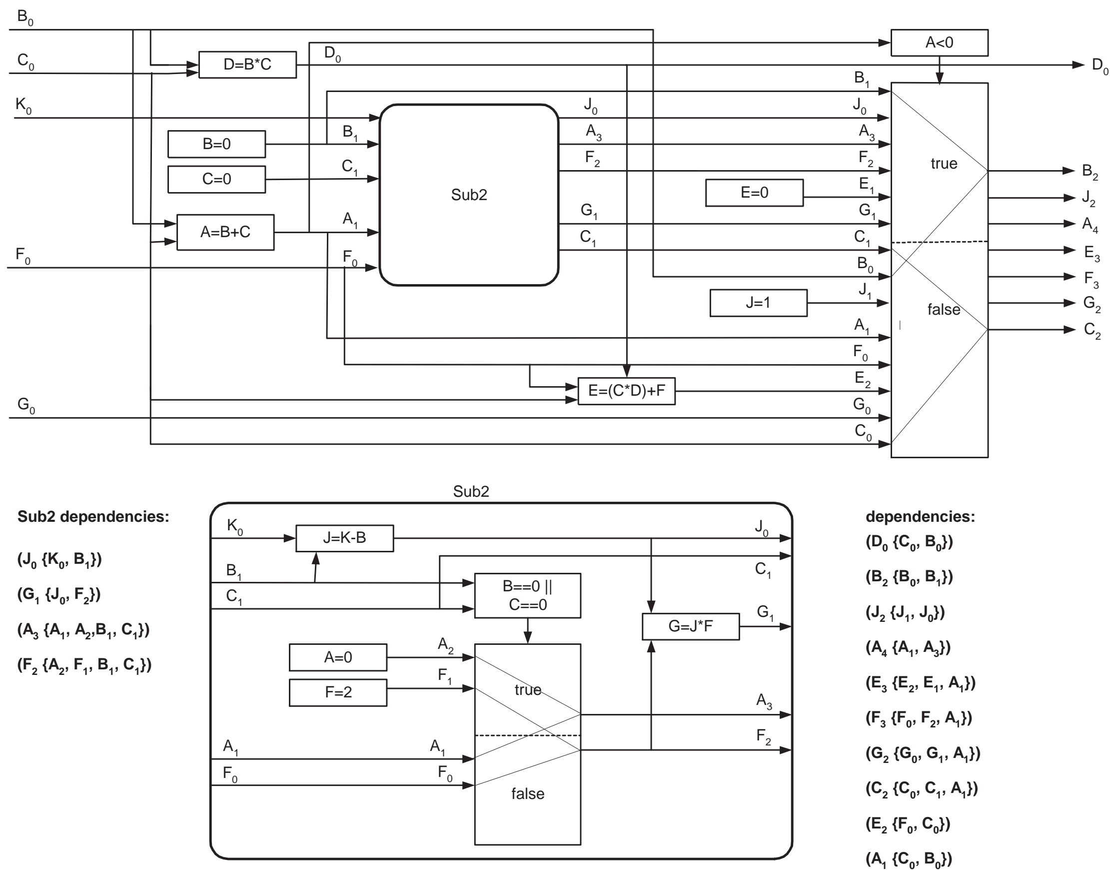

Figure 2. Example FDM

Each component in the model can affect the propagation of a faulty value. Assignment statements for example will always produce a faulty LHS given a faulty RHS. Some multiple input operators such as +, -, * also have this characteristic if a single input is suspect because there is a 1:1 mapping between fault input and output. If both inputs are suspect (ie $occ \neq 0$) there is the possibility that a correct result is produced [3] (simplest case for the commutative operations is that input values are swapped). The table describes the possibilities for the +,-,* operators.

Input a	Input b	output
definite	definite	possible
definite	possible	possible
definite	impossible	definite
possible	possible	possible
possible	impossible	possible
impossible	impossible	impossible

Conditional control statements tend to change definite faults into possible faults and require a more complex model. When the proposed fault does not affect the operation of the condition input, and hence does not alter the execution flow of the program, conditions for fault propagation (particularly the conditions that must exist to preserve **definite** faults) can be added to strengthen the results. This avoids conditional statements weakening the occurrence values and highlighting the specific conditions where a fault can be masked. One area of investigation will be to determine how useful (and complex) these conditions become when analysing the behaviour of a failure.

[3] It may be useful to characterize such situations as 'very likely' to provide finer granularity for occurence propagated through some operators if the ranges of each input are large enough but we do not pursue this here

5 Outline Example

Figure 3 shows the result of potential input failures using the model in figure 2. The occurence values for input variables are provided by an engineer based on the likelihood of the input being faulty and may be derived from another FMEA or statistical hardware component failure information.

The RPN value provides an indication of the faults that potentially cause the most problematic effects and it can be seen that faults on inputs B and C both belong to the same equivalence class although they do have different risk priority. If required, an explanation can be provided for specific faults by means of the paths through the code shown by a program slice or the conditions that lead to a definite fault as shown in figure 4

6 Detailed internal faults – bugs

Mutation testing is an approach used to improve the quality of testing by inserting 'random' faults in code and executing it. If one or more tests fail then the proposed fault (mutation) is 'killed' and confidence is increased that the test suite covers the area of the code appropriatly, otherwise a new test is required that will fail in the presence of the

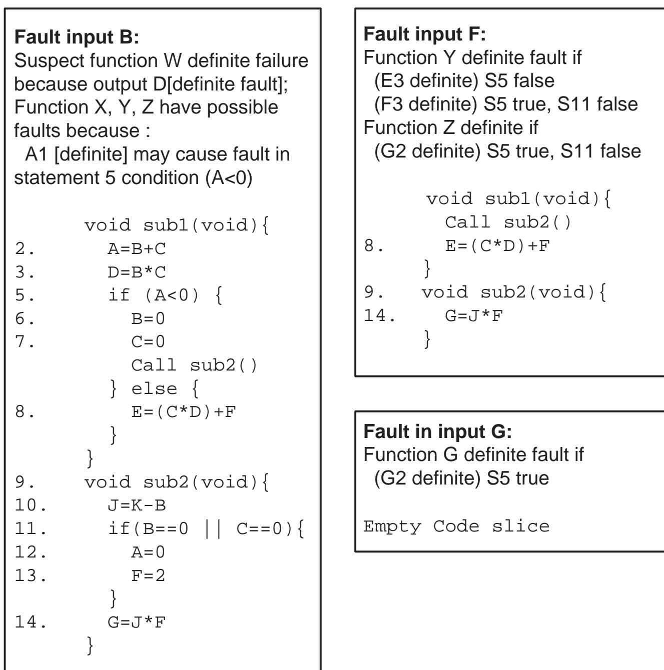

Figure 4. Program slices

mutated program. At its most detailed level FMEA is related to mutation testing since latent faults exist in software although they may *appear* to develop over time because the *use* of the software changes sightly, causing an untested and faulty fragment of the behaviour space to start affecting system function. Mutations can be viewed as proposed fixes to potential (undiscovered) faulty behaviours.

Given a subset of the variables that the FMEA has shown to be accociated with a high RPN, the statements that produce these variables can be considered as high risk and worth considering as potential mutants. Assuming that tests of the software execution can be captured, the possibility of considering some categories of possible internal faults at the level of faulty statements can be investigated. The aim is to detect possible faults not found by compilers. Faults can be generated from simple rules that analyse the code and may include functional modifications such as operator substitution, and minor structural changes such as use of an incorrect variable with a 'similar' symbol, or appearing on adjacent lines in the code. The generation heuristics that can create mutations that include plausible bugs is somewhat dependent on the language and is examined in the literature [6]. Clearly certain faults are precluded in more strongly typed/scoped languages and faults prevented by compilers are not considered.

The large number of possibilities precludes manual consideration of the possible effects of each proposed fault. The aim however is to use verified tests (ie. capture executions of the system where an engineer has indicated that correct behaviour was obtained) to exonerate the majority of the potential faults by showing that the verified output would *not* have been achieved in the presence of any of the proposed mutations. The vast majority of mutations will either:

- allow tests to be dismissed immediately because no behaviour difference is found following the mutant statement or the statement is not executed during the test.
- cause tests to produce a different value that will propagate rapidly to an output. A minimum number of tests should therefore be required since one test failure is enough to 'kill' the proposed mutant.
- cause tests to produce a different value or cause a change in control. Many other variables are likely to be affected, only one of which needs to reach an output to kill the mutant.

Therefore the only computationally expensive mutants are those that produce a non-local behaviour difference for many tests that do not propagate to an output. This is the situation when a value is stored internally or a state change made that does not contribute to an output during the test. The FMEA however can explain what code must be executed to make the fault visible thus providing a hint as to an enhanced test.

The remaining possibilities either form equivalent code, indicate undetected faults, or poor test coverage. For these the model can provide information about the potential effects of the failure and conditions that must exist to allow the fault to be made visible.

6.1 Generating Tests

Development methods may include low level tests as part of the basic functional testing such as JUnit tests in Java.

A mechanism to capture the variable values during execution of the test is required to enable local propagation through the mutated model. The mechanism to achieve this is present in debugging tools; however there are some practical issues because of the quantity of data that could potentially be generated and the need to avoid storage of repeated execution of statements with the same values.

For embeddeded types of system we might capture executions of the software in the simulated environments that are often used to provide input values and model the hardware aspects of the system. In this situation an engineer simply needs to verify that the software functions correctly (ie all outputs had the expected behaviour). It is worth noting that a single execution may produce many tests of a single statement. This is useful where the values are different but produces a requirement to find a compact representation of an execution log continaing repeated values.

6.2 Exonerating Mutations

The internal faults considered thus far modify behaviour for every execution of a section of code. (Although the two behaviours might have a significant overlap, for example $<$ used where $<=$ intended) Certain faults, for example run time faults that are often captured via exception mechanisms, occur because an undefined (or unintended) part of the behaviour of an otherwise correct statement is being used. One example of this could be overflow of an 8 bit counter during the increment causing its value to be reset to zero. In this situation a potential failure mode of increment operator behaviour will occur if the value 255 occurs. Since the system will be designed to avoid these failures by not operating in this region of behaviour, it is not likely that a test will exercise them, therefore the only way to exonerate them is to show that they cannot occur within the specification of the system. In some cases value range propagation may achieve this. For the remainder there is either a potential problem, complex logic that prevents the problem, or unspecified application constraints. It would appear that all of these are worth highlighting to an enginer so that the constraints/assumptions can be made explicit or the documentation updated to explain assumptions so they are understood during subsequent maintenance of the code.

Cause	Occ	Failure Mode	Effect Summary	Sev	Det	RPN
Input B	4	D [d], B, J, E, F, G, C	Function **W[d]**, X, Y, Z	4	8	128
Input C	7	D [d], B, J, E, F, G, C	Function **W[d]**, X, Y, Z	4	8	224
Input F	2	F, E, G	Function Y, **Z**	1	9	18
Input G	3	G	Function **Z**	1	9	27
Input K	3	G, J	Function **X**, Z	6	2	36

Function W: Sev. 4. Det 8. output D
Function X: Sev. 6. Det 2. output J
Function Y : Sev. 2. Det 1. output F, E
Function Z: Sev. 1. Det 9 output G

[d] indicates definite fault
[p] indicated posssible fault

Figure 3. Example outline FMEA for inputs shown in figure 2

		Test1	*Test2*	*Test3*
Relevant Inputs: B_0, C_0, Relevant outputs: D_0		C_0=2, B_0=0, E_0=2, D_0=0	C_0=2, B_0=1, E_0=3, D_0=1	C_0=2, B_0=2, E_0=3, D_0=4
example mutations from D=B*C	D=C*B	D_0=0	D_0=1	D_0=4
	D=B+C	**D_0=2**	***D_0=3***	*D_0=4*
	D=B-C	**D_0=-2**	***D_0=-1***	***D_0=0***
	D=E*C	**D_0=4**	***D_0=6***	***D_0=6***
Relevant Inputs: A_1, C_0, D_0, F_0 Relevant outputs: E_3		C_0=2, F_0=0, D_0=3, A_1=-1, masked fault (E_3=0)	C_0=2, F_0=0, D_0=3, A_1=1, E_3=6	C_0=2, F_0=3, D_0=2, A_1=3, E_3=7
example mutations from E=(C*D)+F	E=D*(C-F)	E_2=6, E_3=0	E_2=6, E_3=6	**E_2=-2, E_3=-2**
	E=(C*D)-F	E_2=6, E_3=0	E_2=6, E_3=6	**E_2=1, E_3=1**
	E=(C+D)-F	**E_2=5,** E_3=0	**E_2=5, E_3=5**	***E_2=1, E_3=1***
	E=(C*D)-E	**E_2= undef**, E_3=0	E_2= undef, E_3=undef	E_2= undef, E_3=undef

Figure 5. Example mutation analysis

6.3 Mutations in the example

Extending the analysis to include each of the internal variables allows the use of occurence values from the 3 levels identified in section 4. One posibility is to map the 'possible' level to 5 and 'definite' to 10. A fault in D_0 for example causes a definite fault in function W only giving an RPN of 8*4*10 = 320. E_2 causes a possible fault in function Y and an RPN of 2*1*5 = 10. Where a statement contributes to more than one function the highest risk value is taken. A threshold may be set to select variables for further analysis that have the highest risk potential. The statements that generate each selected variable are used to generate mutations that represent both statement level behaviour changes, such as operator substitution, and structural changes such as variable substitution.

Figure 5 shows the effect of using 3 tests to investigate 4 possible mutations of statement 3. The first test exonerates (in bold type) all but one mutant. The second and third tests fail to remove the mutant; however, notice that it is not necessary to consider those already eliminated (shown *emphasized*). In this example we have found an exactly equivalent program. In practice the mutant generation heuristics would avoid generating such obvious examples.

The second statement (9) mutated in figure 5 only affects function Y. The occurence value from a suspect statement was identified on a three level scale and for a fault in Statement 9 E_2 is definite E_3 is possible making and occurance of 10 (2*1*5=10). According to the FMEA input C provides a higher risk for function Y than a failure of statement 9 and therefore we may decide it is not a priority for mutation testing. Several possible mutants are shown in figure 5 to illustrate several tests that eliminate all the selected mutants.

7 Limitation and further work

Several aspects of the FDM model are not covered by the example. Most notably iteration statements can be transformed into a sequence of conditional to preserve an acyclic FD graph. Alternatively by allowing cycles in the graph and using a traversal algorithm that detects cyclic behaviour in the occurence propagation the close mapping to program statements can be preserved and will be discussed in a future paper.

Real software also contains features including dynamic data structures, pointers/ aliasing, and recursion are not considered here. Some of these features have already been considered in the context of model based debugging and the next step will be to determine the characteristics relevant to SFMEA. The proposed SFMEA approach is applicable to any imperative software; it should be possible to generate the models from many high and low level languages given the established work in program transformation that supports data flow analysis [2]. Some applications such as embedded controllers are particularly suitable for initial investigation. They have a small and simple interface (via hardware connections), well defined and reasonably distinct functions, and often no dynamic data structures.

Predicting the nature of failure modes that include timing and control loop stability are relevant to embedded systems and have previously proven problematic for a manual software FMEA. A detailed knowledge of the dynamic behaviour is required and the proposed models cannot provide this. Given that verification of the nominal operation of the system is not part of an FMEA, knowledge regarding the potential function failures is enough detail and further investigation of exactly how behaviour might be modified is part of the investigation that may be carried out upon consideration of the FMEA result.

Many of the programs used in embedded systems are auto coded from state chart style models. These descriptions also hide the use and reuse of variables and a future step will be to analyse failure propagation in these models.

REFERENCES

[1] J. B. Bowles, 'Failure modes and effects analysis for a small embedded control system', in *Annual Reliability and Maintainability Symposium*, pp. 1–6. IEEE, (January 2001).

[2] J. Collard, *Reasoning about program transformations: imperative programming and flow of data*, Springer-Verlag, ISBN 0-387-95391-4, 1st edn., 2003.

[3] L. Console, G. Friedrich, and D. Dupre, 'Model-based diagnosis meet error diagnosis in logic programs', in *IJCAI 93*, pp. 1494–1499, Chambery, (August 1993). Morgan Kaufmann.

[4] G. Friedrich, M. Stumptner, and F. Wotawa, 'Model-based diagnosis of hardware designs', *Artificial Intelligence*, **111**(2), 3–39, (1999).

[5] P. L. Goddard, 'Software fmea techniques', in *Reliability and Maintainability Symposium*, pp. 118–123. IEEE, IEEE, (January 2000).

[6] W. E. Howden, 'Weak mutation testing and completeness of test sets', *IEEE transactions of Software Engineering*, **8**(4), 371–379, (1982).

[7] R. Lutz and R. Woodhouse, 'Experience report: Contributions to sfmea requirements analysis', in *2nd International Conference on Requirements Engineering*, pp. 44–51, available from http://ieeexplore.ieee.org/, (April 1996). IEEE.

[8] C. Mateis, M. Stumptner, and F. Wotawa, 'Debugging of java programs using a model-based approach', in *10th International Workshop on the Principles of Diagnosis (DX'99)*, pp. 166–173, (June 1999).

[9] D. Nguyen, 'Failure modes and effects analysis for software', in *Annual Reliability and Maintainability Symposium*, pp. 219–222. IEEE, (January 2001).

[10] N. Ozarin and M. Siracusa, 'A process for failure modes and effects analysis of computer software', in *Annual Reliability and Maintainability Symposium*. IEEE, (January 2002).

[11] C. J. Price, 'Function-directed electrical design analysis', *Artificial Intelligence in Engineering*, **12**(4), 445–456, (1998).

[12] C. J. Price, D. R. Pugh, N. A. Snooke, J. E. Hunt, and M. S. Wilson, 'Combining functional and structural reasoning for safety analysis of electrical designs', *Knowledge Engineering Review*, **12**(3), 271–287, (1997).

[13] C. S. Spangler, 'Equivalence relations within the failure mode and effects analysis', in *RAMS 99*. IEEE, IEEE Press, (January 1999).

[14] S.R.Luker, 'Failure mode, effects and criticality analysis (fmeca) for software', *5th Fleet Maintenance Symposium*, 731–735, (Oct 1995). Virginia Beach, VA (USA).

[15] K. Wong, 'Looking at code with your safety goggles on', in *Ada-Europe International Conference on Reliable Software Technologies (LNCS 1411)*, pp. 251–262. Springer, (1996).

[16] F. Wotawa, 'Analysing models for software debugging', in *12th International Workshop on the Principles of Diagnosis (DX'01)*, pp. 197–204, (June 2001).