Combining Functional Modelling and Qualitative Fault Propagation to Enable Failure Mode Analysis of Software Systems

Neal Snooke and **Jonathan Bell** [1]

Abstract.
Failure Modes and Effects Analysis (FMEA) is a widely used reliability engineering method that provides direct product based evidence of quality, however its application to software is infeasible without automated analysis tools. This paper describes the application of a functional modelling method that when combined with a component - connection model of software, and qualitative fault propagation, enables automated Software FMEA (SFMEA). The result is a broad analysis of fault behaviour using abstract models directly produced from the implementation. A small example code fragment demonstrates the nature of the analysis provided by this novel technique and illustrates how the results could be used.

1 Introduction

The increase in the use of software within the electromechanical engineering community where failure analysis is widely used has produced interest in extending FMEA into the software itself [3, 14, 6, 8] however there has been little progress in developing an FMEA style analysis for general use in software engineering.

This paper proposes that SFMEA will assist software engineers to improve the uniformity of software quality by providing an analysis that is based on a detailed analysis of the implementation behaviour while providing a broad and abstract consideration of hypothetical fault effects. With suitable tools a SFMEA will provide the ability to find potential problems that may not be detected by either testing or high level design analysis. The paper is structures as follows. Firstly, the the main characteristics distinguishing the proposed SFMEA from other software analysis methods are discussed in section 2. We then describe a functional interpretation model in section 3 recently developed for hardware analysis and outline our use of it for software. A running example is developed throughout the paper to illustrate the concepts and concludes in a complete automated FMEA illustration. Section 4 will describe a qualitative fault propagation based analysis method for software behaviour. The FMEA analysis algorithm is then described in section 5 leading to the production and discussion of failure modes in section 6.

[1] University of Wales,Aberystwyth, United Kingdom email:nns@aber.ac.uk

2 A novel software analysis

The achievement of software quality levels is almost exclusively based around standards that define development *processes* and practices. While development processes are vitally important (in any engineering endeavour) they do not guarantee the quality of a product. As observed [12] we should look for *evidence* of software safety and quality. To perform such a task requires identification of the functional requirements followed by an analysis of the possible failure modes of the product, mapped to the effects on the safety / quality requirements. Testing is often the only evidence based analysis performed and although testing can *initially* give indications of quality, once the results are used to improve the software, the tests are no longer a reliable indication of overall quality. It is likely that the system outside of the scope of the actual tests performed is near the quality achieved at the very first execution of the code. Put another way, testing only finds faults, it does not help prevent or assess them. The reason for this is clear. Tests can only ever verify a tiny sample of the expected behavioural envelope of the software and they are often focused on verification of nominal function.

The idea of FMEA for software may appear to be a contradiction. Software codes do not wear out or suffer hidden manufacturing (replication) faults and since in essence software is pure logic, it is easy to be seduced into believing it can and will be perfect. In contrast, all physical system engineers accept that both their product and the environment it operates in will be subject to failure and experience shows that (although it is less readily accepted) this also the case for software. Formal methods are often proposed as the solution to the problem however they can be difficult to use, specification capture remains a problem, and pragmatically these methods are too expensive for the majority of software. All substantial software will enter unanticipated regions of behaviour during its life and all software will have to deal with external failures (either from hardware or other software) and of course no matter how much range and consistency checking is done it will always be possible for an incorrect but valid input to exist and it is useful to know, when this happens, how badly it may affect the system.

An FMEA aims to locate the worst case effects of a comprehensive set of hypothetical faults or system failure modes. FMEA does not verify functionality - that is a primary role of testing - a nominally functioning system is assumed and the

analysis is only concerned with the effects of potential internal or external faults. This requires a more abstract analysis than testing. In particular code execution is not feasible and as far as possible specific values are not considered unless they have a very special behavioural significance. Traditional FMEA tasks have benefited from qualitative techniques [15, 10] that allow prediction of worst case effects for whole regions of system behaviour, since even non software systems may have too many parameters and attributes to test all numerical value permutations.

All software processes input values to produce output values and therefore the most concrete form of any fault is its effect on the production of output. Faults may take a number of forms with basic faults concerning missing, unexpected, mistimed or incorrect values. Some applications will benefit from higher level abstraction and interpretation of these outputs. For example resource problems such as lack of processor performance or memory may affect output timing and require an in depth understanding of the behaviour of the computational infrastructure, such as the operating system, and we will not further consider specialised faults in this paper although they clearly form part of a comprehensive system wide FMEA and may be addressed with the use of suitable models of the relevant computational infrastructure.

Part of the reason why faults are less predictable in software is due to the complex structuring that is possible, whereas physical constraints tend to limit structural complexity in other engineering disciplines. For example physical components require spatial partitioning and material strength considerations together with many other aspects that lead to visible layers of structuring and separation of function. Modern software languages enforce high levels of structuring and this helps to inherently constrain faults. Decades ago high level languages introduced typed data and procedures to help structure data and code and these have been refined ever since. Recently OO methods help structure and partition data with code. These methods make it possible to successfully construct larger systems and for the same reasons make an FMEA analysis useful in that potential fault impacts are constrained[2]. For example, in figure 1, a language has been used that separates memory into data and instructions and prevents data from being executed. Moreover, structuring of the instructions ensures that faults in some groups of instructions can only affect some output (functionality). Also the language allows the heap to be partitioned, preventing faults in some locations from propagating to others. Languages that support facilities such as arbitrary pointer arithmetic clearly allow faults to propagate very widely. In the extreme, assembly language code allows any part of the memory to be executed. This allows virtually any fault to have unlimited effects and thus an FMEA of the code could not make any effect predictions.

In summary SFMEA provides the following characteristics:

- an evidence based analysis that deals with a wide range of possible behaviours at a lower level of precision, unlike testing which provides precise behaviour with limited behaviour coverage.
- it assumes a wide range of hypothetical faults *could* occur and aims to determine the possible effects and assign significance to them. Any statement, module or subsystem is assumed to have the potential to fail or produce incorrect output and the task is to prioritise the results to identify design or implementation issues that might lead to faults having disproportionate or unexpected consequences. Engineering effort can then be expended to address the most significant faults.
- aim to attain better uniformity of quality and identify areas of design or implementation that allow significant failures to occur.
- analysis can be carried out at any stage of the design - the earlier the better - however for this paper we will analyse the source code to correlate the potential effects of faults in the system as implemented with the effects that are to be expected from the high level design and functional specification.

[2] Malicious programs often exploit weaknesses in languages and compilers to circumvent this structure and SFMEA does not intend to address such issues.

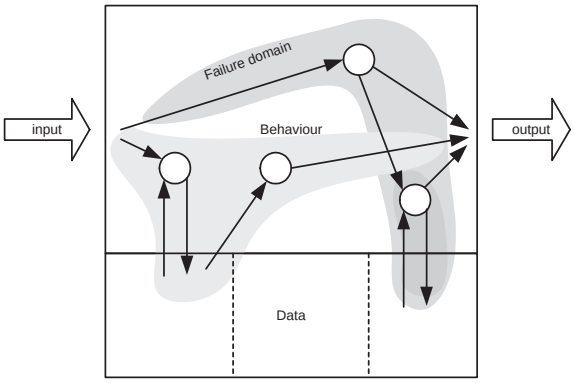

Figure 1. Failure domain localisation in software

3 Function Identification

Functional[3] interpretation is a vital part of organising and abstracting the results of structural and behavioural analysis into the form an FMEA able to identify significant potential issues. We use a definition of function described by [2] with some additions (shown emphasised).

> An object *or system* O has a function f if it achieves an intended goal(purpose) by virtue of some external *behavioural* trigger T resulting in the achievement of an external *behavioural* effect E.

The predicate $\mathrm{Tr}(f)$ indicates that function f is triggered and $\mathrm{Ef}(f)$ provides the truth of the effect. The four possible function states: *achieved*; *inoperative*; *failed* and *unexpected*; are provided by the predicates $\mathrm{Ac}(f), \mathrm{In}(f), \mathrm{Fa}(f), \mathrm{Un}(f)$ respectively:

$$\mathrm{In}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \neg\mathrm{Ef}(f)$$

$$\mathrm{Fa}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \neg\mathrm{Ef}(f)$$

$$\mathrm{Un}(f) \Leftrightarrow \neg\mathrm{Tr}(f) \wedge \mathrm{Ef}(f)$$

$$\mathrm{Ac}(f) \Leftrightarrow \mathrm{Tr}(f) \wedge \mathrm{Ef}(f)$$

[3] Here we use function in its teleological sense as distinct from its use as a type of subroutine. In this paper the term method is used to indicate such structural aggregations.

Functions may be decomposed into subsidiary functions[2] to build a functional hierarchy. These decomposed functions may be incompletely specified and fall into 3 categories that specify a relationship between any pair of trigger, effect and purpose. Some software may require use of a functional model that includes a range of subsidiary function types. For the purposes of a simple example we will consider the trigger to be program execution and therefore all functions share the same trigger and the only required subsidiary functions relate effects to purpose using a purposive incomplete function (PIF). The running example will involve a simple expenses payment program with the following functions:

```
FUNCTION pay_expenses
    ACHIEVES pay money owed
    BY
        run_program
        TRIGGERS
            PIF display_expenses_total
            AND
            PIF print_cheque

FUNCTION make_envelope
    ACHIEVES allow delivery of expenses cheque
    BY run_program TRIGGERS address_envelope

PIF display_expenses_total
    ACHIEVES verify amount
    BY
        show_expense_items_on_screen
        AND
        show_total_on_screen

PIF print_cheque
    ACHIEVES make payment
    BY print_in_figures AND print_in_text AND print_details
```

Notice here that in general the conjunction of two subsidiary functions does not mean both sub functions are required in any specific execution. If they are complete sub functions with triggers and effects it is the triggers that determine which are produced for a specific execution. If a fault causes one sub function to fail the parent function fails but this does not mean the system fails for any specific execution, but it will fail due to the faulty sub function sometime.

3.1 Function failure risk

Generally for FMEA, a Risk Priority Number (RPN) is calculated for each potential fault as the product of maximum severity, detectability and occurrence values [13, 9]. The absolute values of the numbers have little significance however they provide a summary, allowing ranking and prioritisation of perceived risk. For hardware systems an occurrence value is used to indicate component reliability by providing likelihood values for each fault. For external faults related to hardware this approach should be possible, however for faulty software elements occurrence is much more difficult to determine. It has been suggested [11] that we might use software metrics such as code complexity to provide occurrence values since complexity has been shown to be linearly correlated to defect rate. We use only severity and detection values until further investigation has been carried out. For the purpose of FMEA we require the severity and detectability of each potential function failure and this information is provided within the function description using the FAILURE CONSEQUENCE, SEVERITY and DETECTABILITY. For the expenses example we may enhance the function model as follows:

```
FUNCTION pay_expenses ACHIEVES pay money owed
        FAILURE CONSEQUENCE incorrect expenses payment
        SEVERITY 9 DETECTABILITY 5

FUNCTION display_expenses_total ACHIEVES verify expense...
        FAILURE CONSEQUENCE check expense total manually
        SEVERITY 5 DETECTABILITY 6

FUNCTION print_cheque ACHIEVES transfer money
        FAILURE CONSEQUENCE Cheques to be written by hand
        SEVERITY 8 DETECTABILITY 9

FUNCTION make_envelope ACHIEVES allow delivery of cheque
        FAILURE CONSEQUENCE Cheques to be delivered by hand
        SEVERITY 6 DETECTABILITY 3
```

4 Fault propagation model

The behaviour of the system and fault effects are generated with a fault propagation model outlined in this section. We first convert the program into Single Assignment Form [7] to clarify fault *propagation through memory* by allowing each symbolic location to be written once only by adding a subscript to each variable. An oracle (ϕ) 'function' is used to select from several variables and produce a result variable subscripted Φ, where non linear code requires execution information.

The analysis uses qualitative fault propagation. This notion is supported by providing each variable with a fault domain, $\mathbb{F} = \{\mathbb{D}, \mathbb{P}, \mathbb{I}\}$, where $\mathbb{D}$ is a definite fault, $\mathbb{P}$ is a possible fault, and $\mathbb{I}$ is not faulty. Even if only "definite" (certain) faults are inserted in a program the propagation may result in "possible" faults because value information is not used. Typically this indicates that for some executions the value is faulty and for other executions it is not. In this section we model how faults may be combined by *propagation through statements* to provide results for the fault function $\mathcal{F}(\mathtt{v}) : \mathcal{M} \mapsto \mathbb{F}$.

A fault propagation graph for program p is defined $\mathcal{FP} = \langle \mathcal{M}, A \rangle$ treating memory $\mathcal{M}$ as vertices and statement fragments A as directed arcs. If $\mathtt{x}, \mathtt{y} \in \mathcal{M}$ an arc $(\mathtt{x}, \mathtt{y}) \in A$ exists if a statement creates a potential fault propagation from $\mathtt{x}$ to $\mathtt{y}$. Each statement s responsible for a sub graph:

$$\exists \mathcal{M}_s \subseteq \mathcal{M}, \exists \mathrm{A}_s \subseteq A : \mathcal{FP}_s = \langle \mathcal{M}_s, A_s \rangle$$

This sub graph represents the structure of the fault propagation model for the statement.

Considering statement 3 from figure 2 as an example we get for $\mathcal{FP}_3$, $\mathcal{M}_3 = \{\mathtt{a}_1, \mathtt{a}_2, \mathtt{b}_1\}$ and $A_3 = \{(\mathtt{a}_1, \mathtt{a}_2)(\mathtt{b}_1, \mathtt{a}_2)\}$ representing the fact that the new value of a depends on the previous value of a and also b. The behaviour of the fault propagation for qualitative faults requires knowledge of the semantics of the statement.

For a statement $\mathcal{FP}_s = \langle \mathcal{M}_s, A_s \rangle$ we define a fault propagation mapping between the variables $\mathcal{M}_s$ based on the state-

```
1,2   a=0; b=2;
3     a=a+b;
4     for i = 0 to x {

5       if (j<constant){

6         c=b;
7         c=c*2;
        }
8       c=c+1;
9       b=a;
      }
```

```
a1=1; b1=1;
a2=a1+b1;
iterate {n = {1..x⊥},
  b2▷0 ≡ b1, c1▷0 ≡ c⊥
  if1=(j1<aconstant)
  condition{if1
    c1▷n=b2▷(n−1);
    c2▷n=c1▷n*2;
  } cΦ1▷n=φ( c1▷(n−1), c2▷n)
  c3▷n=cΦ1▷n+1;
  b3▷n=a2;
} bΦ1=φ( b1, b3▷N)
  cΦ2=φ( c1, c3▷N)
```

Figure 2. Code Fragment and SAF

ment type and operators. For the basic math operators requiring 2 operands $\mathcal{F}(\mathtt{m}), \mathcal{F}(\mathtt{n})$, producing a result $\mathcal{F}(\mathtt{r})$ then $\mathcal{FP} = \langle\{\mathtt{m},\mathtt{n},\mathtt{r}\},\{(\mathtt{m},\mathtt{r}),(\mathtt{n},\mathtt{r})\}\rangle$ and the fault mapping model provides $\mathbb{F}\times\mathbb{F}\mapsto\mathbb{F}$. This is illustrated for a selection of basic operators in table 3. More complex expressions have fault propagation tables generated by combining primitive operators according to the structure of the statement provided by the AST.

$\mathcal{F}(\mathtt{m})$	$\mathcal{F}(\mathtt{n})$	$+,-,*,\&,\lvert$ $\mathcal{F}(\mathtt{x})$	$\frac{\mathrm{m}}{\mathrm{n}}$	$\mathrm{m}<\mathrm{n}$
$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$
$\mathbb{I}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$
$\mathbb{I}$	$\mathbb{D}$	$\mathbb{D}$	$\mathbb{P}$	$\mathbb{P}$
$\mathbb{P}$	$\mathbb{I}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$
$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$
$\mathbb{P}$	$\mathbb{D}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$
$\mathbb{D}$	$\mathbb{I}$	$\mathbb{D}$	$\mathbb{D}$	$\mathbb{P}$
$\mathbb{D}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$
$\mathbb{D}$	$\mathbb{D}$	$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$

Figure 3. Fault model

4.1 Condition statement propagation

For each variable x assigned within the sub blocks of a condition we have the graph fragment:

$$\mathcal{FP}_{cond} = \langle\{\mathtt{x}_T,\mathtt{x}_F,\mathtt{x}_\Phi\},\{(\mathtt{x}_T,\mathtt{x}_\Phi)(\mathtt{x}_F,\mathtt{x}_\Phi)\}\rangle$$

In addition faults in the condition value:

$$\mathcal{FP}_{cond} = \langle\{\mathrm{cond},\mathtt{x}_\Phi\},\{(\mathrm{cond},\mathtt{x}_\Phi)\}\rangle$$

Figure 4 shows the fault value propagation model. The right hand column includes constraints in parenthesis that can be used to strengthen propagations in specific circumstances. The top part of figure 5 shows the graph for the conditional statement in example 2.

cond	$\mathtt{x}_T$	$\mathtt{x}_F$	$\mathtt{x}_{\Phi 1}$
$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$
$\mathbb{I}$	$\mathbb{I}$	$\mathbb{P}/\mathbb{D}$	$\mathbb{P}/\mathbb{D}$ ($\mathbb{I}$ if$_1$)
$\mathbb{I}$	$\mathbb{P}/\mathbb{D}$	$\mathbb{I}$	$\mathbb{P}/\mathbb{D}$ ($\mathbb{I}$ $\overline{\text{if}_1}$)
$\mathbb{I}$	$\mathbb{P}/\mathbb{D}$	$\mathbb{P}/\mathbb{D}$	$\mathbb{P}/\mathbb{D}$
$\mathbb{I}$	$\mathbb{P}/\mathbb{D}$	$\mathbb{D}/\mathbb{P}$	$\mathbb{P}(\mathbb{D}$ if$_1)$/ $\mathbb{P}(\mathbb{D}$ $\overline{\text{if}_1})$
$\mathbb{P}/\mathbb{D}$			$\mathbb{P}$

Figure 4. Fault model for condition

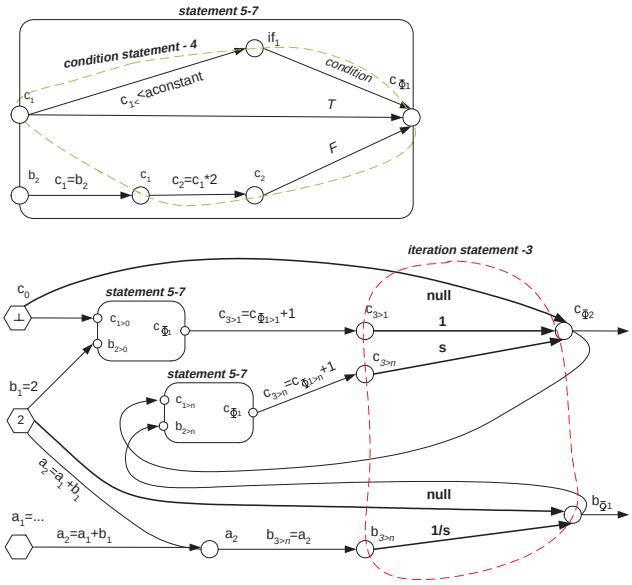

Figure 5. Propagation model for example code fragment

4.2 Iteration statement propagation

Loops may produce a set of variables for a single assignment statement. The loop statement produces a variable ($\mathtt{x}_{\Phi m}$) for each accessible variable written in the statement block. The loop has three significant propagation links covering the non execution, single, and multiple executions for each variable. Multiple executions require the model to include propagation of values from the current or previous iteration only[4] (if x_b is any variable) as follows:

$$A_{loop} = \begin{cases} (\mathtt{x}_\mathtt{a},\mathtt{x}_\Phi) & \text{if null loop} \\ (\mathtt{x}_{\mathtt{a}\triangleright 1},\mathtt{x}_\Phi) & \text{if single iteration} \\ (\mathtt{x}_{\mathtt{a}\triangleright(n-1)},\mathrm{x}_{b\triangleright n}) & \text{if multiple iterations} \\ (\mathtt{x}_{\mathtt{a}\triangleright n},\mathtt{x}_\Phi) & \end{cases}$$

The lower section of figure 5 shows the fault propagation model for the code in figure 2. By graph traversal we find:

$$A_{loop} = (\mathtt{a}_1,\mathtt{b}_{\Phi 1})(\mathtt{b}_1,\mathtt{b}_{\Phi 1})(\mathtt{a}_1,\mathtt{c}_{\Phi 2})(\mathtt{b}_1,\mathtt{c}_{\Phi 2})(\mathtt{c}_\perp,\mathtt{c}_{\Phi 2})$$

[4] This remains true if the loop contains conditions, since the conditional statement always produces a new SAF output for each iteration.

4.3 Modular structures

Method and procedure definitions represent a propagation sub graph that is conceptually inserted for each calling statement occurrence.

We propagate faults down through the abstract call tree (ACT) using breadth first propagation, propagating as far as possible at each level before inserting faults into subroutines. Clearly some sections of the fault graph will be traversed more than once both due to multiple calls instances and also when additional faults appear at block inputs to a previously traversed block. Using the qualitative faults, fault introduction is monotonic in the sense that the fault propagation can only map faults as follows: $\mathbb{I} \xrightarrow{s\in\mathcal{S}} \{\mathbb{I}|\mathbb{P}|\mathbb{D}\}$, $\mathbb{D} \xrightarrow{s\in\mathcal{S}} \{\mathbb{P}|\mathbb{D}\}$, $\mathbb{P} \xrightarrow{s\in\mathcal{S}} \mathbb{P}$. The order of propagation therefore does not affect the result but may improve efficiency.

The qualitative nature of the analysis and fault mode behaviour makes it possible that the same permutation of faults occur often making reuse of propagation results at the method level a possible an area of future work.

4.4 Linking functions to behaviour

The functional model must be linked to the behaviour of the program via the triggers and effects. For example triggers may be an interrupt or the execution of the code representing an event in high level programs. For our simple expenses program the trigger for all of the program functions is simply the action of running the program, which we consider as execution of the first line of code. The effects will be system outputs and at the lowest level could be memory locations or ports that influence external devices. At higher levels a system call and its parameters may be considered as the output. All of these forms of output can be treated as the execution of code that assigns to a suitable set of variables in the ACT. In the simple expenses example, effects are linked to the variables that represent the payment amount and employee number values that are passed to output devices such as the display or printer. Notice that the engineer must decide on the system boundary. For example, library code is often not of concern since it is either trusted, has already had a separate FMEA performed, or the code is not available. A concrete case is provided by a statement (in Java) such as `System.out.println(x)` whereby an output is attached to the variable $\mathtt{x}_\mathtt{n}$ accessed as the actual parameter in the external `println` method.

Single variables are sometimes not enough to capture the effects of a function and we therefore allow multiple variable expressions to determine the truth of $\mathrm{Ef}(f)$. Logical operators are used to combine variables where more than one output provides the effect required of a function. Figure 6 shows how the fault propagation results are combined.

Notice a disjunction of effects does *not* mean that in some circumstances one effect is expected and in others the other is expected. In that situation we actually have two functions with different triggers and the trigger is the differentiating circumstance. If a fault occurs with only one element of a disjunction the function will *always* still be achieved. Most software examples where we appear to have one effect OR another effect are in fact two conjoined sub functions with different triggers.

The code extract in figure 7 could implement the expenses payment problem. To avoid (in this compressed example) having to consider external library functions for I/O we treat assignment to some specific variables as the output of the program. In this example the show expenses items on screen and show total on screen effects both require interpretation of OUTPUT2, however these are in fact unique instantiations of a specific subroutine within a block. A dot notation is used to separate levels in the ACT (an IDE GUI can present the appropriate call tree fragments). Thus the effects for the example are:

e_1	e_2	$e = e_1 \wedge e_2$	$\mathrm{Ef}(f)$	$e = e_1 \vee e_2$	$\mathrm{Ef}(f)$
$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$	T	$\mathbb{I}$	T
$\mathbb{I}$	$\mathbb{P}$	$\mathbb{P}$	F(poss)	$\mathbb{I}$	T
$\mathbb{I}$	$\mathbb{D}$	$\mathbb{I}$	F(def)	$\mathbb{I}$	T
$\mathbb{D}$	$\mathbb{P}$	$\mathbb{P}$	F(def)	$\mathbb{P}$	F(poss)
$\mathbb{P}$	$\mathbb{P}$	$\mathbb{P}$	F(poss)	$\mathbb{P}$	F(poss)
$\mathbb{D}$	$\mathbb{D}$	$\mathbb{D}$	F(def)	$\mathbb{D}$	F(def)

Figure 6. Effect operators

Expenses.findExpenseTotal#1.display.OUTPUT2
AND
Expenses.findExpenseTotal#2.display.OUTPUT2
    IMPLEMENTS display_result.show_expense_items_on_screen
Expenses.findExpenseTotal#3.display.OUTPUT2
    IMPLEMENTS display_result.show_total_on_screen
Expenses.printCheque.PRINTER_T
    IMPLEMENTS print_cheque.print_in_figures
Expenses.printCheque.PRINTER_S
    IMPLEMENTS print_cheque.print_in_text
Expenses.printCheque.PRINTER_P
    IMPLEMENTS print_cheque.print_details
Expenses.produce_envelope.OUTPUT_ENVELOPE
    IMPLEMENTS address_envelope

4.5 Failure Reporting

The function model reports failure effects:

$$\mathrm{R}(f) \quad \text{if} \quad \mathrm{Fa}(f) \vee \mathrm{Un}(f)$$

In adddition we find that faults causing changes in execution path can cause spontaneous operation when the trigger and effects are in condition code.

$$\mathrm{R}(f) \quad \text{if} \quad \mathrm{Ac}^{\mathrm{Sp}}(f) \vee \mathrm{In}^{\mathrm{Sp}}(f)$$

The consequences of failure are usually reported in an FMEA:

$$\mathrm{R}(c_f) \quad \text{if} \quad \mathrm{Fa}(f)$$

For $\mathrm{Un}(f)$ the function cannot be said to been achieved since what has actually happened is merely an unexpected behaviour that can also be specified using UNEXPECTED CONSEQUENCES clauses in the IMPLEMENTS section. Significant effects are reported

$$\mathrm{R}(e) \quad \text{if} \quad (\mathrm{Tr}(f) \wedge \neg e) \vee (\neg \mathrm{Tr}(f) \wedge e)$$

For hierarchical function descriptions the state of parent function f is determined from subsidiary functions a, b where $\otimes$ indicates any boolean operator:

$$\mathrm{Tr}(f) \quad \text{if} \quad \mathrm{Tr}(a) \otimes \mathrm{Tr}(b)$$

```
class Expenses {
  double RATEA, RATEB;
  double expense_cost=0;
  double milage_cost=0;
  int employee_no;

  public void Expenses(float hotel, float
      sundries, int milage,
      String address, int emp) {
    RATEA = 0.38;
    RATEB = 0.31;
    employee_no=emp;
    milage_cost = calculateMileage(milage);
    expense_cost = findExpenseTotal(hotel,
      sundries);
    printCheque(employee_no);
    produceEnvelope(address);
  }

  public double calculateMileage(int miles){
    double ar=0;
    double br=0;
    if (miles>400){
      br=(miles-400)*RATEB;
      miles=miles-400; }
    ar=miles*RATEA;
    return (ar+br);
  }

  private double findExpenseTotal(double hotel,
      double sundries){
    double result = hotel+sundries;
    display("Hotel: ", hotel);
    display("Sundries: ", sundries);
    display("EXPENSES: ", result);
    return result;
  }

  private void printCheque(int emp_no){
    String comment="OUTPUT: ";
    double total_cost=expense_cost+milage_cost;
    String PRINTER_P = "Cheque"+emp_no;
    employee_no=emp_no;
    double PRINTER_T = total_cost;
    double PRINTER_S = formatText(total_cost);
  }

  public void produceEnvelope(String address){
    String OUTPUT_ENVELOPE=address+employee_no;
  }

  public void display(String prompt,
    double answer){
    String OUTPUT1 = prompt;
    double OUTPUT2 = answer;
  }

  String formatText(double number){
   ...
  }
}
```

Figure 7. Code for example

$$\mathrm{Ef}(f) \quad \text{if} \quad \mathrm{Ef}(a) \otimes \mathrm{Ef}(b)$$

it is necessary to account for partial functionality. For subsidiary function conjunctions or disjunctions the most specific risk *consequences* are included using the following rules allowing partial functionality to mitigate failure, although of course, the failure of the top level function is reported.

$$\mathrm{R}(c_f) \quad \text{if and only if} \quad \mathrm{Fa}(a) \wedge \mathrm{Fa}(b)$$

$$\mathrm{R}(c_a) \quad \text{if} \quad \mathrm{Fa}(a) \wedge \neg\mathrm{Fa}(b)$$

5 Generating effects

Considering faults in every variable clearly produces a large number of results even for a small fragment of code. The consistency and abstraction of the results available using the functional model. The result can initially be divided into external and internal faults. The top section of Figure 8 shows the effects of external faults. A glance at this result verifies many of our expectations. An error in the hotel or sundries inputs causes a failure in **pay expenses** which is severe but because the error will show up in the **display expenses** function failure, is detectable. A fault in the mileage input gets the highest risk because only the **print cheque** function fails and this is not as detectable. A fault in the address input, as we might expect, can only causes a failure in the envelope production resulting in a lower risk (the cheques are crossed). The variable emp represents employee number and causes a failure in both the printed cheques and envelopes. It gets a high value because this fault is not detectable in the verification display. This result might ask an engineer to question if the employee number should be included in the display function to increase the detectability of the fault.

The lower section of figure 8 contains variables that have values that are not propagated from other points in the program. We might consider these as a kind of 'hard coded external input'. Both ar and br both assigned zero, however there is no effect for ar but br can cause the **print_cheque** function to fail. Considering the code we find the initialiser br=0 contains a value that can have an impact of the expenses payment function.

6 Failure modes

The results can be automatically grouped by the permutations of function failure. We propose these as the *failure modes* of the software since they are related to the way that the fault propagates through the system resulting in characteristic groups of functions being affected. Figure 9 illustrates the failure modes that occur for the example and associates the root cause modules. The following paragraphs illustrate the steps an engineer might take to analyse these results.

The first row contains the faults that cause no failure. In this example these are unused initialisers and unused results of method calls but in general may represent redundant code or code that does not contribute to an identified function of the system.

The first real failure mode consists of a potential failure of both **display expenses total and print cheque** and can only be caused by failures in the Expenses() or FindExpenseTotal() methods. This seems reasonable and it makes sense that only variables associated with the two expense items are involved.

Block	Faulty Statement	Variable affected	Failure Mode	Sev/ Det	Potential Effects
Expenses(...){...}[#S4]	Expenses(...){...}[#S4]	**hotel[#V6] sundries[#V7]**	Pay Expenses failed; (because Display expenses total failed;Print cheque failed;)	9 5	incorrect expenses payment, Need to check expense items total manually Cheques must be witten by hand
Expenses(...){...}[#S4]	Expenses(...){...}[#S4]	**milage[#V8]**	Pay Expenses failed; (because Print cheque failed;)	9 8	Cheques must be witten by hand
Expenses(...){...}[#S4]	Expenses(...){...}[#S4]	**address[#V9]**	Envelope failed;	6 3	envelope must be addressed by hand
Expenses(...){...}[#S4]	Expenses(...){...}[#S4]	**emp[#V10]**	Pay Expenses failed; (because Print cheque failed;) Envelope failed;	9 8	Cheques must be witten by hand envelope must be addressed by hand
Expenses(...){...}[#S4]	=[#S5]	**RATEA[#V12] RATEB[#V14]**	Pay Expenses failed; (because Print cheque failed;)	9 8	Cheques must be witten by hand
calculateMileage(...){...}[#S19]	=0[#S20]	**ar[#V32]**	All functions achieved	9 8	
calculateMileage(...){...}[#S19]	=0[#S21]	**br[#V33]**	Pay Expenses failed; (because Print cheque failed;)	9 8	Cheques must be witten by hand
calculateMileage(...){...}[#S19]	(miles>400)[#S22]	**_ifcondition[#V34]**	Pay Expenses failed; (because Print cheque failed;)	9 8	Cheques must be witten by hand
calculateMileage(...){...}[#S19]	=((miles-400)*RATEB)[#S24]	**br[#V36]**	Pay Expenses failed; (because Print cheque failed;)	9 8	Cheques must be witten by hand
calculateMileage(...){...}[#S19]	=(miles-400)[#S26]	**miles[#V37]**	Pay Expenses failed; (because Print cheque failed;)	9 8	Cheques must be witten by hand

Figure 8. Summary FMEA for external inputs

S U M M A R Y		
Blocks	**Faults**	**Failure Mode**
[#S1'class Expenses{}' , #S4'Expenses(...){...}' , #S19'calculateMileage(...){...}' , #S30'findExpenseTotal(...){...}' , #S39'printCheque(...){...}' , #S53'display(...){...}']	[#V1'RATEA' , #V2'RATEB' , #V3'expense_cost' , #V4'milage_cost' , #V5'employee_no' , #V32'ar', #V59'comment' , #V74'prompt' , #V77'OUTPUT1']	**[]**
[#S4'Expenses(...){...}' , #S30'findExpenseTotal(...){...}']	[#V6'hotel' , #V7'sundries' , #V43'hotel' , #V44'sundries']	**[Display expenses total, Print cheque]**
[#S4'Expenses(...){...}' , #S19'calculateMileage(...){...}' , #S30'findExpenseTotal(...){...}' , #S39'printCheque(...){...}' , #S56'formatText(...){...}']	[#V8'milage' , #V12'RATEA' , #V14'RATEB' , #V20'milage_cost' , #V24'expense_cost' , #V30'miles' , #V33'br' , #V35'RATEB' , #V36'br' , #V37'miles' , #V38'br' , #V39'miles' , #V40'RATEA' , #V41'ar' , #V60'total_cost' , #V61'expense_cost' , #V62'milage_cost' , #V63'PRINTER_P' , #V66'PRINTER_T' , #V67'PRINTER_S' , #V79'number']	**[Print cheque]**
[#S4'Expenses(...){...}' , #S39'printCheque(...){...}' , #S50'produceEnvelope(...){...}']	[#V9'address' , #V64'employee_no' , #V70'address' , #V72'OUTPUT_ENVELOPE' , #V73'employee_no']	**[Envelope]**
[#S4'Expenses(...){...}' , #S39'printCheque(...){...}']	[#V10'emp' , #V16'employee_no' , #V57'emp_no']	**[Print cheque, Envelope]**
[#S53'display(...){...}']	[#V75'answer' , #V78'OUTPUT2']	**[Display expenses total]**

Figure 9. Example result - failure modes

The **print cheque** failure mode has most faults and also can be caused by the greatest amount of the code (not surprising since this the only major purpose of the code). If the produceEnvelope() or display() methods had appeared we would be concerned.

Failure of the **envelope** function is caused by faults in the produceEnvelope() method as is to be expected and if the Expenses() method was a cause for concern then given a suitable GUI/IDE we find that only the address variable within this high level method causes the failure mode which makes sense. The PrintCheque() method appears anomalous however. Why would a fault in the module associated with printing cheques cause a failure in Envelope production? Again the system can report that this could occur if there was a fault in the line employee_no = emp_no. Indeed should the employee_no instance variable be set in this method? Why is the parameter emp_no being used when there is a class variable containing this information? These questions indicate that something is probably wrong with the implementation/design of this method or class.

The **print cheque and envelope** failure mode might seem unlikely at first glance until we see that potential faults are related to employee number which appears on both the envelope and cheque but is not used in the display expenses function. By adding the employee number to the display expenses function we would cause the failure mode to be replaced with one that contains **print cheque and envelope and display expenses** thereby reducing the RPN for these faults.

When considering the set of failure modes produced it seems useful to consider:

- Are any of the failure modes unexpected ? Put another way are we happy that a fault could cause each set of failed/spontaneous functions. Are unexpected failure modes due to implicit assumptions?
- Do the structural components of the software that cause each failure mode seem reasonable? Given a knowledge of the design, are there any packages/modules/functions etc where it is unclear that they should be able to cause a failure mode ? An engineer could accept an effect at any level of structural blocking allowing many other faults at lower levels to be removed from consideration.

For both of these questions it possible to provide more information about how the faults propagate through the software at various levels of detail by providing call graphs, program slices and the conditions that must exist to produce the failure mode.

7 Conclusion

This paper demonstrates the concept of an FMEA analysis where software plays the major role in the system being analysed. Since many faults occur not with the high level design for which many tools already exist, but at the interface between design and implementation choices, we use an analysis that considers the behaviour of the actual implementation.

A functional model allows interpretation of faults analysed by static analysis allowing system specific failure modes to be produced. We have demonstrated the kind of results that an automated FMEA may be able to produce and how an engineer may process them, however for life-sized systems failure mode prioritisation and presentation requires further research.

The functional model constitutes the main additional cost apart from consideration of the results. The functional model is however simple and may be of benefit as a design document in its own right since it clarifies the purpose of the system, the decomposition of this purpose, and how it is to be achieved by the system. Although they have not been described in this paper, the functional model supports function sequences and timing constraints[1], and thus should be able to describe sophisticated software behaviour. The meta function descriptions available in the function model can be included in the future to allow the analysis to include fault mitigating, warning, recharging functions that interpret, for example, exception handlers and resource control. There are a number of issues that need to be resolved with respect to conditional triggers, subsystem inclusion, and fault reporting in deep function hierarchies. For real systems, library code and operating systems are used, where there is no access to potential failure modes and fault propagation. A tool would therefore require the ability to provide descriptions of external fault propagation models.

The proof of concept prototype has demonstrated the feasibility of a SFMEA analysis although it requires more development to the behavioural model including shape analysis techniques [16], and software diagnosis models [5, 4] to allow the behavioural model to propagate faults through abstract dynamic data structures, aliased variables and multiple points of execution. A usable tool would require sophisticated reporting, tracing and explanation functions supported by a GUI.

REFERENCES

[1] J. Bell and N. Snooke, 'Describing system functions that depend on intermittent and sequential behavior', in *18th International Workshop on Qualitative Reasoning (QR '04)*, pp. 51–57, (2004).

[2] J. Bell, N. Snooke, and C. J. Price, 'Functional decomposition for interpretation of model based simulation', in *19th International Workshop on Qualitative Reasoning (QR '05)*, pp. 192–198. ISBN3-9502019-0-4, (2005).

[3] 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).

[4] Rong Chen, Daniel Köb, and Franz Wotawa, 'Exploiting static abstraction of data structures for debugging', in *MONET Workshop on Model-Based Systems at ECAI 2004*, Valencia, Spain, (August 2004).

[5] Rong Chen and Franz Wotawa, 'Debugging with an enriched dependency-based model or how to distinguish between aliasing and value assignment', in *Proceedings of the International Workshop on Qualitative Reasoning (QR-2003)*, Brasilia, Brazil, (2003).

[6] T. Cichocki and J. Górski, 'Failure mode and effect analysis for safety-critical systems with software components', *SAFECOMP 2000*, **LNCS 1943**, 382–394, (2001).

[7] J. Collard, *Reasoning About Program Transformations*, Springer, 2003. ISBN 0-387-95391-4.

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

[9] IEC-60812, 'Analysis techniques for system reliability - procedure for failure mode and effects analysis (fmea)', Technical Report IEC 60812, IEC, http://www.iec.ch/, (2006).

[10] M. Lee and A. Ormsby, 'Qualitative modelling of the effects of electrical circuit faults', *Artificial Intelligence in Engineering*, **8**, 293–300, (1993).

[11] S R Luke, 'Failure mode, effects and criticality analysis (FMECA) for software', in *5th Fleet Maintenance Symposium*, pp. 731–735, (Oct 1995). Virginia Beach, VA (USA).
[12] J McDermid, 'Software safety: Where's the evidence?', in *6th Australian Workshop on Industrial Experience with Safety Critical Systems (SCS '01)*. Australian Computer Society, (2001). Available: http://www-users.cs.york.ac.uk/jam/.
[13] Society of Automotive Engineers, 'SAE ARP 5580 recommended failure modes and effects analysis (FMEA) practices for non-automobile applications', Technical Report ARP 5580, SAE, http://www.sae.org/, (2001).
[14] 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).
[15] 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).
[16] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm, 'Solving shape-analysis problems in languages with destructive updating', *ACM Transactions on Programming Languages and Systems*, **20**(1), 1–50, (January 1998).