# *An Automated Software FMEA*

Chris Price and Neal Snooke
Department of Computer Science, Aberystwyth University
cjp{nns}@aber.ac.uk

*This paper is in the Proceedings of the*
*International System Safety Regional Conference, Singapore, April 2008*

**Abstract**

The concept of software failure mode and effects analysis (FMEA) has grown in attractiveness over recent years as a way of assessing the reliability of software. Like its hardware counterpart, software FMEA is immensely tedious for an engineer to perform, as well as being error-prone. This paper presents the implementation of a novel method for automating code-level software FMEA based on treating the implemented software as a model of the desired system and propagating faults through the model to identify dependencies.

The method provides results at a level where they can be understood and acted on by software engineers. A tool implementing this method has been applied to a travel expenses payment program, and some of the automatically produced results are presented. Such automation extends significantly the range of software for which software FMEA becomes a realistic proposition. The analysis is tractable, and has been shown to provide useful results for software engineers.

One important use of this analysis is to focus further testing. The software FMEA can be used to improve automated or source code embedded testing since tests can exonerate many potential faults allowing the FMEA analysis to present an engineer with a reduced set of potential faults.

## 1. Introduction

Failure mode and effects analysis (FMEA) is a systematic safety analysis method that identifies the possible system failure modes associated with a system, and evaluates the effects on the operation of the system, should the failure mode occur. The focus of the analysis is on used the analysis to improve the safety of the system design. Typically, FMEA is practiced on physical systems, and the failure modes considered are the failures of physical components, caused by wear or other damage to the system. As software was introduced into automotive and aeronautic systems, it has been included in FMEA as a component (the ECU containing the software) with no failure modes, and the system FMEA has been produced assuming that the software works correctly.

More recently, there has been a focus on performing FMEA on the software itself. One of the unique difficulties with software systems is the complex relationship between faults and effects. A minor fault can, for example, cause a complete crash of a software system or have almost invisible but very complex, subtle, and long lasting side effects. The result is that software often has very non-uniform quality in terms of the effects of potential failures, and it is not clear when effort is available, where it should be expended to improve quality. An FMEA provides just this information allowing targeting of effort at the highest risk areas

Clearly, software components do not fail in the same manner as hardware components — a function or method does not break over time because it has become worn or damaged. Software FMEA considers all potential faults such as faulty inputs or software bugs (mutations) that could exist and ensures the worst-case consequences are known, possibly prompting actions to reduce risk. A software bug may be treated analogously to a hardware component failure, the essential difference being that hardware failures occur over time whereas software bugs exist undetected but usually only affect a very small (untested) region of the overall system behaviour.

Code-level software FMEA has been performed for some years [2, 6, 8, 12], but has been considered impractical except when applied to small pieces of highly critical code because of its cost. On the other hand, software FMEA of a more abstract specification of the system can ignore important implications of failures, especially where code is not automatically generated from the abstract specification.

This paper presents a tractable method for automating code-level software FMEA, providing results at a level where they can be understood and acted on by software engineers. A tool implementing this method has been applied to a travel expenses payment program, and some of the automatically produced results are presented. Such automation extends significantly the range of software for which software FMEA becomes a realistic proposition.

## 2. Automating FMEA

### 2.1 Relevant previous work

This work is inspired by recent success in automated electrical system FMEA [14], where automated analysis is now in routine use in industry [16]. In that work, the expected behaviour of the system is simulated from the model, and results are abstracted to the level of the function of the system. A comprehensive set of possible failures is identified from the components that compose the structure of the system. For each possible failure, the simulation and abstraction is repeated, and the results with the failure present are compared with the results when no failure is present. The difference between the two sets of results gives the system-level effect of that failure. For electrical system FMEA, the software produces automated reports such as the example shown in table 1.

| Item/Fn | Potential Failure Cause | Potential Failure Mode | Potential Failure Effect | Sev | Occ | Det |
|---|---|---|---|---|---|---|
| (1628) | The component IGN-SWITCH has failure switch stuck at start and component FR_LOCK_ACTUATOR has failure switch stuck at position b. | Regardless of any event change, the 'doors locked' function was never achieved. | Doors failed to lock. | 6 | 1 | 4 |

**Table 1: Example from generated electrical FMEA report**

Related techniques to those used in the automated FMEA in [14] have been applied to the debugging of software [3]. By treating a piece of software as a model of the desired system, it is possible to reason about the ways in which faults can be propagated through the model in order to derive the possible causes of a given set of symptoms.

### 2.2 Software FMEA automation

The research described in this paper adapts the idea of automated FMEA from the previous work in the electrical domain. There are three main aspects to this work:

*Automated model construction*. The work in [3] was able to explore single execution paths through a piece of software. In order to generate an FMEA for a failure on a piece of software, it is necessary to propagate all possible effects. This means that the techniques used in [3] are not adequate for performing software FMEA, and so techniques are developed capable of automatically constructing a fault propagation model which can generate all possible effects of a failure.

*Injection and propagation of faults*. All possible faults that could occur in the software need to be identified, so that their effect on the overall system can be explored. Given a specific fault, the fault propagation model automatically constructed from the software can be used to decide what parts of the piece of software could be affected by a specific fault, and in what way they can be affected.

*Identification of system level effects*. Generating a list of all variables affected by a fault would be far too much detail to report to an engineer. It is necessary to abstract the results of the analysis to the system level in order to report on the effect of a fault.

The next three sections will describe how each of these subgoals is achieved, in order to attain the overall goal of automatically generating a useful software FMEA report.

## 3. Automated model construction

The source code of the software to be analyzed is parsed and transformed into a fault propagation model. This is essentially a graph, where the source code statements are the edges and the variables are the nodes of the graph. It is then possible to use the graph to reason about how the effect of a fault can propagate through the program.

This is clearly language-dependent, and this paper describes an automated software FMEA system that works for a large subset of the JAVA language, and is applicable to other imperative languages. Limitations of this approach will be addressed in section 7.2, but generally correspond to the kind of coding restrictions recommended when constructing safety-critical software [10].

Fault propagation is complicated by the reuse of memory, and this problem has been overcome by using a Single Assignment Form (SAF) model [7] that transforms each memory write to allow symbolic memory locations to be logically written only once. SAF is used to generate a graph with distinct nodes for each value of a variable. This is illustrated for a simple code fragment in figure 1.

| | source code | SAF | $\mathcal{FP}$ arcs |
|---|---|---|---|
| 1 | a=1; | $a_1$=1; | $(\emptyset, a_1)$ |
| 2 | b=2; | $b_1$=2; | $(\emptyset, a_2)$ |
| 3 | a=a+b; | $a_2$=$a_1$+$b_1$; | $(a_1, a_2), (b_1, a_2)$ |
| 4 | b=b*b; | $b_2$=$b_1$*$b_1$; | $(b_1, b_2)$ |
| 5 | c=a*b; | $c_1$=$a_2$*$b_2$; | $(a_2, c_1), (b_2, c_1)$ |

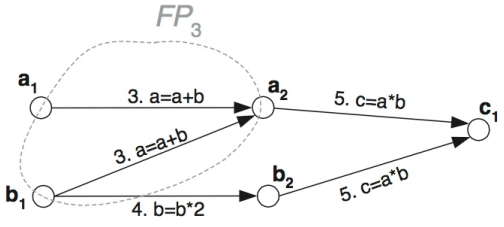


**Figure 1: Example propagation model**

Two issues arise during the process of building a fault propagation graph using SAF. Non-linear (conditional) code does not give an unique substitution for variables written in alternative execution paths. Secondly, multiple executions of a statement (e.g. in a loop) generate an indeterminate, possibly infinite, number of variables.

### 3.1 Conditions

For fault analysis we assume all faults in the reaching definition of a post conditional variable will propagate for some execution. If this is not the case, then the program must contain control structures with dead branches or complex algorithmic constraints and *if the resulting failure modes turn out to be significant*, the FMEA will highlight the issue. Either a set of faults will have no effect, or an effect will be generated where the engineer knows of application constraints that explain why the failure modes could not happen. Collard [7] uses an oracle 'function' ($\phi$) to aggregate multiple reaching definitions. For example $\phi(x_1,x_3)$ in figure 2, where $x$ following the loop depends on $x_1$ for e={12345}, and $x_3$ for e={125}. This work adopts the subscript notation $x_{\Phi 1}$ to represent the conditional statement output as a unique location. The fault propagation graph constructed indicates that, in the worst case, code in the conditional could affect the value of variables involved in the conditional statements.

| | | |
|---|---|---|
| 1 | x=1; | $x_1$=1; |
| 2 | if (j<aconst){ | $if_1$=($j_1$<aconst)<br>condition{$if_1$ |
| 3 | x=n; | $x_2$=n; |
| 4 | x=x*2; | $x_3$=$x_2$*2; |
| | } | }$x_{\Phi 1}$=$\phi$($x_1$,$x_3$) |
| 5 | x=x+1; | $x_4$=$x_{\Phi 1}$+1; |

**Figure 2: Simple condition**

### 3.2 Iteration

For fault propagation we do not wish to consider either the specific instances of a statement, or a specific execution iteration. This requires a set of variable instances, demonstrated by the need for statements 4 and 5 in figure 3, to be transformed using an additional set of variables. An additional notation is used to create a symbolic notation for each of these iteration variables.

For fault propagation, variables that are not accessible outside the loop are not interesting, as long as any effects have been propagated to memory that is accessible outside the loop. For scalar variables, this means that any variables referenced in the *first* execution of the loop are either assigned prior to the loop, or as part of a preceding step in the loop code. For subsequent executions there is the additional

possibility that a value written in a previous execution is used. Several iterations may be necessary to propagate faults to some output variables. For example, in figure 3, an initial fault in statement 1 affects *c* but not until the second iteration. $\perp$ is used to indicate a value defined prior to program execution, for example an uninitialised variable. When faults are propagated through this model, an arbitrary number of iterations will be assumed, allowing for worst case fault dependencies.

| | | |
|---|---|---|
| 1 | `a=`*expression...;* | $a_1$=0; |
| 2 | `b=2;` | $b_1$=2; |
| 3 | `for i = 0 to n {` | iterate $\{n = \{1..n_\perp\}$, $b_{2\triangleright 0} \equiv b_1$ |
| 4 | `c=b;` | $c_{1\triangleright n}$=$b_{2\triangleright(n-1)}$ |
| 5 | `b=a;` | $b_{2\triangleright n}$=$a_1$ |
| | } | } $b_{\Phi 1}$=$\phi$( $b_1$, $b_{2\triangleright N}$) |
| | | $c_{\Phi 1}$=$\phi$( $c_\perp$, $c_{1\triangleright N}$) |

**Figure 3: Example iteration**

The Φ subscripted outputs ensure that for each iteration the model only ever propagates errors from the immediately preceding execution, simplifying the propagation graph.

Non-scalar variables typically allow any element of the variable to be written or read within the loop, dependent on value-based information. Arrays or other data structures traversed by the loop variables may be in single assignment form when a different element is written for each execution of the loop, however, it is not easy to determine this property in the general case.

For FMEA, individual elements are not of interest since we need to consider a level of abstraction appropriate for any possible execution, separating structures that perform different functions and affected by different faults. For the case of arrays and the like, treatment as a homogenous item provides a reasonable approach, since a fault propagating into an array element is likely to affect each statement reading that array for some execution of the program. Clearly, this will weaken the propagation analysis in some special circumstances, e.g. if a particular location in an array is used for a special purpose. In these cases the failure mode understanding may be strengthened, however, because it seems appropriate for an engineer to document an explanation (and check relevant code operation) for significant resulting failure modes believed to be implausible. Often, such behaviours are caused by bad programming practice, or misuse of data structures, and an FMEA highlighting the fact that an alternative design would allow the design and/or implementation language to protect against failure modes is no bad thing.

Failures of the loop termination condition variable $x_\perp$ cannot increase the fault propagation output set of the loop, however it could cause an incorrect number of executions of the loop, potentially causing non-termination if the values used to produce the iteration condition expression are written in the loop. Non-termination is a failure mode that is unique to software because it has the characteristic of causing all the functions of a thread to fail. The analogy in electrical circuits is the short circuit that absorbs the available power from the system. The electrical short circuit is treated as a special case during analysis and potential non-termination within software would appear to warrant a similar special treatment in the presentation of the results.

## 4. Injection and propagation of faults

All software faults can be characterized as faulty values:

- faulty I/O
- faulty transfer functions (whether due to wrong specification or wrong implementation)
- loss of transfer function (due to crashing or hanging)
- missing outputs
- unexpected outputs

In order to perform an FMEA, it is assumed that *any* value can become faulty and the propagation of that faulty value is explored to all statements that depend on it. This is achieved by propagating the faulty value using the fault propagation model already discussed, until the effects on system outputs have been derived. The scope and path of the propagation define a failure mode.

## 5. Identification of system level effects

Functional interpretation [13] is a vital part of organising and abstracting the results of structural and behavioural analysis into the form of an FMEA report able to flag significant potential problems. Information regarding the purpose(s) of the system is required, and a functional model is used to provide this information in a structured form. The functional model described in this section identifies the purposes of the system by means of associations with the system interface.

We use a definition of function described by [1] with some additions (in italics). "An object *or system* O has a function f if it achieves an intended goal (purpose) by virtue of some external trigger T resulting in the achievement of an external *(behavioural)* effect E."

Two function states are defined. Tr(f) returns a Boolean indicating that function f is triggered and Ef(f) provides the achievement of the effect. The inoperative, failed, unexpected and achieved function states are defined respectively as follows:

$$In(f) \Leftrightarrow \neg Tr(f) \wedge \neg Ef(f)$$

$$Fa(f) \Leftrightarrow Tr(f) \wedge \neg Ef(f)$$

$$Un(f) \Leftrightarrow \neg Tr(f) \wedge Ef(f)$$

$$Ac(f) \Leftrightarrow Tr(f) \wedge Ef(f)$$

The function interpretation language allows functions to be decomposed into subsidiary functions to build a functional hierarchy.

A good deal of intuition regarding the function model can be obtained from an example functional description for a simple travel expenses payment program shown below:

```
FUNCTION pay_expenses
 ACHIEVES pay_money_owed
 BY run_program
  TRIGGERS display_expenses_total
  AND print_cheque

FUNCTION make_envelope
 ACHIEVES allow_delivery_cheque
 BY run_program
 TRIGGERS  address_envelope

FUNCTION display_expenses_total
 ACHIEVES verify_expense_items
 BY show_expense_items_on_screen
    AND
    show_total_on_screen

FUNCTION print_cheque
 ACHIEVES transfer_money
 BY print_in_figures
    AND
     print_in_text
     AND
     print_details
```

The purpose (*ACHIEVES*), trigger (i.e. *run_program*) and effect (e.g. *address_envelope*) symbols all refer to elements that will be described in the following sections.

### 5.1 Linking functions to behaviour

The functional model must be linked to the behaviour of the program via the triggers and effects. These will be system inputs and outputs and could be memory locations, or ports that influence external devices or system calls at higher levels.

The triggers and effects from the function definitions need to be mapped onto variables in the program implementing the functions, either global variables or instantiated variables.

The code in figure 4 represents a much simplified program class implementing the expenses payment problem. To avoid having to consider external library functions for I/O in this compressed example, we treat assignment to some specific variables as the output of the program. The discussion and example below shows how the functions shown previously are mapped to this code.

```
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){
    ...
    OUTPUT2 = answer;  }
```

**Figure 4: Code for example**

In this example, both the *show_expenses_items_on_screen* and the *show_total_on_screen* effects require interpretation of *OUTPUT2* (as each of them invokes display), however these are different instantiations of the same subroutine. The dot notation is used to traverse the abstract call stack (ACS) resulting in the following definition of effects for the expenses example:

```
Expenses.findExpenseTotal#1.display.OUTPUT2
AND
Expenses.findExpenseTotal#2.display.OUTPUT2
  IMPLEMENTS
  display_expenses_total.show_expense_items_on_screen

Expenses.findExpenseTotal#3.display.OUTPUT2
 IMPLEMENTS
 display_expenses_total.show_total_on_screen

Expenses.printCheque.PRINTER_T
 IMPLEMENTS   print_cheque.print_in_figures   ...etc
```

### 5.2 Faults and program execution

When faults are propagated through the model of the program, they can affect the expected achievement of functions in several ways: They can lead to functions not being achieved when they should have been, and also to functions spontaneously being achieved when they should not have been. Each of these occurrences is reported in the FMEA results.

For hierarchical function descriptions, the state of a parent function is determined from its subsidiary functions. It is necessary to account for partial functionality. For subsidiary function conjunctions or disjunctions the most specific risk failure consequences are included, allowing partial functionality to mitigate failure, although of course, the failure of the top level function is reported

### 5.3 Function failure risk

Generally for FMEA, severity, detectability and occurrence values are calculated (as was illustrated in table 1 for the electrical FMEA example). The absolute values of the numbers have little significance, but they allow generation of a ranked summary, providing prioritization of perceived failure mode risk. Software metrics such as code complexity have been shown to be correlated to defect rate, and have been suggested [9] as a way to provide occurrence (fault likelihood) values. Severity and detection values can

be associated with system functions, and linked to the description of purpose. These values are decided for each purpose, in the range 1-10 as documented in the FMEA literature [15], with larger values indicating greater significance.

For the expenses example, we enhance the function model as follows:

```
PURPOSE pay_money_owed  DESCRIPTION "reimburse expenses to employee"
  FAILURE CONSEQUENCE  "incorrect expenses payment"
  SEVERITY 9  DETECTABILITY 5

PURPOSE verify_expense_items
  FAILURE CONSEQUENCE  "Need to check expense items and total manually"
  SEVERITY 5  DETECTABILITY 6

PURPOSE transfer_money
  FAILURE CONSEQUENCE  "Cheques must be written by hand"
  SEVERITY 8  DETECTABILITY 9

PURPOSE allow_delivery_of_cheque
  FAILURE CONSEQUENCE  "envelope must be addressed by hand"
  SEVERITY 6  DETECTABILITY 3
```

## 6. Results for Expenses Program

The software FMEA tool has been applied to the travel expenses program linked with the functional description given in the previous section. The software FMEA tool generated a fault propagation model of the type described in section 3 from the Java code of the expenses program. It then calculated the result of propagating the effect of a fault in each input and variable in the system, and interpreted the results using the functional description, and condensed the results into a failure mode summary.

Figure 5 shows an example output from the fault propagation of external faults. It shows that an error in the *hotel* or *sundries* inputs causes a failure to the *pay_expenses* function which is severe but because the error will show up in the *display_expenses* function failure, is detectable.

Notice that the FMEA does not guarantee that a fault will result in all (or even any) of the function failings provided in the failure mode - it provides a worst case scenario. A system may only partially exhibit a failure mode mechanism during an actual failure scenario dependent on the exact nature of the fault.

| 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 and total manually Cheques must be witten by hand |

**Figure 5: Summary FMEA for external inputs**

The FMEA results focus attention on the most significant failure modes, and how faults may potentially cause failure of the system. The task for a system reliability engineer is to use their understanding of the system and its design from the FMEA to ensure that the failure modes are acceptable. This may be done in several ways by checking:

- that faults have the correct level of detectability in relation to their severity
- that the correct fault mitigating functions are in place (e.g. exception handlers that mitigate the effects of low level subsidiary function failure)
- that unexpected failure modes do not exist, for example, because implementation ‘breaks’ the design concepts leading to unexpected propagation of failure effects.

### 6.1 Failure modes

Propagating the potential faults for every variable produces a large number (of the same order as lines of code, since statements tend to assign or reassign a variable) of individual results like the one shown in figure 5. This is far too much detail in practical cases for engineers to look at every result, but the

consistency of function identification allows these results to be automatically grouped by the permutations of function failure produced.

Figure 6 illustrates all the failure modes that occur for the example, showing the modules that are able to produce each of those failure modes. The centre column provides detail that need not be shown in a summary report, and production software would allow the detailed fault causes to be tracked and viewed where necessary. The following paragraphs illustrate how an engineer might analyse these results.

| **S U M M A R Y** | | |
|---|---|---|
| **Blocks** | **Faults** | **Failure Mode** |
| [#S1'**class Expenses**{}' ,<br>#S4'**Expenses**(...){...}' ,<br>#S19'**calculateMileage** (...){...}' ,<br>#S30'**findExpenseTotal** (...){...}' ,<br>#S39'**printCheque** (...){...}' ,<br>#S53'**display**(...){...}' ] | [#V1'RATEA' , #V2'RATEB' , #V3'expense_cost' ,<br>#V4'milage_cost' , #V5'employee_no' , #V32'ar',<br>#V59'comment' , #V74'prompt' , #V77'OUTPUT1' ] | **[]** |
| [#S4'**Expenses**(...){...}' ,<br>#S30'**findExpenseTotal** (...){...}' ] | [#V6,#V43'hotel' , #V7'sundries' , #V44'sundries' ] | **[Display expenses total,<br>Print cheque]** |
| [#S4'**Expenses**(...){...}' ,<br>#S19'**calculateMileage** (...){...}' ,<br>#S30'**findExpenseTotal** (...){...}' ,<br>#S39'**printCheque** (...){...}' ,<br>#S56'**formatText** (...){...}' ] | [#V8'milage' , #V12'#V40'RATEA' ,<br>#V14,#35'RATEB' , #V20,#V62'milage_cost' ,<br>#V24'expense_cost' , #V30,#V37,#V39'miles' ,<br>#V33,#V36,#V38'br' , #V41'ar' , #V60'total_cost' ,<br>#V61'expense_cost' , #V63'PRINTER_P' ,<br>#V66'PRINTER_T' , #V79'PRINTER_S'] | **[Print cheque]** |
| [#S4'**Expenses**(...){...}' ,<br>#S39'**printCheque** (...){...}' ,<br>#S50'**produceEnvelope** (...){...}' ] | [#V9,#V70'address' , #V64,#V73'employee_no' ,<br>#V72'OUTPUT_ENVELOPE' ] | **[Envelope]** |
| [#S4'**Expenses**(...){...}' ,<br>#S39'**printCheque** (...){...}' ] | [#V10'emp' , #V16'employee_no' , #V57'emp_no' ] | **[Print cheque, Envelope]** |
| [#S53'**display**(...){...}' ] | [#V75'answer' , #V78'OUTPUT2' ] | **[Display expenses total]** |

**Figure 6: Example result - failure modes**

The first row contains the faults that cause no failure. In this example, this is due to unused initialisers and unused results of method calls, but in general they could represent redundant code or code that does not contribute to an identified (subsidiary) function of the system.

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

The *print_cheque* failure mode has the greatest number of potential faults, and also can be caused by the greatest amount of the code (unsurprising since this the only major purpose of the code), although the methods listed do contribute in some way to the production of cheques, and at least the *produceEnvelope*() and *display*() methods do not cause faults in the cheque printing.

Failure of the *envelope* function can be 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, the tool will show 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 block associated with printing cheques cause a failure in Envelope production? The tool can report that this could occur for 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? This failure mode and associated questions indicate that something is probably wrong with the implementation/design of this method or class. It is worth noting that testing could never have found this issue. It is a latent problem just waiting for another use of the method that passes in a different value or seemingly simple edit of the method, that results in a value the designer did not expect.

The *print_cheque* and *Envelope* failed failure mode might seem unlikely at first glance until we see that all potential faults are related to employee number, which appears on both the envelope and cheque. Note that it was not intended to be on the envelope, and thus was not identified as an output of the envelope function, so some code would be identified as having no fault consequences, another cause for concern. 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 risk for these faults, by increasing detectability.

For identified concerns, more detail about how the failure mode occurs at various levels could be provided by call graphs, program slices and the conditions that must exist to produce the failure mode. At this point an engineer may find:

- The implementation has failure characteristics (flaws) that are not expected from the design.
- Identified failure modes do not actually exist because of implicit assumptions about the way the software will be used, eg. specific value combinations preclude certain execution paths or event sequences do not exist in practice. These should all be documented (ideally in an automatically checkable form) to ensure that in future versions the assumptions still hold.

## 7. Conclusions

### 7.1 Achievements

The automatically generated software FMEA report described in this paper has the following characteristics:

- It provides an evidence-based analysis that deals with a wide range of possible behaviors at a low level of precision.
- It considers hypothetical faults and determines the possible effects assigning significance to them. Any statement, module, or subsystem is assumed to have the potential to fail or produce incorrect output, and it assesses the potential effects.
- The results can be used to identify design or implementation issues that might lead to faults with disproportionate or unexpected consequences, identifying areas of design or implementation that allow significant failures to occur. Engineering effort can then be expended to address the most significant faults.

In the example given, the relationship between size of the functional description and the size of the program code is not representative. A typical system will have much more code, but would not have a proportionally larger functional description, making construction of the functional description a reasonable overhead.

### 7.2 Limitations

The work described here does not cover all constructs in all languages. It would be impossible to apply these techniques to assembler programming. However, modern software languages both encourage and enforce higher 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. Object-oriented methods provide common structures to partition data with code. These techniques make software FMEA analysis feasible in that they constrain potential fault impacts.

The main limitations of this approach at present come from the need to trace dependencies between variables. This is impossible with several programming constructs such as pointer arithmetic, generation of dynamic structures, keeping track of variables in dynamic heap based structures, and recursion. It is reassuring to note that the types of programming construct that cannot be addressed with this approach are similar to the types of construct that the MISRA C guidelines for safety critical embedded software recommend should be avoided [10]. Embedded software is the obvious area for early application of this software FMEA tool, because of the safety critical demands and the limited, well defined external functionality.

In order to apply these techniques to general software, the ability to handle dynamic data structures is vital. For more complex dynamic data structures where a finer fault propagation granularity is required to

maintain the strength of the analysis, it is necessary to logically segment the heap into fault propagation regions. Shape analysis and related research appears to provide some possible approaches [4,5].

### 7.3 Potential of this approach

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 endeavor), they do not guarantee the quality of a product.

Testing is often the only evidence based analysis performed on software, 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.

As observed by McDermid [11] we should look for evidence of software safety. To perform such a task requires identification of the safety and functional requirements, followed by an analysis of the possible failure modes of the product, mapped to the effects on the safety requirements. Code-level software FMEA performs the required analysis, but is intensive of effort and impractical for all but the smallest, most important pieces of code. Higher level software FMEA can conceal much of the detail that actually affects system safety. The automated software FMEA presented here is practical for software composed of at least thousands of lines of code. The results could be combined into system level software FMEA at a more abstract level. The paper has presented different ways of reporting the results to engineers, and it is expected that the most appropriate ways of presenting results will evolve as engineers try to use the results.

One important use of this type of analysis is to focus further testing. The software FMEA can be used to improve automated or source code embedded testing since tests can exonerate many potential faults allowing the FMEA analysis to present an engineer with a reduced set of potential faults. Thus an iterative cycle of software FMEA and test generation targeted to the specific structure and behaviour of the implementation will ensure the maximum number of potential faults have been exonerated and possible unexpected effects of the remainder are known.

### References


[1] J. Bell, N. Snooke, and C. J. Price. Functional decomposition for interpretation of model based simulation. 19th International Workshop on Qualitative Reasoning (QR'05). 192–198, Graz, Austria, 2005.

[2] J. B. Bowles, Failure modes and effects analysis for a small embedded control system. Annual Reliability and Maintainability Symposium, 1-6, 2001.

[3] R. Chen and F. Wotawa. Debugging with an enriched dependency-based mode. Proceedings of the International Workshop on Qualitative Reasoning (QR-2003), Brasilia, Brazil, 2003.

[4] R. Chen, D. Kob, and F. Wotawa, Exploiting static abstraction of data structures for debugging. MONET Workshop on Model-Based Systems, ECAI 2004. Valencia, Spain, August 2004.

[5] J. C. Corbett, Using shape analysis to reduce finite-state models of concurrent Java programs. ACM Transactions on Software Engineering and Methodology. 9(1), 51–93, 2000.

[6] T. Cichocki and J. Gorski. Failure mode and effect analysis for safety-critical systems with software components. SAFECOMP 2000, vol. LNCS1943, 382-394, 2001.

[7] J. Collard, Reasoning About Program Transformations. Springer, 2003.

[8] P.L. Goddard. Software FMEA techniques. Annual Reliability and Maintainability Symposium, 118-123, 2000.

[9] S. R. Luke. Failure mode, effects and criticality analysis (FMECA) for software. 5th Fleet Maintenance Symposium, 731–735, Virginia Beach, VA (USA), Oct 1995.

[10] ISO/TR 15497: 2000. Road vehicles – Development Guidelines for Vehicle Based Software. 2000.

[11] J. McDermid. Software safety: Where's the evidence? 6th Australian Workshop on Industrial Experience with Safety Critical Systems (SCS '01). Australian Computer Society, 2001, http: //www-users.cs.york.ac.uk/ jam/.

[12] N. Ozarin and M. Siracusa. A process for failure modes and effects analysis of computer software. Annual Reliability and Maintainability Symposium, 2002.

[13] C. J. Price. Function Directed Electrical Design Analysis, Artificial Intelligence in Engineering, 12(4), 445–456, 1998.

[14] C. J. Price, N. A. Snooke, S. D. Lewis. A layered approach to automated electrical safety analysis in automotive environments. Computers in Industry, 57: 451–461, 2006.

[15] Society of Automotive Engineers SAE ARP 5580: recommended failure modes and effects analysis (FMEA) practices for non-automobile applications. SAE, http: //www.sae.org/, Tech. Rep. ARP 5580, 2001.

[16] P. Struss and C. J. Price. Model-based systems in the automotive industry, AI Magazine, special issue on Qualitative Reasoning, 24(4), 17–34, Winter 2003.