Applying Failure Mode Modular De-Composition (FMMD) across the Software/Hardware Interface

R.Clark*, A. Fish† , C. Garrett†, J. Howse†
*Energy Technology Control, UK. r.clark@energytechnologycontrol.com
†University of Brighton, UK

Keywords: static failure mode modelling; safety-critical; software fmea

Abstract

This paper presents a modular variant of Failure Mode Effects Analysis (FMEA), Failure Mode Modular De-Composition (FMMD), a methodology which can be applied to software, and is compatible and integrable with FMMD performed on mechanical and electronic systems. Software generally sits on top of most modern safety critical control systems and defines its most important system wide behaviour and communications. Currently standards that demand FMEA for hardware (e.g. EN298, EN61508), do not specify it for software, but instead specify good practise, review processes and language feature constraints. This is a weakness. Where FMEA traces component failure modes to resultant system failures, software has been left in a non-analytical limbo of best practises and constraints. If software and hardware integrated FMEA were possible, electro-mechanical-software hybrids could be modelled, and so we could consider 'complete' failure mode models. Presently FMEA, stops at the glass ceiling of the computer program: FMMD seeks to address this, and offers additional test efficiency benefits.

1 Introduction

This paper describes a modular FMEA process that can be applied to software. This modular variant of FMEA is called Failure Mode Modular de-composition (FMMD). Because this process is based on failure modes of components, it can be applied to electrical and/or mechanical systems. The hierarchical structure of software is then examined, and definitions from contract programming are used to define failure modes and failure symptoms for software functions. With these definitions we can apply the FMMD modular form of FMEA to existing software[1].

[1]Existing software excluding recursive [10][16.2] code, and unstructured non-functional language.

2 FMEA Background

Failure Mode Effects Analysis is the process of taking component failure modes, tracing their effects through a system and determining what system level failure modes could be caused. FMEA dates from the 1940s where simple electro-mechanical systems were the norm. Modern control systems nearly always have a significant software/firmware element, and not being able to model software with current FMEA methodologies is a cause for criticism [9][Ch.12]. Difficulties in integrating mechanical and electronic/software failure models are discussed in [1].

Current work on Software FMEA SFMEA usually does not seek to integrate hardware and software models, but to perform FMEA on the software in isolation [14]. Work has been performed using databases to track the relationships between variables and system failure modes [7], to introduce automation into the FMEA process [21] and to provide code analysis automation [18]. Although the SFMEA and hardware FMEAs are performed separately, some schools of thought aim for Fault Tree Analysis (FTA) [12, 17] (top down - deductive) and FMEA (bottom-up inductive) to be performed on the same system to provide insight into the software hardware/interface [6]. Although this would give a better picture of the failure mode behaviour, it is by no means a rigorous approach to tracing errors that may occur in hardware through to the top (and therefore ultimately controlling) layer of software.

2.1 Current FMEA techniques are not suitable for software

The main FMEA methodologies are all based on the concept of taking base component failure modes, and translating them into system level events/failures [3, 13]. In a complicated system, mapping a component failure mode to a system level failure will mean a long reasoning distance; that is to say the actions of the failed component will have to be traced through several sub-systems, gaug-

ing its effects with other components. With software at the higher levels of these sub-systems, we have yet another layer of complication. In order to integrate software, we need to re-think the FMEA concept of simply mapping a base component failure to a system level event. One strategy would be to modularise FMEA; to break down the failure effect reasoning into small modules from the bottom-up. If we pre-analyse modules, and they can be combined with others, into larger sub-systems, we eventually form a hierarchy of failure mode behaviour for the entire system. With higher level modules, we can reach the level in which the software resides. For instance, to read a voltage into software via an Analogue to Digital Converter (ADC) we rely on an electronic sub-system that conditions the input signal and then routes it through a multiplexer (MUX) to the ADC. We could easily consider this conditioning and MUX circuit a 'module'; with its failure mode model defined, modelling the software to hardware interface becomes possible. This failure mode model, would give us the ways in which the signal conditioning and multiplexer could fail. We can use this to work out how our software could fail, and with this develop a modular FMEA model of the software.

3 Modularising FMEA

In outline, in order to modularise FMEA, we must create small modules from the bottom-up. We can do this by taking collections of base components that perform (ideally) a simple and well defined task called functional groupings. We can then analyse the failure mode behaviour of a functional grouping using all the failure modes of all its components. When we have its failure mode behaviour, called the symptoms of failure from the perspective of the functional grouping, we treat the functional grouping as a derived component, where the failure modes of the derived component are the symptoms of failure of the functional grouping. We can then use derived components to build higher level functional groupings until we have a complete hierarchical model of the failure mode behaviour of a system. An example of this process, applied to an inverting op-amp configuration is given in [4].

FMMD, the process. The main aim of FMMD is to build a hierarchy of failure behaviour from the base component level up to the top, or system level, with analysis stages between each transition to a higher level in the hierarchy. The first stage is to choose base components that interact and naturally form functional groupings. The initial functional groupings are collections of base components. From the point of view of fault analysis, we are not interested in the components themselves, but in the ways in which they can fail. A functional grouping is a collection of components that perform some simple task or function. In order to determine how a functional grouping can fail, we need to consider all the failure modes of its components. By analysing the fault behaviour of a 'functional grouping' with respect to all its components failure modes, we can determine its symptoms of failure. In other words we have taken a functional grouping, and analysed how it can fail according to the failure modes of its components, and then determine the functional grouping failure symptoms. We then create a new derived component which has as its failure modes, the failure symptoms of the functional grouping from which it was derived.
We use the symbol 'D' to represent the creation of a derived component from a functional grouping. This symbol is convenient for drawn hierarchy diagrams. We define the D function, where $\mathcal{G}$ is the set of all functional groupings and $\mathcal{DC}$ is the set of all derived components, $D(\mathcal{G}) \rightarrow \mathcal{DC}$. We show an FMMD hierarchy in figure 1. There are three functional groups comprised of base components. These are analysed individually using FMEA. That is to say their component failure modes are examined, and thus the ways in which the functional groupings can fail. The ways in which a functional grouping can fail, can be viewed as symptoms of failure for the functional grouping. The 'D' function is now applied to create derived components. These are shown in figure 1 above the functional groupings. Now that we have derived components, we can use them to form a higher level functional grouping. We apply the same FMEA process to this and can derive a top level derived component (which has the system—or top—level failure modes).

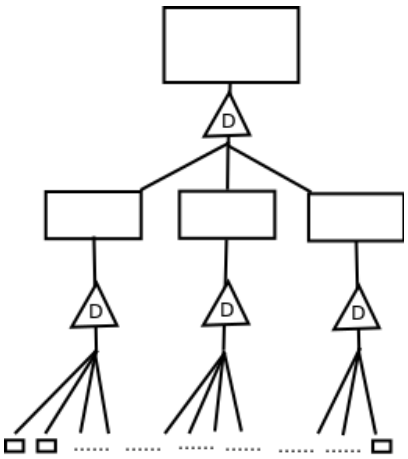

Figure 1: FMMD Hierarchy

Note the diagram of the FMMD hierarchy is very similar to a simple non-recursive programmatic function call tree.

4 FMEA applied to Software

With modular FMEA i.e. FMMD we have the concepts of failure modes of components, functional groupings and symptoms of failure for a functional group. A programmatic function has similarities with a functional grouping as defined by the FMMD process. An FMMD functional grouping is placed into a hierarchy. A software function is placed into a hierarchy, that of its call-tree. A software function typically calls other functions and uses data sources via hardware interaction, which could be viewed as its components. It has outputs, i.e. it can perform actions on data or hardware which will be used by functions that may call upon it. We can map a software function to a functional grouping in FMMD. Its failure modes are the failure modes of the software components (other functions it calls) and the hardware from which it reads values. Its outputs are the data it changes, or the hardware actions it performs.
When we have analysed a software function—treating failure conditions of its inputs as failure modes—we can determine its symptoms of failure. We can thus apply the D function to software functions, by viewing them in terms of their failure mode behaviour. To simplify things, software already fits into a hierarchy. For electronic and mechanical systems, although we may be guided by the original designers concepts of modularity in design, applying FMMD means deciding on the members for functional groupings and the subsequent hierarchy. With software already written, that hierarchy is fixed/given.

4.1 Software, a natural hierarchy

Software written for safety critical systems is usually constrained to be modular [19][vol.3] and non recursive [10][15.2]. Because of this we can assume a direct call tree. Functions call functions from the top down and eventually call the lowest level library or IO functions that interact with hardware/electronics.
What is potentially difficult with a software function is deciding what are its failure modes, and later what are its failure symptoms. With electronic components, we can use literature to point us to suitable sets of failure modes [2, 5, 20].With software, only some library functions are well known and rigorously documented enough to have the equivalent of known failure modes. Most software is 'bespoke'. We need a different strategy to describe the failure mode behaviour of software functions. We can use definitions from contract programming to assist here.

4.2 Contract programming description

Contract programming is a discipline [11] for building software functions in a controlled and traceable way. Each function is subject to pre-conditions (constraints on its inputs), post-conditions (constraints on its outputs) and function wide invariants (rules). A precondition, or requirement for a contract software function defines the correct ranges of input conditions for the function to operate successfully. For a software function, a violation of a pre-condition is in effect a failure mode of 'one of its components'. A post condition is a definition of correct behaviour by a function. A violated post condition is a symptom of failure of a function. Post conditions could be either actions performed (i.e. the state of hardware changed) or an output value of a function. Invariants in contract programming may apply to inputs to the function (where they can be considered failure modes in FMMD terminology), and to outputs (where they can be considered failure symptoms in FMMD terminology).

4.3 Software FMMD

For the purpose of example, we chose a simple common safety critical industrial circuit that is nearly always used in conjunction with a programmatic element. A common method for delivering a quantitative value in analogue electronics is to supply a current signal to represent the value to be sent [16][p.934]. Usually, $4mA$ represents a zero or starting value and $20mA$ represents the full scale, and this is referred to as $4\rightarrow20mA$ signalling. $4\rightarrow20mA$ has an electrical advantage as well because the current in a loop is constant [16][p.20]. Thus resistance in the wires between the source and the receiving end is not an issue that can alter the accuracy of the signal. This circuit has many advantages for safety. If the signal becomes disconnected it reads an out of range $0mA$ at the receiving end. This is outside the $4\rightarrow20mA$ range, and is therefore easy to detect as an error rather than an incorrect value. Should the driving electronics go wrong at the source end, it will usually supply far too little or far too much current, making an error condition easy to detect. At the receiving end, one needs a resistor to convert the current signal into a voltage that we can read with an ADC.

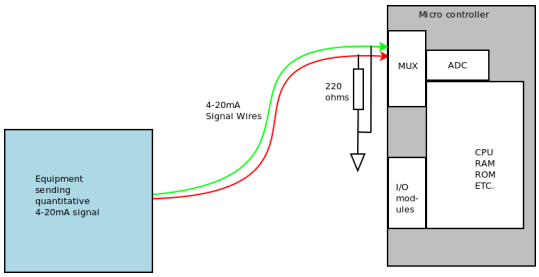

Figure 2: Context Diagram for $4\rightarrow20mA$ loop

The diagram in figure 2 shows some equipment which is sending a $4\rightarrow20mA$ signal to a micro-controller system. The signal is locally driven over a load resistor, and then

read into the micro-controller via an ADC and its multiplexer. With the voltage detected at the ADC the multiplexer can read the intended quantitative value from the external equipment.

4.4 Simple Software Example

Consider a software function that reads a 4→20mA input, and returns a value between 0 and 999 (i.e. per mil $^0/_{00}$) representing the current detected with an additional error indication flag. Let us assume the 4→20mA detection is via a 220Ω resistor, and that we read a voltage from an ADC into the software. Let us define any value outside the 4mA to 20mA range as an error condition. As a voltage, we use ohms law [16] to determine the voltage ranges: $V = IR$, $0.004A{*}220\Omega = 0.88V$ and $0.020A{*}220\Omega = 4.4V$. Our acceptable voltage range is therefore $(V \geq 0.88) \wedge (V \leq 4.4)$.
This voltage range forms our input requirement. We can now examine a software function that performs a conversion from the voltage read to a per mil representation of the 4→20mA input current. For the purpose of example the 'C' programming language [8] is used[2]. We initially assume a function **read_ADC** which returns a floating point value representing the voltage read (see code sample in figure 3).

```
/***********************************************/
/* read_4_20_input()                           */
/***********************************************/
/* Software function to read 4mA to 20mA input */
/* returns a value from 0-999 proportional     */
/* to the current input.                       */
/***********************************************/
int  read_4_20_input ( int * value ) {
  double input_volts;
  int error_flag;
  /* require: input from ADC to be
              between 0.88 and 4.4 volts */
  input_volts = read_ADC(INPUT_4_20_mA);

  if ( input_volts < 0.88 || input_volts > 4.4 ) {
    error_flag = 1; /* Error flag set to TRUE */
  }
  else {
    *value = (input_volts - 0.88) * ( 4.4 - 0.88 ) * 999.0;
    error_flag = 0; /* indicate current input in range */
  }
  /* ensure: value is proportional (0-999) to the
             4 to 20mA input                      */
  return error_flag;
}
```

Figure 3: Software Function: **read_4_20_input**

We now look at the function called by **read_4_20_input**, **read_ADC**, which returns a voltage for a given ADC channel. This function deals directly with the hardware in the micro-controller on which we are running the software. Its job is to select the correct channel (ADC multiplexer) and then to initiate a conversion by setting an ADC 'go' bit (see code sample in figure 4). It takes the raw ADC reading and converts it into a floating point[3] voltage value.

[2]C coding examples use the Misra [10] and SIL-3 recommended language constraints [19].

[3]the type 'double' or 'double precision' is a standard C language floating point type [8].

```
/***********************************************/
/* read_ADC()                                  */
/***********************************************/
/* Software function to read voltage from a    */
/* specified ADC MUX channel                   */
/* Assume 10 ADC MUX channels 0..9             */
/* ADC_CHAN_RANGE = 9                          */
/* Assume ADC is 12 bit and ADCRANGE = 4096    */
/* returns voltage read as double precision    */
/***********************************************/
double  read_ADC( int  channel ) {
  int timeout = 0;
  /* require: a) input channel from ADC to be
              in valid ADC range
              b) voltage ref is 0.1% of 5V     */

  /* return out of range result  */
  /* if invalid channel selected */
  if ( channel > ADC_CHAN_RANGE )
     return -2.0;
  /* set the multiplexer to the desired channel */
  ADCMUX = channel;
  ADCGO = 1; /* initiate ADC conversion hardware */
  /* wait for ADC conversion with timeout */
  while ( ADCGO == 1 || timeout < 100 )
     timeout++;
  if ( timeout < 100 )
       dval = (double) ADCOUT * 5.0 / ADCRANGE;
  else
       dval = -1.0; /* indicate invalid reading */
  /* return voltage as a floating point value */
  /* ensure: value is voltage input to within 0.1% */
  return dval;
}
```

Figure 4: Software Function: **read_ADC**

We now have a very simple software structure, a call tree, where *read_4_20_input* calls *read_ADC*, which in turn interacts with the hardware/electronics. This software is above the hardware in the conceptual call tree—from a programmatic perspective—software is reading values from the 'lower level' electronics. FMEA is always a bottom-up process and so we must begin with this hardware. The hardware is simply a load resistor, connected across an ADC input pin on the micro-controller and ground. We can identify the resistor and the ADC module of the micro-controller as the base components in this design. We now apply FMMD starting with the hardware.

4.5 FMMD Process — 4→20mA programmatic value implementation.

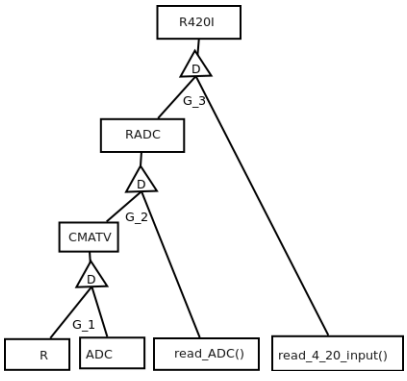

Figure 5: FMMD Hierarchy for 4→20mA input

We analyse the 4→20mA to programmatic value implementation in three stages using FMMD. We start with

the hardware, and then add each of the software functions. This forms the fault mode hierarchy represented in figure 5.

Functional Grouping - Convert mA to Voltage - (CMATV) This functional grouping contains the load resistor and the physical Analogue to Digital Converter (ADC). Our functional grouping, G_1 is thus the set of base components: $G_1 = \{R, ADC\}$. We now determine the failure modes of all the components in G_1. For the resistor we can use a failure mode set from the literature [20]. Where the function fm returns a set of failure modes for a given component we can state: $fm(R) = \{OPEN, SHORT\}$. For the ADC we can determine the following failure modes:

- STUCKAT — The ADC outputs a constant value,
- MUXFAIL — The ADC cannot select its input channel correctly,
- LOW — The ADC output is always LOW, or zero ADC counts,
- HIGH — The ADC output is always HIGH, or max ADC counts.

We can use the function fm to define the failure modes of an ADC thus: $fm(ADC) = \{STUCKAT, MUXFAIL, LOW, HIGH\}$. With these failure modes, we can analyse our first functional group, see table 1.

Table 1: G_1: Failure Mode Effects Analysis

Failure Scenario	**failure effect**	**Symptom ADC**
1: R_{OPEN}	resistor open, voltage on pin high	$HIGH$
2: R_{SHORT}	resistor shorted, voltage on pin low	LOW
3: $ADC_{STUCKAT}$	ADC reads out fixed value	V_ERR
4: $ADC_{MUXFAIL}$	ADC may read wrong channel	V_ERR
5: ADC_{LOW}	output low	LOW
6: ADC_{HIGH}	output high	$HIGH$

We now collect the symptoms for the hardware functional group, $\{HIGH, LOW, V\_ERR\}$. We now create a derived component to represent this called $CMATV$. We can express this using the 'D' function thus: $CMATV = D(G_1)$. As its failure modes are the symptoms of failure from the functional group we can now state: $fm(CMATV) = \{HIGH, LOW, V\_ERR\}$.

Functional Group - Software - Read_ADC - (RADC) The software function *Read_ADC* uses the ADC hardware analysed as the derived component CMATV above. The code fragment in figure 4 states preconditions, as */* require: a) input channel from ADC to be in valid ADC range b) voltage ref is 0.1% of 5V */*. From the above contractual programming requirements, we see that the function must be sent the correct channel number. A violation of this can be considered a failure mode of the function, which we can call $CHAN\_NO$. The reference voltage for the ADC has a 0.1% accuracy requirement. If the reference value is outside of this, it is also a failure mode of this function, which we can call V_REF. Taken as a component for use in FMEA/FMMD our function has two failure modes. We can therefore treat it as a generic component, *Read_ADC*, by stating: $fm(Read\_ADC) = \{CHAN\_NO, VREF\}$
As we have a failure mode model for our function, we can now use it in conjunction with the ADC hardware derived component CMATV, to form a functional grouping G_2, where $G_2 = \{CMSTV, Read\_ADC\}$. We now analyse this hardware/software combined functional grouping.

Table 2: G_2: Failure Mode Effects Analysis

Failure Scenario	**failure effect**	**Symptom RADC**
1: $CHAN\_NO$	wrong voltage read	VV_ERR
2: $VREF$	ADC volt-ref incorrect	VV_ERR
3: $CMATV_{V\_ERR}$	voltage value incorrect	VV_ERR
4: $CMATV_{HIGH}$	ADC may read wrong channel	$HIGH$
5: $CMATV_{LOW}$	output low	LOW

We now collect the symptoms of failure for the functional grouping analysed (see table 2) as $\{VV\_ERR, HIGH, LOW\}$. We can add as well the violation of the postcondition for the function. This postcondition, */* ensure: value is voltage input to within 0.1% */* , corresponds to VV_ERR, and is already in the failure mode set for this functional grouping. We can now create a derived component called $RADC$ thus: $RADC = D(G_2)$ which has the following failure modes: $fm(RADC) = \{VV\_ERR, HIGH, LOW\}$.

Functional Group - Software - voltage to per mil - VTPM This function sits on top of the $RADC$ derived component determined above. We look at the preconditions for the function *read_4_20_input* to determine its failure modes. Its pre-condition is */* require: input from ADC to be between 0.88 and 4.4 volts */*. We can map this violation of the pre-condition, to the fail-

ure mode VRNGE; we can state $fm(read\_4\_20\_input) = \{VRNGE\}$. We can now form a functional group with the derived component $RADC$ and the software component $read\_4\_20\_input$, i.e. $G_3 = \{read\_4\_20\_input, RADC\}$.

Table 3: G_3: Read_4_20: Failure Mode Effects Analysis

Failure Scenario	**failure effect**	**Symptom RADC**
1: RI_{VRGE}	voltage outside range	OUT_OF_RANGE
2: $RADC_{VV_ERR}$	voltage incorrect	VAL_ERR
3: $RADC_{HIGH}$	voltage value incorrect	VAL_ERR
4: $RADC_{LOW}$	ADC may read wrong channel	OUT_OF_RANGE

The failure symptoms for the functional grouping are $\{OUT\_OF\_RANGE, VAL\_ERR\}$. The postcondition for the function $read\_4\_20\_input$, */* ensure: value is proportional (0-999) to the 4 to 20mA input */* corresponds to the VAL_ERR and is already in the set of failure modes. For single failures these are the two ways in which this function can fail. An OUT_OF_RANGE will be flagged by the error flag variable. The VAL_ERR will mean that the value read is simply wrong. We can finally make a derived component to represent a failure mode model for our function $read\_4\_20\_input$ thus: $R420I = D(G_3)$. This new derived component has the following failure modes: $fm(R420I) = \{OUT\_OF\_RANGE, VAL\_ERR\}$. We can now represent the software/hardware FMMD analysis as a hierarchical diagram; see figure 5.

5 Conclusion

The FMMD method has been demonstrated using an the industry standard $4 \rightarrow 20mA$ input circuit and software. The derived component representing the $4 \rightarrow 20mA$ reader shows that by taking a modular approach for FMEA, i.e. FMMD, we can integrate software and electro-mechanical models. With this analysis we have stages along the 'reasoning path' linking the failure modes from the electronics to those in the software. Each functional grouping to derived component transition represents a reasoning stage. With traditional FMEA methods the reasoning distance is large, because it stretches from the component failure mode to the top—or—system level failure. We now have a derived component for a $4 \rightarrow 20mA$ input. Typically, more than one such input could be present in a real-world system: we can thus re-use this analysis for each $4 \rightarrow 20mA$ input in the system. Additionally, using FMMD we can determine a failure model for the hardware/software interface [15].

References

[1] Volker Bachmann and Richard Messnarz. Improving safety and availability of complex systems by using an integrated design approach in development. *Journal of Software: Evolution and Process*, 2012.

[2] Reliability Analysis Center. Failure mode/mechanisms distributions 1991. *United States Department of Commerce*, 1991.

[3] Neal Snooke Chris Price. An automated software fmea. *International System Safety conference singapore 2008*, 2008.

[4] R Clark, A Fish, C Garrett, and J Howse. Developing a rigorous bottom-up modular static failure modelling methodology. *6th IET International Conference on System Safety, 2011*, 2011.

[5] United States DOD. *Reliability Prediction of Electronic Equipment*. DOD, 1991.

[6] Peter L. Goddard. Validating the safety of embedded real-time control systems using fmea. *Reliability and Maintainability Symposium (RAMS), 1993 Proceedings - Annual*, 1993.

[7] Baiqiao HUANG. Software fmea approach based on failure modes database. *Reliability, Maintainability and Safety, 2009. ICRMS 2009. 8th International Conference*, 2009.

[8] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[9] Nancy Leveson. *Safeware: System safety and Computers ISBN: 0-201-11972-2*. Addison-Wesley, 2005.

[10] Gavin McCall. *MISRA:C:2004 Guidelines for the use of the C language in critical systems ISBN 978-0-9524156-4-0*. Hobbs, 2004.

[11] R. Mitchel. *Design By Contract by Example*. Adisson-Wesley, 2002.

[12] NASA. Fault tree handbook with aerospace applications. *NASA Handbook*, 2002.

[13] Shawulu Hunira Nggada. Software failure analysis at architecture level using fmea. *International Journal of Software and its applications Vol 6. 1*, 2012.

[14] N. Ozarin. A process for failure modes and effects analysis of computer software. *Reliability and Maintainability Symposium, 2003. Annual*, 2003.

[15] N.W. Ozarin. Applying software failure modes and effects analysis to interfaces. *Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual*, pages 533 – 538, jan. 2009.

[16] Winfield Hill Paul Horowitz. *The Art of Electronics*. Cambridge, 1989.

[17] US Nuclear reg commission. Fault tree handbook. *Nuclear Safety Analysis Handbook*, 1981.

[18] N Snooke. Model-driven automated software fmea. *Reliability and Maintainability Symposium (RAMS), 2011 Proceedings - Annual*, 2011.

[19] E N Standard. En61508:2002 functional safety of electrical/electronic/programmable electronic safety related systems. British standards Institution http://www.bsigroup.com/, 2002.

[20] E N Standard. En298:2003 gas burner controllers with forced draft. British standards Institution http://www.bsigroup.com/, 2003.

[21] Danhua Wang. An approach of automatically performing fault tree analysis and failure mode effects techniques to software. *Software Engineering and Data Mining (SEDM), 2010 2nd International Conference*, 2010.