An Object-Based Approach to Modelling and Analysis of Failure Properties

M. Čepin[1], R. de Lemos[2], B. Mavko[1], S. Riddle[2], A. Saeed[2]
[1] Reactor Engineering Division, "Jožef Stefan" Institute, Ljubljana, Slovenia

[2] Department of Computing Science, University of Newcastle upon Tyne, United Kingdom

Abstract

In protection systems, when traditional technology is replaced by software, the functionality and complexity of the system is likely to increase. The quantitative evidence normally provided for safety certification of traditional systems cannot be relied upon in software-based systems. Instead there is a need to provide qualitative evidence. As a basis for the required qualitative evidence, we propose an object-based approach that allows modelling of both the application and software domains. From the object class model of a system and a formal specification of the failure properties of its components, we generate a graph of failure propagation over object classes, which is then used to generate a graph in terms of object instances in order to conduct fault tree analysis. The model is validated by comparing the resulting minimal cut sets with those obtained from the fault tree analysis of the original system. The approach is illustrated on a case study based on a protection system from the Nuclear Industry.

Keywords: safety analysis, object-oriented modelling, fault tree analysis

1 Introduction

Increasingly traditional technology (hydraulic, pneumatic, electronic) is being replaced by software in process control systems. A typical consequence is that the functionality, and hence complexity, of the software-based system tends to increase, making the system harder to certify as assurance must be provided that the overall system risk is not increased. Evidence for certification is normally provided by conducting safety analysis.

The quantitative evidence normally provided for traditional systems cannot be relied upon in software-based systems, due to the difficulty of obtaining estimates of failure rates. Instead developers will need to place greater reliance on qualitative evidence. The object-based approach we propose begins with a model of an original system implemented in conventional technology, supported by the results of traditional safety analysis. The specification is used to derive an abstract object model which is independent of the technology in which it may be implemented. Safety analysis conducted on this model provides the required qualitative evidence that risk has not increased. It also establishes criteria for assessing and certifying the software to be developed, by providing a specification which reflects the failure properties of the original system.

2 Method description

The method proposed in this paper consists of a set of techniques from the software and application domains which are used to model the structure and behaviour of the existing system, and to conduct safety analysis. The result of the method is a formalised fault tree which can be directly compared with one produced for the original system.

The starting point for the method is a functional model of the original system, and its fault tree with resultant minimal cut sets. The method then proceeds as follows:

Object class model of system structure. The original system is analysed and, using the notation of the object view of OMT [5], an object class model is produced which represents the structure of the system.

Formal definition of class structure and behaviour. For each object class, its behaviour and structure is formally specified using a modified form of *interactor* [2]. Instead of employing operational techniques (e.g. statecharts) to express the dynamic view of OMT, we employ an axiomatic notation [1].

Causal analysis of object class model. From the object class model and interactor specifications, a causal model is derived for the propagation of failure behaviours through object classes. This step results in a graph of failure propagation over object classes which is termed an *impact structure* (modified from the form in which it appears in [7]).

Fault tree instantiation. The graph of failure propagation over object classes can then be instantiated for a particular top event and used as a basis to build a fault tree over object instances.

Minimal cut sets comparison. Comparing the minimal cut sets from the fault tree derived from the impact structure with the minimal cut sets derived from the original system, the validity of the object class model can be assessed: if any new causes are included in the cut sets then safety can be affected [10].

The techniques used in the method are established techniques employed in the fields of software engineering (OMT and first order predicate logic) and safety analysis (fault tree analysis and failure mode and effect analysis), and novel techniques for safety analysis (impact structure).

The paper illustrates the method outlined above by introducing as a case study a protection system for a Nuclear Power Plant. We begin by providing an overview of the system itself, and then go through each step in the method to perform the modelling and analysis of the case study.

3 ESFAS case study description

The case study system is the Engineered Safety Features Actuation System (ESFAS) employed as part of the protection system in a Nuclear Power Plant [8].

The system monitors parameters in the plant and, in the event of abnormal plant conditions, activates Engineered Safety Features, for example a safety injection signal. These safety features maintain the safety of the reactor by providing core cooling, so reducing the damage to fuel and fuel cladding, and preventing the release of radioactive materials.

The ESFAS initiates a safety injection signal if the value of one or more of the parameters (pressurizer pressure, steam line pressure, containment pressure) exceeds a defined safety limit during normal plant operation. The signal can also be actuated manually.

3.1 Structure of the ESFAS

The ESFAS consists of three or four redundant analog channels to measure each of the diverse pressure parameters, and two digital trains employing solid-state logic to vote on the actuation of a safety signal in the applicable conditions.

Redundancy is employed to ensure that no single failure can prevent actuation: only two out of the four channels are needed to provide an actuation signal. This is known as 2/4 voting, or 2/3 when there are only three channels. 2/4 voting is used when the same parameter is also used for control functions. Each digital train is capable of producing independently a safety injection signal.

3.2 ESFAS Case study simplifications

The resolution of modelling is selected so that only one parameter, measured on four channels, is monitored. The system for our purposes consists of two redundant trains, four redundant channels and two manual switches. The case study system is referred to as ***ESFAS_SI_Small***, the small version of the ESFAS Safety Injection system which is fully developed in [9]. The block diagram presented in Figure 1 represents the functional model of the system.

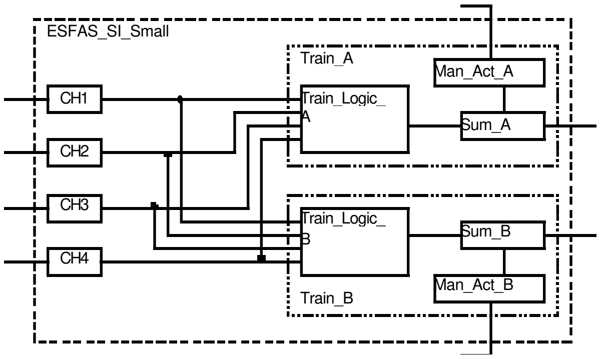

Figure 1: ***ESFAS_SI_Small*** Block Diagram

4 Case study application

Having provided an overview of the system we now illustrate the proposed method in more detail. We begin by forming an object class model of the structure of the system.

4.1 Object class model of system structure

We use the notation from the object view of OMT for our object class model, which shows the objects in the system and their relationships. The object class model is formed by:

- identifying object classes - examining components of the system and abstracting common entities;
- identifying associations - examining structural and inheritance relationships between object classes, which leads to a hierarchical diagram of aggregation and specialisation of object classes.

The resulting object class model for the case study is shown in Figure 2. This model decomposes ***ESFAS_SI_Small*** into an aggregation (signified by a diamond) of ***Channel*** and ***Train*** classes. The numbered black circles on the arcs signify multiplicity: there are four channels and two trains. The ***Channel*** has an *attribute*, or state variable, which is the value of the setpoint for that channel (a constant).

4.2 Formal definition of class structure and behaviour

From the object class model we now go on to formally specify the structure and behaviour of the object classes of the system. An *interactor* [1, 2] provides a formal axiomatic representation of the structure and behaviour of an object class. It is divided into declarations and predicates in a similar way to a Z schema [4]. Table 1 is an interactor for the ***ESFAS_SI_Small*** class.

The declarations consist of a *composed_of* field (defining the structure) which says that the object class consists of two instances of ***Train*** and an indexed sequence of ***Channel***s. The

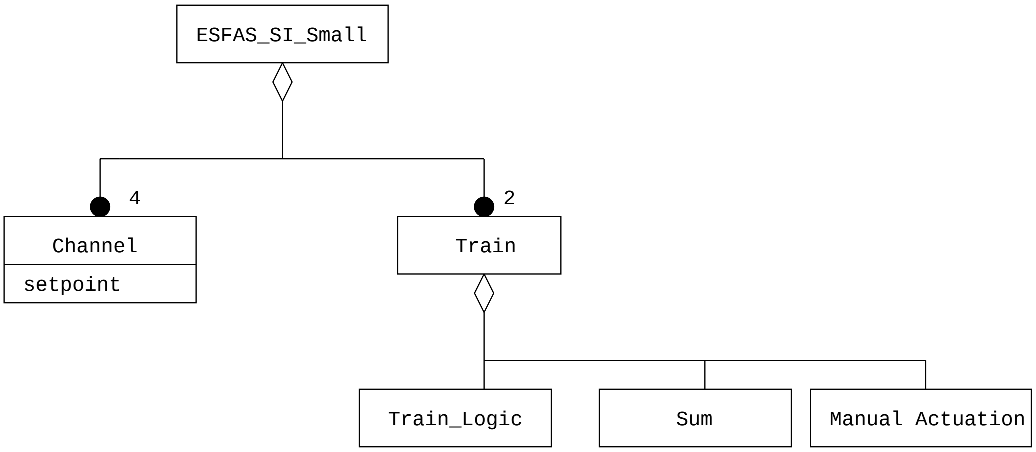

Figure 2: ***ESFAS_SI_Small*** Object Class Diagram

interactor ***ESFAS_SI_Small***		
composed_of:		
$train\_A, train\_B$	***Train***	
$chann$	seq ***Channel***	
constants:		
$setpoint$	$\mathbb{R}$	Pressure setpoint
variables:		
input variables:		
$reading$	seq $\mathbb{R}$	Pressure input readings
$manual\_A, manual\_B$	$\mathbb{B}$	Manual actuation inputs
output variables:		
$esfas\_A, esfas\_B$	$\mathbb{B}$	ESFAS actuation signals
state variables:		
$high\_pressure$	$\mathbb{B}$	
invariant:		
$high\_pressure \Leftrightarrow$ $(\exists\, i, j \in \{1...4\} \bullet (i \neq j) \wedge (reading(i) > setpoint) \wedge (reading(j) > setpoint))$		Pressure is high if and only if at least two channel setpoints are exceeded
structure:		
$\forall\, i \in \{1...4\}.chann(i).reading = reading(i) \wedge$ $train\_A.voters(i) = chann(i).channel\_signal \wedge$ $train\_B.voters(i) = chann(i).channel\_signal$		
$esfas\_A = train\_A.vote$		
$esfas\_B = train\_B.vote$		
$manual\_A = train\_A.manual\_act$		
$manual\_B = train\_B.manual\_act$		
behaviour:		
normal:		
$((high\_pressure \vee manual\_A) \Leftrightarrow esfas\_A)$ $\wedge ((high\_pressure \vee manual\_B) \Leftrightarrow esfas\_B)$		An actuation signal is produced if and only if pressure is too high, or the relevant manual actuation is present.
failure:		
$(\neg\, high\_pressure \wedge ((\neg\, manual\_A \wedge esfas\_A) \vee (\neg\, manual\_B \wedge esfas\_B))$		An actuation signal is produced despite no automatic or manual signal – spurious actuation
$\vee$		
$(high\_pressure \wedge (esfas\_A \wedge \neg\, esfas\_B)) \vee (\neg\, esfas\_A \wedge esfas\_B)$		Benign failure condition - only one actuation signal is produced
$\vee$		
$(high\_pressure \wedge (\neg\, esfas\_A \wedge \neg\, esfas\_B)) \vee (manual\_A \wedge \neg\, esfas\_A) \vee (manual\_B \wedge \neg\, esfas\_B)$		Failure to produce actuation signal when needed – critical failure

Table 1: Interactor for object class ***ESFAS_SI_Small***

indexed sequence is an abstraction of an array. Constants and variables are dealt with next: each declaration gives name, type and an optional comment for variable listed.

The predicates consist of an *invariant*, a *structure* field defining how sub-classes communicate (via input and output variables) and how these variables relate to the inputs and outputs of the object class being specified in the interactor, and a *behaviour* field which defines the possible behaviour modes of the object class.

In terms of the variables defined in the interactor, the requirements that the system must satisfy can be summarised as follows:

1. Whenever the high pressure threshold is exceeded on at least two channels, at least one of the trains must generate a safety injection signal:

   $high\_pressure \Rightarrow (esfas\_A \vee esfas\_B)$

2. The safety injection signal on each train can be independently actuated by a manual switch:

   $(manual\_A \Rightarrow esfas\_A) \wedge (manual\_B \Rightarrow esfas\_B)$

3. The injection signal should not be activated spuriously:

   $(esfas\_A \Rightarrow high\_pressure \vee manual\_A)$
   $\wedge\ (esfas\_B \Rightarrow high\_pressure \vee manual\_B)$

Disjunction of the above requirements characterises the normal behaviour of ***ESFAS_SI_Small***, specified in the interactor (Table 1) as

$$((high\_pressure \vee manual\_A) \Leftrightarrow esfas\_A)$$
$$\wedge\ ((high\_pressure \vee manual\_B) \Leftrightarrow esfas\_B)$$

A consequence of the normal behaviour is that $high\_pressure \Rightarrow (esfas\_A \wedge esfas\_B)$. The normal behaviour satisfies all the requirements.

Failure behaviours are identified either by negating the normal behaviour of the object class or by applying the FMEA technique to identify possible failure behaviours of the object class [6]. By negating the normal behaviour predicate for ***ESFAS_SI_Small***, we obtain

$$(\neg\ high\_pressure \wedge (\neg\ manual\_A \wedge esfas\_A) \vee (\neg\ manual\_B \wedge esfas\_B)$$
$$\vee\ (high\_pressure \wedge (\neg\ esfas\_A \vee \neg\ esfas\_B))$$
$$\vee\ (manual\_A \wedge \neg\ esfas\_A) \vee (manual\_B \wedge \neg\ esfas\_B)$$

The first line in this predicate is related with spurious generation of an actuation signal. The remaining lines are related with a combination of critical and benign failures (benign being the case when one train fails but the requirements are still satisfied). This can be rewritten to separate out the failure modes. The benign failure is specified as

$$(high\_pressure \wedge ((esfas\_a \wedge \neg\ esfas\_b) \vee (\neg\ esfas\_a \wedge esfas\_b)))$$

and critical failure is specified as

$$(high\_pressure \wedge (\neg\ esfas\_a \wedge \neg\ esfas\_b))$$
$$\vee\ (manual\_a \wedge \neg\ esfas\_a) \vee (manual\_b \wedge \neg\ esfas\_b)$$

Each of these failure conditions is represented by a failure mode (spurious, benign or critical) in the behaviour of the interactor.

The method defines an interactor for each object class in the model (***Train, Train_Logic, Sum, Manual_Actuation, Channel***). The ***Channel*** is an example of a primitive object class, which is specified by the interactor in Table 2. The failure behaviour is obtained from the negation of the normal behaviour in the same way as above: in this case there is no benign failure.

interactor ***Channel***		
constants:		
setpoint	$\mathbb{R}$	A channel has a constant setpoint
variables:		
input variables:		
reading	$\mathbb{R}$	pressure value
output variables:		
channel_signal	$\mathbb{B}$	channel output
behaviour:		
normal:		
$(reading \geq setpoint) \Leftrightarrow channel\_signal$		A signal is produced by the channel whenever the input reading exceeds the setpoint.
failure:		
$((reading \geq setpoint) \wedge \neg\ channel\_signal)$		Critical failure of the channel: no signal produced despite input reading exceeding setpoint
$\vee$		
$((reading < setpoint) \wedge channel\_signal)$		Spurious signal produced by channel when setpoint is not exceeded

Table 2: Interactor for object class ***Channel***

4.3 Causal analysis of object class model

In this section, we describe a systematic method for the derivation of a graph of failure propagation over the modelled object classes which provides the basis for the analysis of causes and consequences of failures. The graph is known as the *impact structure* [7] and is derived from the information provided within the object class model, and the interactor specifications.

The components of an impact structure are *nodes* which represent object classes and *arrows* (connecting classes) which represent an impacts relation between classes. An *impacts relation* exists between two classes, when the behaviour of one affects the other.

There are two types of impacts relations, impacts between sub-classes of a common aggregate class (intra-impacts) and impacts between a sub-class and its aggregate class (inter-impacts).

intra-impacts For classes $A.B$ and $A.C$ an intra-impact relation exists if $A.B$ impacts $A.C$, and is depicted by a solid arrow connecting $A.B$ to $A.C$.

inter-impacts For class A and sub-class $A.B$ an inter-impact relation exists if $A.B$ impacts A, and is depicted by a broken arrow connecting $A.B$ to A.

There are three ways of composing impacts:

n-impacts For two object classes A and B, B impacts A with multiplicity n if, according to the interactor specification, at least n instances of B must fail in order to impact an instance of A. An *n-impacts* is represented by annotating the arrow with a circle containing the particular number n.

and-impacts For object classes A, B and C, instances of both B and C must fail in order to impact an instance of A. An *and-impacts* is represented by linking the arrows involved in the relation with a $\otimes$ symbol.

or-impacts For object classes A, B and C, instances of either B or C must fail in order to impact an instance of A. An *or-impacts* is represented by linking the arrows involved in the relation with a $\oplus$ symbol.

The method for generating an impact structure is a three stage process, defined as follows.

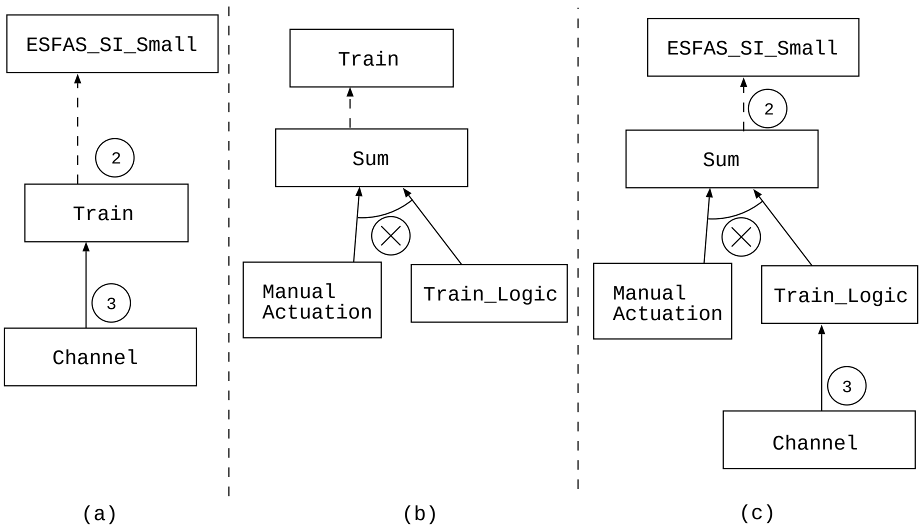

Figure 3: Impact structure for case study. (a) impact structure for the ***ESFAS_SI_Small*** class, (b) impact structure for ***Train*** subclass, (c) complete impact structure for the ***ESFAS_SI_Small***.

Stage 1: Determine Impacts

The impacts of an aggregate class are derived by an examination of the *composed_of* v and structure fields on an interactor.

1. Intra-impacts. A solid arrow is drawn between any two object classes A, B if the behaviour of one object class can affect the other, as specified in the structure field of the interactor for the aggregate object class.
2. Inter-impacts. A broken arrow is drawn between any two object classes $A.B$ and A if A has an output defined in terms of an output of A, B

Stage 2: Simplification of Impacts

The impacts structure obtained from Stage 1, can be simplified by the deletion of arrows and nodes. Firstly, redundant arrows are deleted by examining the transitive closure (impacts+); basically if the removal of an impacts relation will not change impacts+, the corresponding arrow is deleted. Secondly, nodes are deleted by substituting the impact structure of an aggregate class for the aggregate. After the substitution, those components with an impacts relation to the aggregate class will inherit the impacts of that aggregate.

Stage 3: Composition of Impacts

The composition of the impacts obtained from Stage 2, can be identified in two steps.

1. Multiple composition. When there are multiple instances of a class that impact another class, the interactor specifications are examined to determine the multiplicity
2. Logical composition. When more than one class participates in an impact relation with another class, the interactor specifications are examined to determine if the behaviour corresponds to an *or-impacts* or an *and-impacts*.

Figure 3 shows the evolution of the impact structure for the ***ESFAS_SI_Small*** case study. Figure 3.a illustrates the impact structure for the class ***ESFAS_SI_Small***, and depicts an inter-impacts relation between the subclass ***Train*** and its aggregate class ***ESFAS_SI_Small***, an intra-impacts relation between the subclasses ***Channel*** and ***Train***, and a *3-impacts* composition between ***Channel*** and ***Train***. Figure 3.b illustrates an *and-impacts* between subclasses ***Manual***

Actuation and ***Train_Logic*** and subclass ***Sum***, Figure 3.c illustrates a simplified structure derived by merging figures 3.a and 3.b, deleting the subclass ***Train***.

The impact structure provides a compact representation of failure propagation through object classes, thus facilitating the process for conducting a cause–consequence failure analysis of the object class model. Thereby providing an essential step to the systematic derivation of a fault tree from an object class model. This is achieved by instantiating the impact structure of the object class model, in order to obtain a structure representing failure propagation over object instances, from which a fault tree can be then generated.

4.4 Fault tree instantiation

This section presents how to derive a fault tree, for a particular failure event, from an impact structure of an object class model and the specification of interactors of the classes.

A fault tree consists of *nodes*, which represent failure events, and *gates* ("and", "or", and numerical) which represent the causal relationships between the nodes [11].

The method for generating the fault tree from the impact structure and the specifications of the interactors proceeds in two stages:

Stage 1: Generating the nodes

- Some of the nodes of the fault tree are formed by instantiating the impact structure, thus producing a node for each object instance of the classes in the impact structure.
- The nodes related to aggregate classes are split into two nodes. A node representing the failure behaviour of the object instance, which is a primitive node, and another node representing those failure behaviours which are to be refined at lower levels.
- For the particular failure event being analysed, each resulting node is annotated with the respective axiomatic specification of the behaviour from the relevant interactor, and when necessary modified to incorporate failure behaviours which are to be refined.

Stage 2: Generating the gates

- The gates of the fault tree are formed by referring to the composition of the impact (*and, or, n*) and introducing the relevant gate. An "or" gate is also used to connect two nodes that were produced by splitting the node representing an aggregate class.

Figure 4 shows the fault tree resulting from the application of this method to the ***ESFAS_SI_Small*** case study. The top node is annotated with the critical failure of the object class ***ESFAS_SI_Small***, the failure event for this tree. The fault tree contains primitive nodes `SUM_A` and `SUM_B` representing the failure behaviours of the class ***Sum***. Similiarly primitive nodes are provided for the failure behaviours of ***Manual_Actuation***, ***Train_Logic***, and ***Channel***. The other intermediate nodes (generated from splitting an aggregate class) of the fault tree represent failure propagation from the primitive failure events to the top failure event. An example of an intermediate node is `NO_SIGN_A`, which represent the propogation of failures from lower levels when `SUM_A` exhibits normal behaviour.

There is an "and" gate joining the failure events assocaited with the two ***Sum*** instances, since the impact structure had an *2-impacts*. The ***Channel***s are connected by a numerical gate "≥ 3", since there are four instances and the impact structure had an *3-impacts*.

The fault tree produced directly from the original system model is shown in Figure 5. Both fault trees have been "pruned" for reasons of space.

In order to validate that the failure properties of the object class model are consistent with the original system model, the minimial cut sets of their respective fault trees are compared [10, 3]. A minimal cut set of a fault tree is a sets of failure, such that if any event of the set occurs the top failure event occurs. For example, one of the minimal cut sets resulting from the fault tree generated from the impact structure: MAN_ACT_A, MAN_ACT_B, TRAIN_LOGIC_A, TRAIN_LOGIC_B.

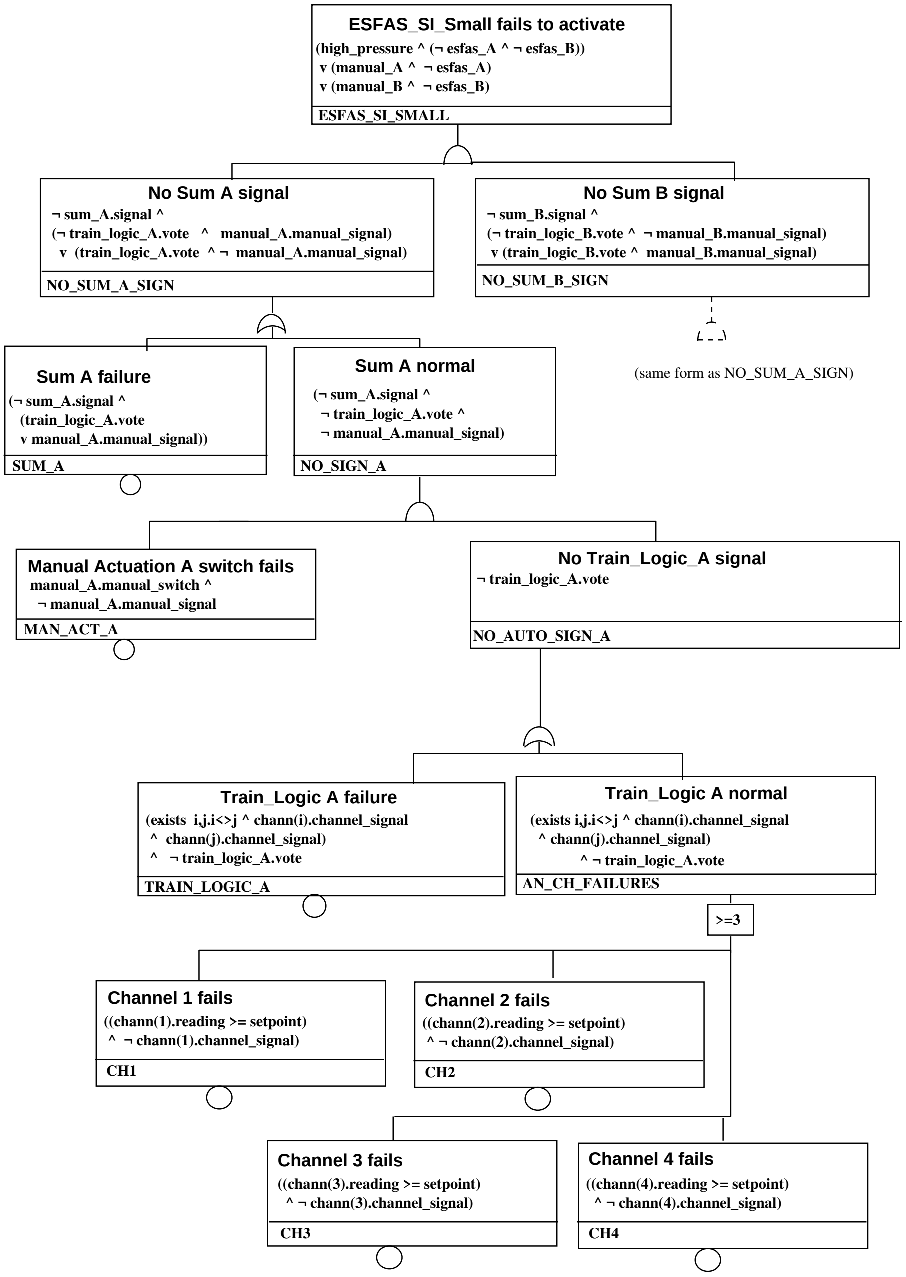

Figure 4: Fault tree generated from impact structure

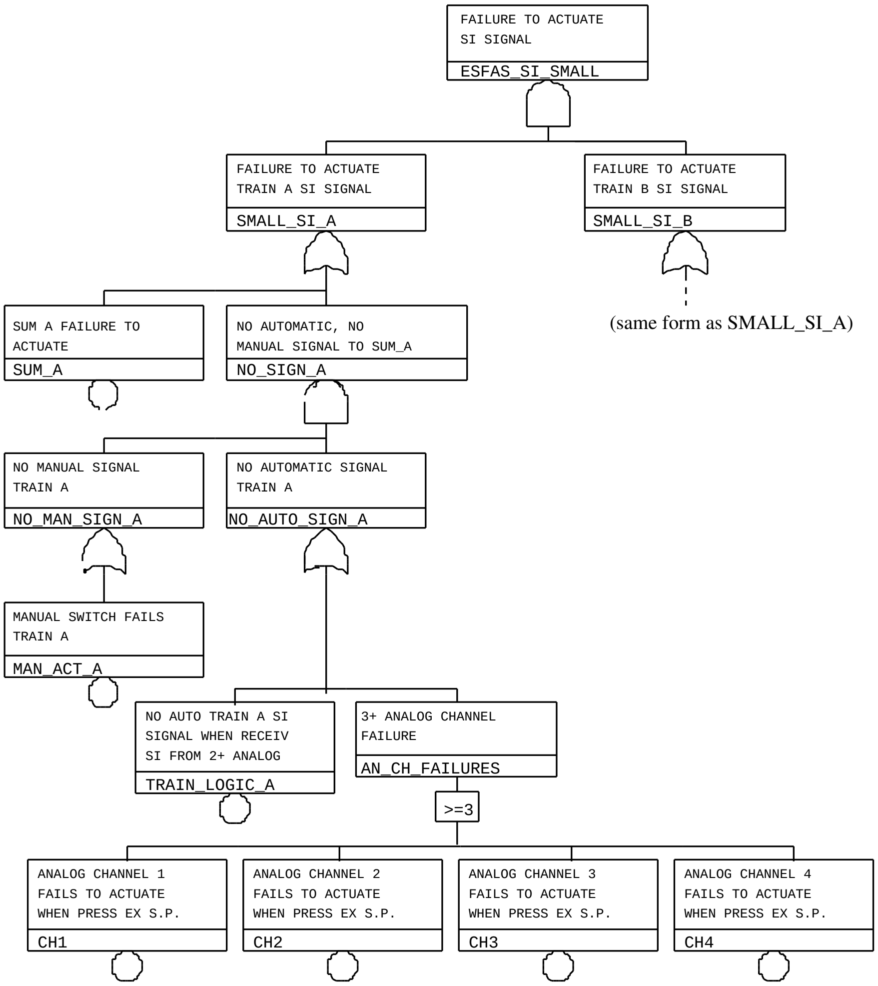

Figure 5: Fault tree generated from original model

If the minimal cut sets are equivalent, we can conclude that the new model is valid with respect to the original system model. If not, the minimal cut sets will either be overlapping or (in the worst case) disjoint. If there is an overlap the reasons must be investigated – a new failure may have been created, or a failure from the original model may have been removed, in the case of disjoint cut sets imply totally incompatible models. For this case study, the minimal cut sets resulting from both fault trees are identical, which serves to validate the model.

5 Conclusions

The proposed method tackles the problems encountered by application domain engineers, when investigating the feasibility of replacing a traditional system with a software based system. The method provides qualitative evidence that a software based system can be developed which exhibits the same failure properties as the traditional system. Thereby, providing assurance that the software will not increase the risk posed by the system.

The method has been applied to a larger case study [9], which demonstrates the scalability of the work. Further work includes formalising causality and investigation of applicability of other safety analysis techniques.

Acknowledgements

The work reported in this paper was carried out as part of the EC Copernicus Joint Research Project ISAT (Integration of Safety Analysis Techniques for Process Control Systems).

References

[1] R de Lemos, B Fields, and A Saeed. Analysis of safety requirements in the context of system faults and human errors. In *Proc. IEEE International Workshop on Systems Engineering of Computer Based Systems*, pages 374–381, Tucson, Arizona, March 1995.

[2] D Duke and MD Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.

[3] J Górski and A Wardziñski. Formalising fault trees. In *Proc. Safety-Critical Systems Symposium*, pages 310–327, Brighton, United Kingdom, 1995.

[4] B Potter, J Sinclair, and D Till. *An Introduction to Formal Specification and Z.* International Series in Computer Science. Prentice-Hall International (UK) Ltd, United Kingdom, 1991.

[5] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorenson. *Object-Oriented Modelling and Design.* Prentice-Hall International Inc., New Jersey, 1991.

[6] A Saeed, R de Lemos, and T Anderson. An approach for the risk analysis of safety specifications. In *Proc. 9th Annual Conference on Computer Assurance (COMPASS '94)*, pages 209–221, Gaithersburg, MD, June 1994.

[7] A Saeed, R de Lemos, and T Anderson. Safety analysis for requirements specifications: Methods and techniques. In *Proc. 14th International Conference on Computer Safety, Reliability and Security: SAFECOMP '95*, pages 27–41, October 1995.

[8] M Čepin and B Mavko. Identification and preparation of case studies. Technical Report TR ISAT 96/8, Jožef Stefan Institute, Ljubljana, Slovenia, June 1996.

[9] M Čepin and S Riddle. Object modelling and safety analysis of Engineered Safety Features Actuation System. Technical Report TR ISAT 96/11, University of Newcastle upon Tyne, United Kingdom, December 1996.

[10] M Čepin and A Wardziñski. On integration of probabilistic and deterministic safety analysis. In *Proc. 3rd Regional Meeting: Nuclear Energy in Central Europe*, Portoroz, Slovenia, 1996. (To appear. Also Technical Report TR ISAT 96/9, Jožef Stefan Institute, Ljubljana, September 1996).

[11] WE Vesely, FF Goldberg, NH Roberts, and DF Haasl. Fault tree handbook. NUREG 0492, US NRC, Washington, 1981.