Safety Cases for Software Application Reuse

P Fenelon, T P Kelly, J A McDermid

High Integrity Systems Engineering Group,
University of York, Heslington,
York Y01 5DD, UK

e-mail: pete, tpk, jam @ minster.york.ac.uk

Abstract

In traditional engineering industries it is common to reuse tried and trusted components as one of the means of ensuring safety. Some low-level software components, e.g. libraries, are reused, but there are difficulties in justifying the reuse of software due to the complexity of interactions in a typical software system. This paper addresses the issue of reusing software applications by considering how to extend the safety case for the use of software in one application based on its use in another. It proposes an approach to analysing the change between two contexts of use of a software system, including analysing small changes in the software, and illustrates this through some examples based on an analysis of a reactor protection system.

1 Introduction

There is a long-established principle, in engineering safety-critical systems, to reuse tried and trusted components as a means of ensuring safety. These principles have been adopted in developing safety-critical software, but often only through the reuse of low level components, e.g. generic components of control laws. There is potential for considerable benefit in reusing larger software components, perhaps even a complete software 'application'. In practice it is unlikely that an item of software will be moved entirely unchanged from one domain to another but, even if it is unchanged, it is not obvious that the safety case can be preserved — the new environment may place new demands on the software, it may have a different distribution of demands thus rendering invalid existing statistical reliability data, and so on. Thus there is a challenge — to adapt the safety case to show how the reuse of an application (or part thereof) in a new domain may be justified. This amounts to adaptive reuse of the safety case, as well as reuse of the software.

We perceive re-use of safety cases as being valuable as the construction of a safety case is an expensive operation involving considerable time and effort on the part of engineers and managers from many disciplines [Ball89]. Almost inevitably however reuse involves change. Even if an application is well-suited to a new domain, small changes in the software will normally be needed. Thus we need a sound method for analysing the potential impact of change, both in the domain and in the software, identifying the aspects of the system design or operation which need to be re-assessed, then justifying the safety of the system in the new domain by reusing and extending the safety case. Such a method will have more than just economic benefits — it will avoid some classes of potential error in analysis and thus may contribute to safety.

We have developed some principles for structuring safety cases, for categorising imposed change (to the software, the system in which it is embedded and its operating environment), and for analysing the impact of change. These principles have been applied retrospectively to the safety case for a reactor protection system (the Stage 9 submission for the Dungeness B SCTS [NE91]), analysing several changes which were made late in the development of the system. In developing these principles, it became clear that they could also apply to the analysis of changes within the same environment — thus we believe that they are of general applicability in safety case evolution and maintenance (special cases of reuse).

Our aim in this paper is to illustrate the principles, and to show how they aid the analysis of change. For brevity we only describe a subset of the principles we have developed, and present a fragment of the example we have undertaken. The details of the example have been elided for ease of presentation, but the conclusions drawn accurately reflect the results of the study. We also include extensions of the analysis principles developed after the project was completed.

We first illustrate the problem in more detail, then present some of the key principles we have developed. The use of the principles is illustrated on a fragment of the safety case we analysed. The example is used to draw some conclusions about the utility of the approach, and to indicate what further developments would be needed to use the approach during development, rather than in retrospectively.

2 The Problem

Safety cases, like other products of complex engineering processes, are developed in an iterative manner, thus they are constantly subject to change during their development. Safety cases may also need to be reconsidered and changed after deployment of the system to which they refer. This may happen for a number of reasons, e.g. because of an unanticipated problem with the system, see for example [Hogberg94], because of a change in requirements or standards, e.g. the NUREGs [Queener94], or the desire to extend plant life [Clarke89] and the need to deal with operational history and changes in standards.

In supporting design iteration we need efficient ways of modifying the safety case, propagating the change and ensuring that we have re-established consistency of the case. We refer to this as *making changes*. In responding to external changes we are concerned with reasoning about the impact of change, showing that the system is still safe in the face of the change, or determining what consequential change is needed in order to preserve safety. We refer to this as *reasoning about change*.

In general we are concerned with an *imposed change*, i.e. one that is made or proposed outside the developer's control, and *consequential change*, i.e. one made by the developer in order to respond to the imposition, to produce an acceptable system and safety case. Our focus is on reasoning about imposed change, although we will need to discuss the analysis of consequential change to give a complete treatment of the situation. In order to handle imposed change efficiently we need to be able to preserve the initial safety case, so far as practical, adding the results of reasoning about the imposed and consequential changes to the safety case to show that the modified system is still safe.

We can illustrate the issues which arise using a simple scenario: a software-controlled reactor protection system, which trips on measured core temperature. Let us say that the reactor has been operating safely for some time with the trip level (referred to as the *set point*) at X degrees, and it is decided to raise the set point to X + 1 degrees (the reactor can be run more efficiently, hence more profitably, at a higher temperature, but the safety margin is reduced). Thus the change of 1 degree in the set point is the imposed change. What reasoning about change is required?

First, the reactor physics need to be considered to ensure that the change is safe in terms of the core operation. Second, evidence needs to be provided that the software implementing the trip function is still safe. The nature of the requisite evidence depends on the details of the change. It is instructive to consider some possibilities.

If the protection system software is recompiled with a change in a literal constant for the trip value, and the compiled code is bitwise identical to the previous version except for the constant, then we can argue that the previous testing evidence, etc. can be carried over to the new system as the change has not affected the program structure, the coverage of the testing programme, etc.

However, if the constant (set point) was changed from, say, 127 to 128 and, as a result, the compiler changed the code from an integer comparison to a shift followed by a 'branch if not zero' instruction (for speed), allowing further optimisation changing the executable code structure, then the testing results would not carry over directly. In this case more tests might be used to reconstruct the safety case. Thus the reasoning about change would be that the original test data, plus the additional tests and test results, were sufficient to show that the change was acceptable. This 'delta' on the safety case is needed to allow the original safety case to be reused.

In general it is a good policy to separate data such as trip points from the code, to simplify reasoning about change. In practice we can accommodate 'small changes' but there is a difficulty in determining when changes cease to be small and the software component needs to be treated as new. The conservative approach is to assume that all changes are 'large', and repeat the complete analysis. Our intention is to offer a more cost-effective way of dealing with small changes.

3 The Principles

The principles we developed were based around a particular approach to the structuring of safety cases, and some evolving techniques for software safety analysis. We amplify on the safety case approach and structuring principles below. We also developed a taxonomy of change for analysing both imposed and consequential change. Whilst this is important as part of an overall method for change analysis, we only need a small part of the taxonomy for discussing the examples, so we give a simple overview of the relevant ideas in section 3.4 below.

3.1 Types of Safety Case

In developing the principles, we identified two forms of software safety case — 'black box' and 'white box'. The 'black box' case places reliability figures on the software, and assumes no knowledge of the structure of the software. Thus, with a

'black box' approach it is necessary to use purely stochastic arguments to justify the deployment of the software (application) in the new domain, or to take into account the impact of an imposed change. Even an imposed change in the distribution or frequency of demands on the system could render the (relevant portion of the) safety case invalid — or require very subtle statistical analysis. Consequently we introduced the notion of a 'grey box' scenario, where we took an essentially 'black box' situation, and derived a small amount of pertinent structural information about the software, perhaps using reverse engineering — for example, so that we could show that test results still gave sufficient path coverage to sustain the safety case. However, none of our examples were based on a statistical analysis, so we do not consider the 'black box' approach any further.

The 'white box' approach constructs a deterministic safety case, at least so far as the programs are concerned. Thus the 'white box' case looks at the internal program structures, and employs software engineering and safety analysis techniques, e.g. static analysis and software fault-trees [Leveson83], to show that the proposed deployment of the software in the new domain is sound, or that the imposed change is benign. Probabilities of hazardous events are still derived, but the potential software contribution to a hazard is represented in the structure of the fault-trees, not as a probability of the software failing. Our examples were based on this 'white box' approach. The safety analysis techniques which we used in carrying out the examples are outlined in section 3.3.

3.2 Safety Case Structuring

Most safety cases are structured around a hazard log. We believe that this is appropriate to provide an index into a safety case, once it has been constructed, but inadequate to guide the construction of the safety case. In our approach we use the notion of *goal structuring* to assist in the construction of the safety case, to facilitate the management of scale and complexity in safety cases, and to assist in change management. Some of our early experience in using goal structuring in safety cases are reported in [McDermid94], and more recent applications to the nuclear domain are described in [Wilson95a]. The most fundamental concepts are:

- *goal* — is something that a stakeholder in the design and assessment process wishes to be achieved;
- *strategy* — a strategy is a (putative) means of achieving the goal or collection of goals, e.g. a system concept, or a sequence of activities.

Goals are decomposed through the strategies, and we will refer to sub-goals where this is helpful. Goals may refer to specific technical properties, or may be more general, e.g. to do with commercial objectives such as power plant profitability. Typically the high level goals in a safety case will reflect general principles, and the lower level goals will be interpretations of (derived requirements arising from) those principles for the particular system being considered. For example, the HSE set out some 333 safety principles [HSE92], a subset of which might form part of the top level goals of a safety case, see Figure 1. Here the term 'criticality based incident' refers to situations such as failing to trip which leave the power plant in a critical

situation. Such incidents are distinct from, and less severe than, directly hazardous events such as a release of radioactive material to the environment.

Goal G1 states the intention to comply with the HSE safety assessment principles. The strategy is shown below the goal, and is to work to relevant principles, e.g. the principle (P126) to evaluate the probability of an aircraft crashing into the reactor may be deemed inappropriate for a reactor in a submarine. The sub-goals, G1.1 ... G1.N, are the specific principles which have been selected, and the ellipse is the justification for selecting these principles (see below). Our safety case support tool, SAM [Wilson95b], enables goals to be given simple names, and more details associated with the goals to be stored in a database. For example, a full description of a goal from figure 1 might include the full principle definition, e.g. G1.2 — 'A reactor should be provided with systems which can shut it down safely ...'.

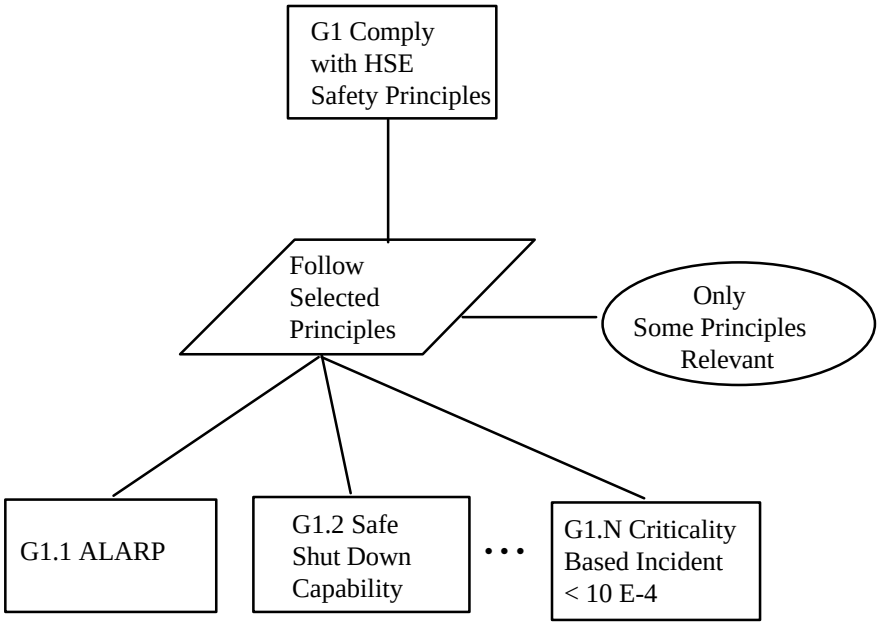

Figure 1: Example Goal Hierarchy Fragment

Some goals may be satisfied directly, e.g. by carrying out an action, or providing a product with the right properties. We use the term *solution* for the action or product which satisfies a goal. Goals with solutions are *leaves* of the goal structure, i.e. they have no strategy or sub-goals. Solutions are typically safety analyses, although they may be arguments (see section 4). Thus a lower level fragment of a safety case might be as shown in figure 2. The solution is represented as a circle and, in this case, is a system level fault-tree analysis.

The lower level goals are traceable to (from) the HSE principles. The terms Pfl and Pfh stand for probability of failing to trip on low temperature and high temperature, respectively. Here we have shown a common form of decomposition, where a particular criticality based incident — failure to trip on demand — is decomposed

into lower level goals representing the contributory failure modes. The justification states that the budgeting (allocation) of rates between the different modes is based on historical evidence/experience.

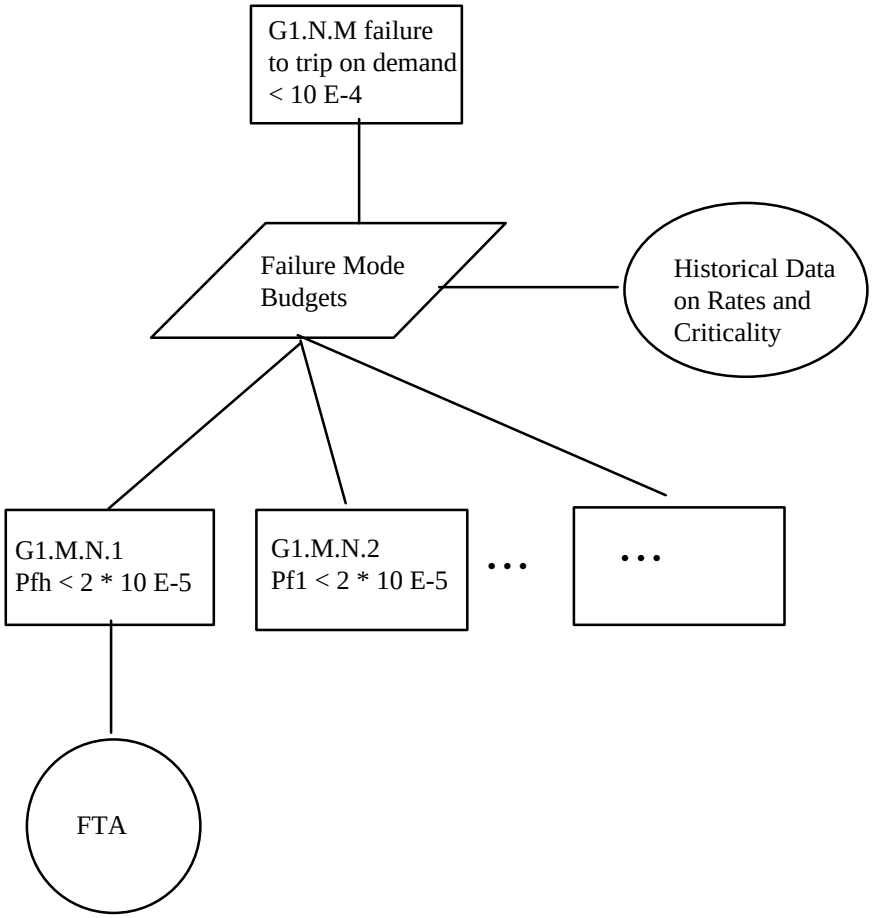

Figure 2: Goal Hierarchy Fragment Showing Solution

We use the term *constraints* to refer to those goals which are not solved directly, but which restrict the way in which other goals are solved, i.e. which restrict the set of allowable strategies (and models, see below). The satisfaction of constraints must be checked at multiple points in the goal hierarchy. Common safety requirements such as 'no single point of failure shall lead to a hazard' are representative of this class of goal. The HSE principles contain a number of constraints (in our terms). For example 'Defence in Depth' and 'Diversity in Detection and Control' are guidelines or constraints against which the design will be evaluated at many different stages in the design decomposition process.

There are other important facets of the structure which will be related to goals or strategies. These include:

- *models* — these represent part of the system of interest, its environment or the organisations associated with the system; goals will often be stated in terms of models, especially when they represent the system design;
- *justification* — a justification is an argument, or other information, e.g. the results of a safety analysis, presented to explain why a strategy is believed to be effective, i.e. that it meets the goals.

In our example, the models of interest are representations of the structure of the software in the trip system. In general, the goals and justifications give a basis for determining the impact of change, see section 3.4.

3.3 Safety Analysis Techniques

In general, we assume the use of standard safety analysis techniques, including the application of fault-trees to analyse software [Leveson83]. However, we also make use of one relatively new technique known as Failure Propagation and Transformation Notation (FPTN) [Fenelon93]. The purpose of this notation is to summarise the 'flow' of failures through complex integrated systems — hence the term propagation. As failures 'flow' they can be become transformed, e.g. the omission of a message may be detected by a watchdog timer, and an extrapolated value substituted, changing the omission failure to a value domain failure — hence the term transformation.

The notation is graphical, and it effectively acts as a summary for a set of fault-trees, enriched with the notion of failure types — omission. commission, etc. Thus FPTN can be thought of as being the functional analogue to FMES (Failure Modes and Effects Summary) — providing a succinct representation of the failure modes and effects, but structured around the system *functional* decomposition, not its *physical* decomposition. (FPTN is also intended to be used as a means of defining derived safety requirements, but we will not discuss this further, as we do not use the notation in this mode in our examples.) FPTN is represented as follows:

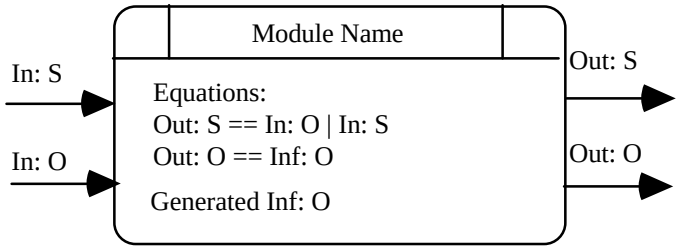

Figure 3: Simple FPTN Module

The notation is similar to Yourdon data-flows, but it is concerned with failures, not data flows. The round-cornered box corresponds to a software module, typically a process or a procedure. The inputs and outputs are failures, not data flows, but they are normally failures associated with the data flows, e.g. wrong data values. The labels before the colon are the flow name, and those after the colon are the 'types' of the failures (see [Bondavalli90] and [Pumfrey94] for more detail on failure types). The types used are O — Omission, C — Commission, S — Subtle, C — Coarse, E

— Early and L — Late. The equations are representations of fault-tree cutsets. They say how the output failure modes are related to the input failure modes — in this case the subtle failure is propagated, and the omission failure is transformed into a subtle value failure. The generated failure arises from an omission failure of the computing infrastructure (Inf) which leads to an omission from the module.

In general we need a collection of FPTN modules to describe a system. We will get chains, or networks, of FPTN modules, and we need to allow module hierarchies, to describe a complex system. At the current state of development the method is largely qualitative, but we are investigating the issues in developing a quantitative calculus based on the FPTN structures. FPTN was developed to model properties of software systems, but can equally well be applied to other forms of discrete/digital system.

In analysing change, we need a further concept, that of the safety critical path (SCP). If a failure condition of a system, represented as an FPTN failure mode, is hazardous (critical) then all the modules which have failure modes that contribute to the hazardous failure mode are members of the SCP. We shall see that the notion of an SCP is important in sections 3.4 and 4, as the criticality of a change depends on whether or not it affects the SCP.

3.4 Taxonomy of Change and Analysis Approach

The analysis approach we developed has three main components:

- a taxonomy for identifying and classifying imposed changes in the environment and the system in which the software is embedded, and for classifying the consequential changes to the software or system;
- a form of analysis, based on the taxonomy and software safety analysis techniques, for identifying which changes are *benign*, and which pose a *challenge* to the system, software or safety case;
- a set of template arguments for justifying the use of the application in the new domain, based on the taxonomy and analysis.

The notions of changes which are benign or pose a challenge is central to the approach. Benign changes do not require any consequential change. An obvious category of benign change is one that either increases the ability of a system to meet its goals, e.g. increases its reliability, or reduces a requirement, e.g. a demand rate for a trip system. Other changes may also be benign, e.g. an increase in a trip demand rate might be benign if there is sufficient margin between the goal and the achieved reliability of the system. A challenge arises where a consequential change is needed to ‘re-substantiate’ the overall safety case.

In general, we do not know *a priori* whether an imposed change is benign or poses a challenge, thus we need a process for analysing the impact of the imposed change:

1 categorise the imposed change — determine the source of the change, e.g. which system or software components are affected, and which goals are directly affected by the change, e.g. apply to the changed component;

2 categorise the change as benign or a challenge (by determining whether or not all the immediate goals are still met);

3 if the change is benign, present an argument why this is the case, e.g. reliability margins are reduced, but the design is still acceptable;

4 if the change is a challenge, determine and make a response (consequential change); note that this may not involve the system, but could be a change in operating procedures, or further testing to gain more evidence of safety;

5 repeat steps 1 to 4 for the consequential change, until all the consequences are shown to be benign, i.e. goals are reached which are now satisfied;

6 present the arguments why the system, incorporating/allowing for the consequential changes is safe (meets its goals).

FPTN can help to distinguish between benign and challenging software changes. An imposed change which is on an SCP is a potential challenge, as is a change which brings some module onto an SCP, when previously it was not on the SCP. A potential challenge is an actual challenge if it causes the goals immediately dependent on the changed module no longer to hold (or puts this in doubt). Changes which do not affect the SCP, or those which do but which do not affect the immediate goals are benign. Thus, for software-based systems, we use FPTN as a key element in step 2 of our approach to change analysis.

Note that the fact that an imposed change represents a challenge does not mean that the system or software will need to be modified as a consequence. It may simply be that more evidence is needed to 'reconstruct' the safety case. Also note that the goals act as 'barriers' — change does not propagate any further if a goal is still met. This principle applies both to imposed and consequential changes.

4 An Example

The example is based on a safety case for a reactor trip system (the Stage 9 submission for the Dungeness B SCTS), and some changes that were imposed. We considered several different situations which affected this system. In all cases, there were new or modified requirements, but the aim was to leave the software unchanged, so far as practical, so as to avoid the cost of reverifying the modified software (see section 2). We briefly describe two of the changes then discuss their analysis in the terms introduced above. For presentational purposes, some details have been suppressed, but the structure of the arguments and the overall findings are unmodified.

4.1 Imposed Changes

The two imposed changes which we will consider are:

1 decreasing the maximum tolerable range of sensor values (i.e. the acceptable difference between the minimum and maximum recorded temperatures);

2 increasing the set point for high temperature trip.

Both changes present potential challenges, but the first is easy to justify as benign by a 'delta' to the safety case, as we shall show.

4.2 System Models

There are three system models of interest: the physical structure of the hardware, the logical structure of one lane of the software and relevant portions of the code. These are as shown in the following figures.

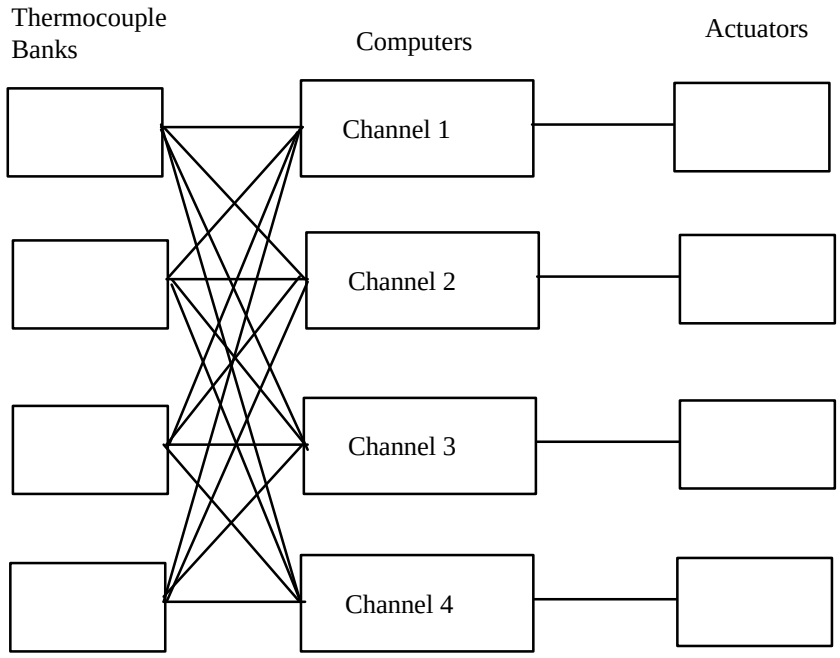

Figure 4: Outline Hardware Structure

The hardware is structured so that each channel needs to periodically send a signal to the actuator logic, otherwise the actuators will trip. This makes the design fail safe, in the event of power loss (another HSE principle).

The functional structure of a single channel is (in much simplified form):

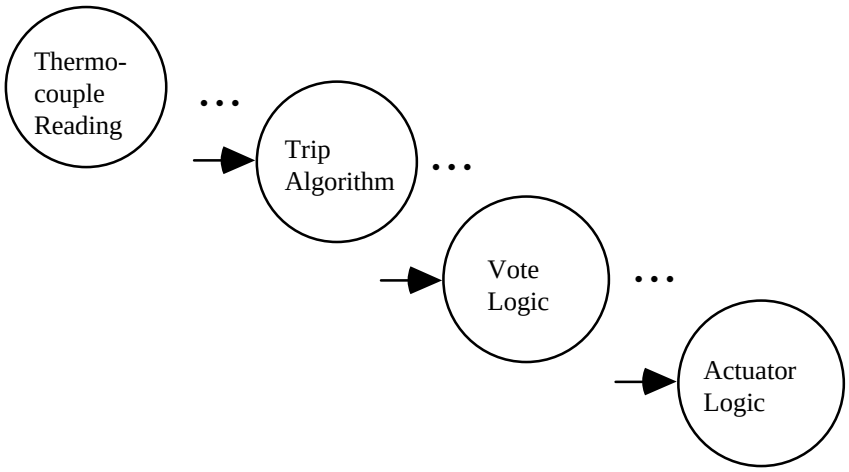

Figure 5: Simplified DFD for Channel Software

Each function is executed periodically. In the complete system there are a lot of functions in each channel, e.g. signal conditioning, but we show some of the key functions above. We will focus on the trip algorithm. A relevant fragment of the trip algorithm code is shown below. It was derived from published trip algorithm specifications, rather than being taken from the system, and is represented in Ada to enable us to use some of our analysis tools [Fenelon93], and simplified for the purposes of exposition. The fragment is:

```
procedure trip_alg is
...
if range_test
      -- $SSAP bg range_trip
      -- affected by change 1 (reduction in max_spread)
      t := range_calc;
      if t > max_spread
          then
              for s in signals loop
              if set(s)
                  then demand_trip (b, s);
              end if;
      end if;
      -- $SSAP eg range_trip
else
      for s in signals loop
          if set(s)
              then
                  -- $SSAP bg hot_trip
                  -- affected by change 2 (reduction in hot_set_point)
                  if temp(s) > hot_set_point
                      then demand_trip (b, s);
                  end if;
                  ...
                  -- $SSAP eg hot_trip
```

Figure 6: Fragment of Trip Algorithm

The variable signals is an array of augmented thermocouple readings pre-processed by the signal conditioning logic. Those signals which are set are from thermocouples which are deemed to be functioning correctly by the signal conditioning logic.

The range test logic determines the maximum spread of the temperature readings, and demands a trip for all the signals which are set. The temperature trip logic cycles through the pre-processed thermocouple signals, and demands a trip for any signal which exceeds the threshold. The function demand_trip passes a trip request to the voter logic which is the next function in the chain. It might be thought that an

omission failure here could be hazardous, however such eventualities are addressed through the hardware redundancy, not the software structures.

The comments -- $SSAP are directives for a prototype fault-tree tool which we used to analyse the software. The terms bg and eg stand for 'begin group' and 'end group' respectively. These groups are 'components' of the program to be analysed, e.g. from which to produce a fault-tree.

These system 'models' serve as the basis for the analysis of change. They are far from sufficient to represent all the safety-relevant facets of the system, e.g. a real trip algorithm is considerably more complex, but are sufficient to illustrate the principles we have developed.

4.3 System Goals

The directly relevant goals for this analysis are G1.M.N.1 and G1.M.N.2 from Figure 2, dealing with the low and high temperature trip probability, respectively. In practice there are additional relevant goals to do with spurious trips (this is an availability issue, not a safety issue, but needs to be considered in developing the safety case), and concerned with operator procedures, e.g. vetoing thermocouples during maintenance operation. Some of the changes we considered had an impact on these goals, but they cannot be addressed in detail within the confines of this paper.

4.4 Safety Analysis

We are concerned with failure to trip on demand, e.g. satisfaction of goals G1.M.N.1 and G1.M.N.2. In considering the trip algorithm this corresponds to the omission of a call to demand_trip, and the consequent failure to request a trip from the voter. For the system overall to fail to trip on demand, we need to have a number of failures, but in change analysis we can focus on the trip algorithm.

The fault-tree for the hot set point logic is as follows. The fault-tree is generated in the spirit of the rules derived by Leveson et al [Leveson83]. The basic form of the fault-tree is produced automatically from the annotated Ada code. The top event is annotated with HighTrip: O to represent the fact that this is an omission failure. This label is used so that the correlation with the event in the FPTN is apparent. Similarly one of the leaves of the fault-tree is labelled to show the effect of failures of functions 'upstream' of the trip algorithm. The labelling with the FPTN failure types is currently a manual operation.

The fault-tree is the solution to the goal G1.M.N.1 in figure 2. In the complete safety case, there would be probabilities associated with some of the leaf events, representing the likelihood of particular hardware failures, e.g. thermocouple mis-readings. This would enable us to provide a top even probability, and show that goal G1.M.N.1 in figure 2 had been met. For our examples we do not need to make statistical arguments, so we do not include and failure rates or probabilities in the fault-trees.

A partial FPTN representation of the trip logic, sufficient for our analysis, is shown in figure 8. This shows possible input failures to the Trip Algorithm, specifically Omission and Subtle value failures of the conditioned thermocouple signals. The

figure shows how the fault-tree of figure 7 contributes to the FPTN. Note that the FPTN for HighTrip includes a failure propagation not in the fault-tree: this arises because there is nothing in the logic of the software fault-trees to explain what happens if the software is not executed. Clearly, this is not a fault in the analysis approach; it merely points out that the FPTN needs to integrate analyses arising from a number of different sources. In a full analysis, we would also be interested in HighOut: C, etc. as these will represent spurious trips.

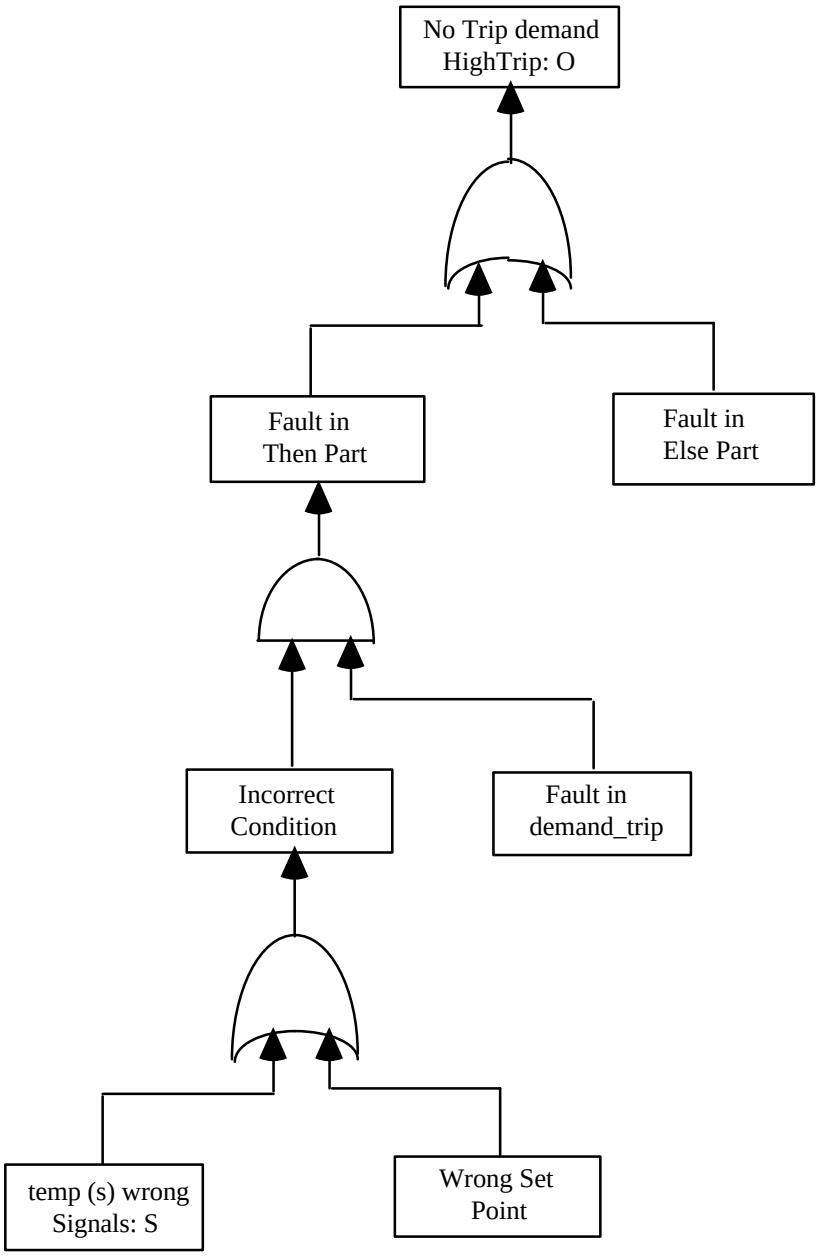

Figure 7: Fault-Tree for Hot Trip

The FPTN module Trip Algorithm is on the SCP. We now have sufficient information to discuss the change analysis.

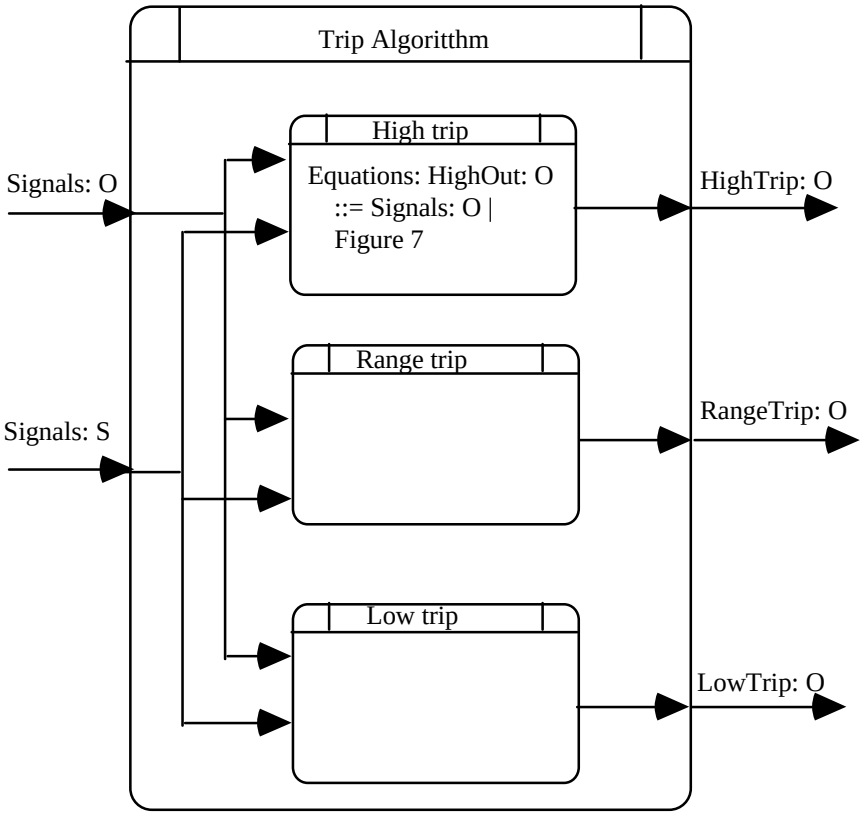

Figure 8: FPTN for Trip Algorithm

4.5 Change Analysis

We now consider the analysis of the changes identified in section 4.1, based on the information introduced above, and using the process outlined in section 3.4.

4.5.1 Decrease Range of Sensor Values

Strictly, the change is a challenge, but one which is 'discharged' immediately by analysis of the nature of the change. Thus we treat it as if it were benign for expository purposes:

1 source — TA module; the value max_spread is reduced.

2 category — benign as this can only lead to more trips, not less.

3 argument — see figure 9.

The argument why the change is benign is summarised in graphical form in figure 9. It is important to understand the context of the argument. The goal which is under challenge by the change is G1.M.N.1. The argument establishes that G1.M.N.1 still holds, even after the change, thus the analysis does not need to be propagated any

further (the rest of the case is not challenged directly; G1.M.N.1 holds so the rest of the case is not challenged indirectly).

The argument form shown is very similar to a fault tree, except that the claim is shown on the right, and the data on which the claim rests is on the left. There are three data used in the argument. First, there is an assertion that narrowing the trip range can increase the trip rate, not decrease it. This assertion could be backed up by analysis of the trip algorithm, but we assume that it will be accepted as a correct informal analysis of the program. The second assertion is that only the trip algorithm on the SCP is affected, and so this is the only element which needs to be considered. This is clearly the case as the change does not affect the flow of data between any of the modules. However this is a crucial issue: the validity of reasoning in terms of goals and dealing with 'local' changes is contingent on the accuracy of the analysis of the scope of the impact of change. The third datum asserts that the testing of the code with the previous range value can be used to justify the current version of the software, as the change has not affected the code. These assertions seem to be 'obvious' but if they were challenged they can be 'backed up' by more detailed analysis of the programs/system, e.g. along the lines discussed in section 2 for the third datum.

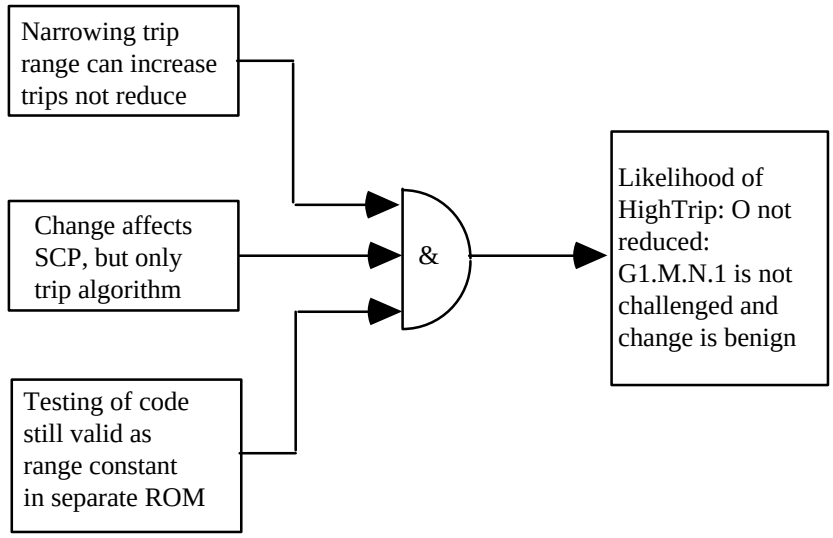

Figure 9: Argument that Narrowing Trip Range is Benign

In this case, the above argument and any supporting evidence would be the 'delta' on the safety case. This example illustrates the notion of argument template introduced in section 3.4. The general form of the argument is: the change is intrinsically benign; the scope of the impact of the change is limited to the item directly affected by the imposed change; the existing evidence for the affected item still holds — hence the claim of the safety case still holds. Many arguments justifying benign changes will be of this general form.

4.5.2 Increase Set Point

Increasing the set point is a challenge, and is analysed in the following way:

1 source — TA module; the value hot_set_point is increased.

2 category — challenge as there is a change on the SCP and it is not obvious that the immediate goal is still met as it could cause the ‘wrong set point’ event in the fault-tree of figure 7, and thus violate G1.M.N.1.

4 there are a number of possible responses, see below.

The challenge can only be met from the point of view of reactor physics, or evidence of successful operational experience. Clearly, there will have been no previous experience of successful and safe operation with temperatures between the old and new set points, so a direct appeal to history would be inadequate. Of the many possible solutions it is perhaps most likely that a modification of operational procedures enabling operation with the increased set point but more stringent monitoring would be required, before the plant could be operated with the new set point in a ‘routine’ manner. This experience might be used to alter the budgeting between G1.M.N.1 and G1.M.N.2 and to modify the justification, thus still satisfying G1.M.N. Here G1.M.N, not G1.M.N.1, would act as the ‘barrier’ in the goal structure beyond which no further challenges propagate.

5 Observations

The examples we have set out above are necessarily fragmentary, but it is our hope that we have illustrated enough of the approach we have adopted to reasoning about change to enable others to adopt the spirit of the approach on their problems.

We have analysed a number of other example changes including ones where there were consequential changes in the operating procedures. We suspect that this form of consequential change is quite common, and it makes clear that it is necessary to model the operating procedures as part of the safety case.

In carrying out the case studies, the broad structure of our approach has remained intact, although it is clear that the ideas could usefully be refined and extended. For example, the distinction between benign and challenging changes is not quite so clear cut as we once thought it was. We should perhaps distinguish three cases:

- justifiable immediately as no safety goals are threatened;
- justifiable after additional analysis;
- justifiable after consequential change.

All changes need to be justified, but the above indicates how easy (or otherwise) it will be to come up with a justification.

Some of the examples we looked at exposed conflicts between goals, especially availability (spurious trip rate) and safety goals. One of the advantages of our approach (not illustrated here) is that the goals need not be confined to safety, and we can use the goal hierarchies to point out conflicts which arise as a result of change. One of the examples we considered involving the operating procedures highlighted a conflict between availability (avoidance of spurious trips) and safety.

The examples we have described have all been retrospective analyses, and clearly the method would be of much more value if it could also be applied in development.

We can see no reason, in principle, why the method outlined here shouldn't work in development, although we would clearly also need procedures for folding the safety case 'delta' back into the main safety case, i.e. so we could connect the reasoning about change into the process of making change. We hope to have the opportunity to try out the ideas on, or in parallel with, a 'live' development in the near future, and this should enable us to put the hypothesis that our method will also apply in development to the test.

An issue which is still unclear is 'what is a small change?', or perhaps more subtly 'how do we know when the overall effect of a sequence of changes ceases to be small?'. We don't have a clear answer on this point. Our expectation is that the process we have described will break down when a change 'falls foul' of some unstated assumption or assumptions. This could occur after one change, or not occur even after several hundred changes. This is an issue which requires more study, but ultimately may rest on a judgement about the quality of the original safety case, in particular the extent to which all the salient information has been made explicit.

Finally, none of the examples we looked at required statistical treatment, although the cases were based around allowable failure rates. We do not know whether or not this is typical, however it was clear in many cases that a sub-text of the arguments presented was to show that the existing statistical data could be used unaltered in the new context. Even if this is not the typical case, there is a strong economic argument for trying to construct the 'delta' on a safety case to justify change in this manner.

6 Conclusions

We have developed a collection of taxonomies and analysis techniques which enabled us to analyse imposed changes, and to modify the system safety case in an appropriate way in response to those changes. These principles address general change control, as well as dealing with the initial problem which motivated the work — moving an application from one domain to another.

The examples we have addressed have enabled us to validate some of our principles, although not all of them could be tested on this small number of example changes we studied, e.g. the statistical ('black box') arguments were not tested. However the examples give us a measure of confidence in the overall approach adopted and it is clear that any method needs to identify the target and scope of change. We believe, therefore, that we have identified some useful principles for analysing and arguing about change, and for dealing with the movement of software from one domain to another. We hope to continue to develop and expand on these techniques as part of our overall programme of work on safety case development and management both in the University, and in co-operation with our industrial sponsors.

7 Acknowledgements

The bulk of the work reported here was supported by the HSE Nuclear Safety Research Programme, controlled by the nuclear Industry Management Committee (IMC). Tim Kelly is supported by a CASE award funded by the EPSRC and Rolls-

Royce and Associates. The SAM tool is being developed in the ASAM-II project, funded by the DTI and EPSRC, and involving BAe Airbus, BAe Military Aircraft, Lloyd's Register, Rolls-Royce Aerospace, Rolls-Royce and Associates, York Software Engineering Ltd and the University of York. Thanks go to all our colleagues in the ASAM-II project.

We are grateful to Gordon Hughes of Nuclear Electric and John Mitchell of the Nuclear Installations Inspectorate for the information and explanations which formed the basis of our case study.

8 References

[Ball89] Preparation of Fully Developed Safety Cases in Response to the NII Safety Audit, P W Ball, The Nuclear Engineer, Vol. 30, No. 2, pp34-40, 1989.

[Clarke89] Magnox Safety Review: Extending the Life of Britain's Work Horses, Nuclear Energy, Vol. 28, No. 4, pp215-220, 1989.

[Bondavalli90] Failure Classification with respect to Detection, A Bondavalli, L Simoncini, First Year Report: ESPRIT BRA Project 3092: Predictably Dependable Computing Systems, May 1990.

[Fenelon93] An Integrated Toolset for Software Safety Analysis, P Fenelon, J A McDermid, Journal of Systems and Software, Vol. 13, pp2-16, 1993.

[Hogberg94] Shutting down five reactors: reasons why and lessons learnt, L Hogberg, Nuclear Europe Worldscan, Vol. 1, No. 2, pp42-43, 1994.

[HSE92] Safety assessment principles for nuclear plants, Health and Safety Executive, 1992.

[Leveson83] Software Fault Tree Analysis, N Leveson, P R Harvey, Journal of Systems and Software, Vol. 3, pp173-181, 1983.

[McDermid94] Support for Safety Cases and Safety Arguments using SAM, J A McDermid, Reliability Engineering and System Safety, Vol. 43, No. 2, pp111-127, 1994.

[NE91] Stage 9 Submission, Dungeness 'B' Power Station, Single Channel trip System Reliability, Nuclear Electric 1991 (Private Communication).

[Pumfrey94] A Development of Hazard Analysis to Aid Software Design, D J Pumfrey, J A McDermid, In Proc. of COMPASS'94, IEEE, pp17-25, 1994.

[Queener94] Reports, Standards and Safety Guides, D S Queener, Nuclear Safety, Vol. 35, No. 2, pp339-344, 1994.

[Wilson95a] No more spineless safety cases: a structured method and comprehensive tool support, S P Wilson, J A McDermid, P Fenelon, P Kirkham, Proceedings of INEC'95: Second International Conference on Control and Instrumentation in Nuclear Installations, Institute of Nuclear Engineers, 1995.

[Wilson95b] ASAM II User Guide, S Wilson, ASAMII/UDOC/95.1, 1995. (Available from the authors.)