A Behaviour-Based Method for Fault Tree Generation

Andrew Rae; University of Queensland; Brisbane, Queensland, Australia

Peter Lindsay; University of Queensland; Brisbane, Queensland, Australia

Keywords: fault tree, hazard analysis

Abstract

This paper presents a new theory of fault trees for complex systems. The theory treats faults as behaviours, and fault-tree gates as operations on those behaviours.

Fault tree analysis is an important and widely used technique for understanding safety critical systems. Traditional fault tree methodologies typically view faults in terms of failure events or conditions of the system being analysed. This paper proposes that a wider view of faults is possible and useful, by considering faults as unusual behaviours of components and unusual component interactions. This wider view is becoming increasingly important due to the development of new technologies (such as software) which depend on sophisticated interactions between components.

We argue that it is feasible and effective to automate the generation of fault trees by describing systems using hierarchically structured models, with components, component failures and component interactions described in terms of behaviours. A detailed methodology for generating fault trees is presented. The methodology includes consideration of design faults, hardware failures, and operator errors.

Introduction

In every large undertaking, it is inevitable that a few minor problems will be overlooked. In engineering projects, these might include, amongst many others: a user interface race condition; a software buffer which can overflow; a component operating outside its design constraints; an operator who misinterprets an alarm signal; a typing error by a programmer; or a small mistake in maintenance. Unfortunately, seemingly minor problems, either alone or in combination with other seemingly minor problems, have an annoying habit of causing major disasters. A user interface glitch contributed to the Therac-25 radiation machine overdosing several cancer patients, with at least one directly caused fatality (ref. 1); a software buffer overflow was involved in the destruction of the Ariane-5 rocket (ref. 2); an O-Ring at the wrong temperature caused an explosion aboard the space shuttle Challenger (ref. 3); a typing error led to the explosion of a Dutch chemical plant (ref. 4); and a dropped lightbulb almost caused a meltdown of the Sancho Reco nuclear reactor (ref. 5). In other words, accidents frequently occur when unforeseen system behaviours arise from seemingly minor component failures and design errors. This phenomenon is referred to as “emergent hazards” or “normal accidents” (ref. 5).

In order for a system to be safe, it is necessary to perform a detailed analysis of the faults which may be present in the system, and to have a clear understanding of the ways in which these faults, either individually or in combination, may affect the behaviour of the system.

This paper deals with a particular technique for safety analysis, Fault Tree Analysis (ref. 6). Fault tree analysis is widely used by industry for both reliability and safety assessment, and has generated a considerable volume of academic literature (ref. 7).

Traditional fault trees imply the notion of a strict chain of fault events, where each fault event is caused by an immediately preceding event or combination of events. Under such a notion, the term “fault" is synonymous with the term “failure". This view of faults is limiting - whilst it may be valid for random failures, it does not apply to systematic faults. In particular, faults due to design or installation error can only be discussed indirectly under such models, by describing how they manifest themselves in response to input events.

This is replaced here by viewing systems as a hierarchy of behaviours. We define behaviour here as “a set or series of acts regarded as a unified whole”.

The behaviour of a subsystem is defined by the behaviour of its components. Fault behaviour propagates from faulty single component behaviour, to faulty subsystem behaviour, and eventually to faulty system behaviour, through composition and abstraction. Unlike many fault tree approaches, where the top event is caused by and occurs after lower level events, system fault behaviour is an abstract view of many component behaviours, some or all of which may be fault behaviours.

This switch from events to behaviours entails a redevelopment of the theory of fault trees. This involves a need for ways to represent systems, to analyse their behaviour, and to express the relationships between component behaviours in the form of a fault tree.

Section 2 introduces fault trees, and discusses previous attempts to automate fault tree generation. Sections 3 and 4 present a new formalisation of fault trees based on behavioural semantics. Section 4 shows how this new view of fault trees facilitates the automatic generation of fault trees from behavioural models.

Fault Trees

Overview of Fault Trees: Fault Tree Analysis (FTA) was first developed in 1961 by H. A. Watson of Bell Telephone Laboratories as part of the safety program for the Minuteman Launch Control System. The technique was further refined by teams at Bell Telephone Laboratories and at Boeing Company (refs. 8, 9). After several presentations at the 1965 Safety Symposium it gained widespread acceptance in the aerospace and nuclear industries, and later also in the chemical process industry (ref. 10). Fault Tree Analysis is now one of the principal methods of system safety analysis across all industries (ref. 11). Fault Tree Analysis is a form of deductive failure analysis. It focuses on a particular undesirable event, termed the top event, and provides a method for successively determining the possible causes of this event. The fault tree itself is a graphical model which depicts the logical relationships between the events which contribute to the top event occurring (ref. 12).

Fault Tree Notation: Figure 1 shows a simple electric circuit, with a battery, a generator, two light bulbs and a switch. Figure 2 shows an example fault tree for this circuit, for the “hazard" of no light provided by the circuit. Note that the trees use logic gates to show, in a graphical form, how faults contribute to the hazard at the top of the tree. An AND-gate indicates that two or more faults must both occur immediately below the gate in order for the fault above the gate to occur. An OR-gate indicates that any one event at the input of the gate will cause the fault above the gate to occur.

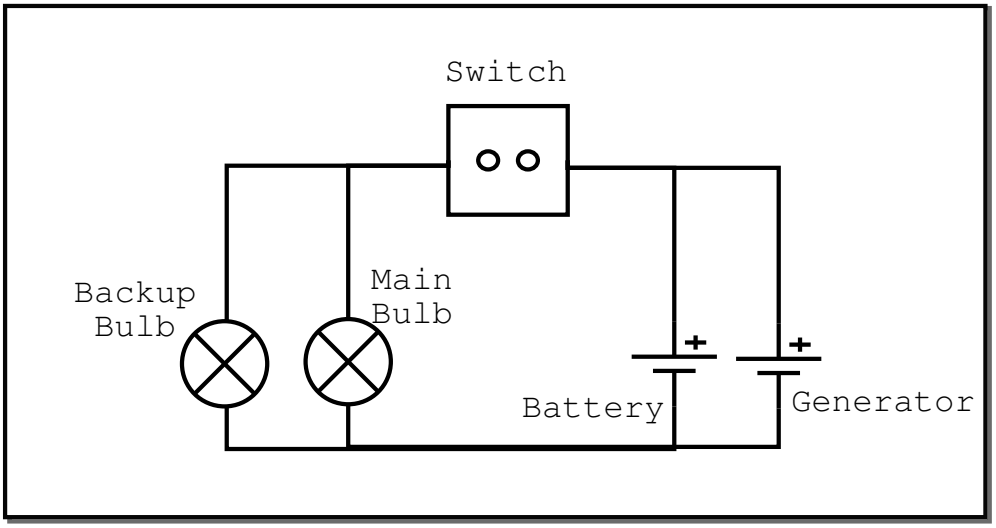

Figure 1 – A Simple Circuit

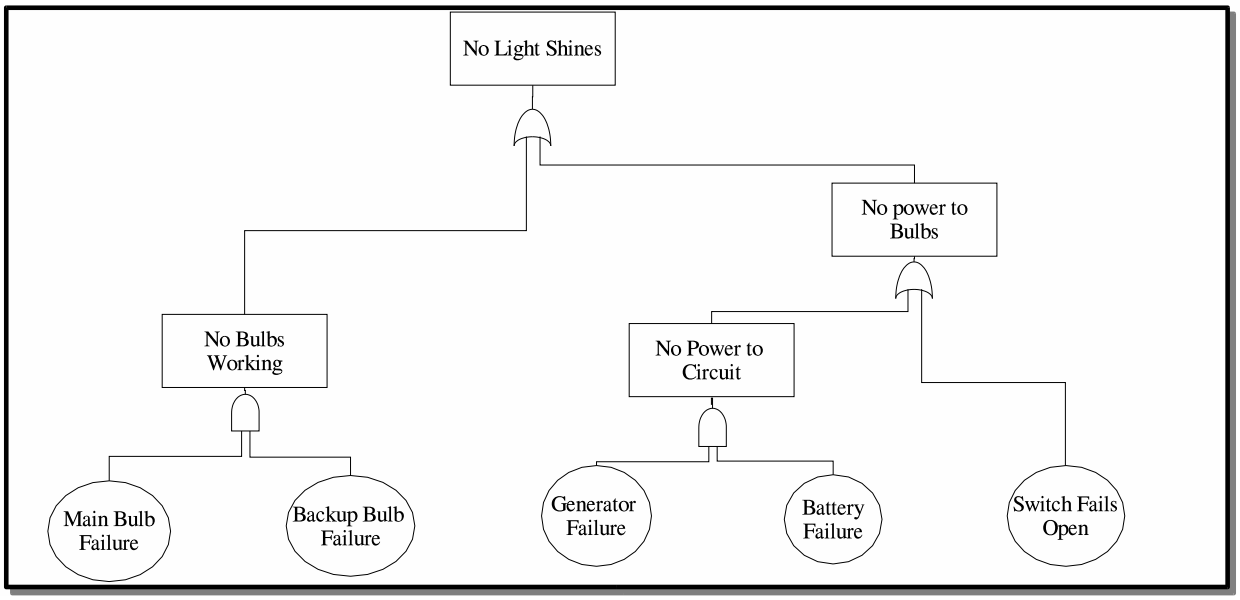

Figure 2 – Example Fault Tree for a Simple Circuit

Fault Tree Generation: The earliest attempts at fault tree generation were made by Fussell (ref. 13), and Powers, Tompkins and Lapp (ref. 14). These techniques generated fault trees by starting at a particular point in a network, and backtracking by considering all of the inputs to that point. This process "unfolds" the network into a tree.

Subsequent research by Salem (ref. 15), Reina (ref. 16), Andow and Lees (ref. 17), Papadopoulos (ref. 18) and others has developed on this fundamental idea, adding sophistication without departing from the basic notion of backtracking from the hazard to the leaf events. The fault trees generated by these methods are quite different from those which would be produced by a human analyst. With the possible exceptions of HiPHOPS (ref. 18), abstraction is not included in either the system model or the resultant fault tree. This is in direct contrast to traditional human-drawn fault trees, where the top section of the tree almost invariably deals with an abstract description of the ways in which the hazard can occur, whilst the lower branches of the tree describe the concrete mechanics of failure.

There is also a strong temporal factor in the placement of events in the tree. Events which occur just before the hazard tend to be placed near the top of the tree, and events which occur some time before the hazard are near the bottom of the tree. The result of this phenomenon is a tree which has a few long branches, with many very short side branches. Such a tree creates the misleading impression that events near the top of the tree are more important than events near the bottom, even though they both may be single points of failure. There are further problems with adapting these techniques to a new application domain. Each of the techniques makes assumptions about the way in which the system behaves which are not often valid for complex, dynamic and interactive systems.

Whilst techniques such as HiP-HOPS (ref. 18) show progress toward addressing some of these problems through the use of hierarchical modelling, this thesis argues that modern computer-controlled systems require a fundamentally new approach to fault tree generation.

Faults as Behaviours

In safety and reliability literature, a distinction is often made between so-called "random failures" and "systematic failures". A random failure is one which occurs sometime in the life of a component according to a probability density function. A systematic failure is one which will always occur under particular circumstances. The stereotypical random failure is a hardware component which undergoes physical degradation – for example, a lightbulb which burns out. The stereotypical systematic failure is a software error which manifests itself under certain input conditions - for example, the early Pentium processor floating-point module, which produced errors only when dividing certain large numbers.

At times, this distinction can be quite artificial. A systematic failure manifests itself according to the system inputs, which may follow a probability density function. The probability density function of a hardware component may be shaped by its operating environment, which is determined by the system design.

A wide section of the system safety literature, e.g. (refs 3, 11) indicates that systematic failures are of increasing concern in system safety, but are not necessarily given the same attention as random failures. There are also difficulties with incorporating both types of failure into the same safety model.

The behavioural view of faults does not distinguish between systematic and random failures. For clarity, the two types of fault will be referred to as “design errors" and “hardware failures". Both are faults which replace the ideal behaviour of a component. For example, previously a blown fuse would be modelled as a random failure which occurs at some time during the life of the fuse. Under a behaviour model, a fuse hardware failure is a fault which replaces the normal behaviour of a fuse. A fuse fault behaves like an ideal fuse, but may undertake an internal transition (a “failure"), after which no current may pass through the fuse.

A design error is modelled in a similar way. A software fault is a behaviour which replaces the normal behaviour of the software. The fault behaviour may have extra transitions, missing transitions, or transitions which lead to different states from those of the normal behaviour.

System Model:

Use of Process Algebra: The method presented in this paper models the behaviour of components using process algebras. A process algebra is a formal approach to specifying concurrent systems in terms of nondeterministic sequential processes that synchronise their behaviours on shared events. Whilst different process algebras have different notations and features, their important property for the task at hand is their ability to concisely represent a labelled transition system in a readable fashion. Using a process algebra, it is possible to fully specify the events which a component can participate in, and the circumstances under which it will engage in, or refuse to engage in, those events. Process algebra models can be compared to each other, and can be analysed by a model checker to determine if they satisfy certain specified properties. An important feature of process algebra is the ability to use hiding and non-determinism to conceal detail. This allows the behaviour of a system to be analysed at the most appropriate level of abstraction for the task at hand. One specific process algebra, Communicating Sequential Processes (CSP) (ref. 19) will be used for the examples in this paper.

In our approach, systems are described as a series of models, with each successive model providing more detail. The complete set of such system models is called a *development*. This allows the behaviour of a system, and thus the faults of a system, to be examined at progressive levels of detail, resulting in a fault tree which incorporates abstract as well as concrete faults. This gradual revelation of detail as further versions of the system are developed parallels the structure of fault trees. A well constructed fault tree traces system failures down into the design, with high level nodes in the tree corresponding to abstract faults, and nodes further down the tree corresponding to detailed, localised faults.

Hierarchical Models: A hierarchical model is one where subsystems are described as compositions of smaller elements. These smaller elements are called *subordinates*. For example, a car may described as having the subordinates `engine', `chassis' and `body'. An engine may be described as having the subordinates `fuel system', `combustion system’ and `power train'. Note that subordinates must be self contained. In order to describe the internal behaviour of an engine (as opposed to its interactions with other subordinates), it should not be necessary to refer to the chassis or the body.

In CSP, hierarchical modelling is performed using process composition. A process may be defined as a composition of subordinate processes. Typically, two separate yet interacting components will be composed using the parallel operator ([||]). This operator allows each component to act independently, unless they share an event. In order to perform a shared event, both components must be ready to undertake the event. The shared event is then executed simultaneously by both processes. Two separate, non-interacting components will be composed using the interleaving operator (|||). This allows each component to act totally independently. Events common to the alphabet of each may be undertaken by either component, but cannot be undertaken by both simultaneously.

The smallest possible subordinates are called *components*. It is required by this methodology that only components may refer directly to events. Larger subsystems refer to events indirectly by forming compositions of processes.

In summary: components are formed from events; subsystems are formed by composing components; system models are formed by composing subsystems; and a development is a series of system models.

Fault Modelling:

Normal and Fault Behaviours: A fault behaviour (or *fault* for short) is defined here as any behaviour of a component other than the normal behaviour. In this methodology, and in fault tree analysis more generally, a distinction is made between “normal" and “fault" behaviour of a component. The normal behaviour is always defined in such a way that the hazard cannot result from all components acting normally. Typically, a *system failure* will result from some components acting normally, and one or more components with faulty behaviour.

Multiple Faults: For each component, multiple fault behaviours may be possible, each representing a different, independent departure from the normal behaviour. For example, a simple switch might have: a normal behaviour; a fault behaviour where it can become stuck on; a fault behaviour where it can become stuck off; a fault representing a manufacturing error where it becomes broken after between ten and fifteen uses; and a “reversed’’ behaviour whereby it is on when it is supposed to be off, and vice-versa.

Since each component can have only one behaviour at a time, the notion of two faults occurring simultaneously only makes sense when those faults represent behaviours of different components. The methodology can of course be used to investigate multiple simultaneous failures of a subsystem by modelling the internals of that subsystem as more than one component.

Distinguishing between Faults and Failures: Modelling faults as behaviours also implies a strong distinction between faults and failures. A failure is an event, here, after which the component’s behaviour changes, whereas a fault is a description of the behaviour of the component throughout its life. For example, consider a light bulb. The normal behaviour of the light bulb might be “Light bulb always gives light when switched on". “Light bulb breaks" is a failure, not a fault. “Light bulb is broken" is not a valid fault either, unless the analyst is considering the possibility of light bulbs being installed after they are already broken. A more typical fault would be expressed as a behaviour: “Light bulb originally gives light when switched on, then light bulb breaks, after which light bulb does not give light".

Note that failures are not always immediately obvious: a component’s behaviour may appear to be normal for some time after a failure, and only diverge from normal behaviour at some later stage.

Examples of Faults:

Design Error: A design error is represented by nondeterministic choice. This represents a component which will sometimes appear to be behaving correctly, yet always has the potential to take an incorrect action. In practice, design errors become manifest through specific triggers, such as a particular set of environmental conditions or inputs. Unless the precise causes of the aberrant behaviour can be modelled, the component will appear to be behaving non-deterministically.

Hardware Failures: A hardware failure is a failure that occurs due to physical deterioration of the component. The associated fault is a behaviour which is initially the same as the normal behaviour, but which diverges from normal behaviour at some point in time. In CSP, this is represented using the “catastrophic interrupt operator" (!!). P = A!!t -> B indicates that process P begins as A, but at any time can change to B after an internal event, t.

Human Errors: A human user may be modelled using non-determinism, catastrophic interrupts, or simply a completely incorrect process, depending on the purpose of the model. A typical user model will involve an operator who behaves perfectly most of the time, but makes occasional, seemingly random errors. These are termed slips and lapses for skill-based and rule-based activities respectively (ref. 20). A slip or a lapse can be modelled by adding non-deterministic choices.

Another type of user model represents a user who evolves between behaviours, perhaps due to fatigue, injury, adoption of bad habits, or even increased experience. Such a fault is similar to a hardware fault - the user appears to be behaving correctly, but at some point in time diverges from normal behaviour. An example of this type of user is an operator on an assembly line, who repeatedly puts a component in their machine, activates a safety lock, drills a hole, and removes the component. Eventually, the user may decide that activating the safety lock is unimportant, and will cease performing the action.

The third type of user is one who makes persistent mistakes, resulting from, for example, incorrect or inadequate training, or poor user interface design. Such a user will never behave correctly.

Environmental Hazards and other Co-effectors: Some faulty behaviours only become realised as hazards under particular environmental conditions (coeffectors). For example, a faulty communication protocol may appear to operate correctly, but may fail when the channel becomes noisy. In such cases, it is useful to model the environment as well as the system. The “normal" behaviour of the environment is one which does not contribute to the hazard. Environmental behaviours which may act as co-effectors are modelled as “faulty" behaviours of the environment, and will appear on the fault tree.

Fault Trees Using Behaviours

Recall that in our proposed approach, levels in fault trees correspond to levels of design detail. These ideas are made more precise below, together with an interpretation of AND-gates and OR-gates.

General Structure of a Fault Tree: A fault tree consists of logic gates and behaviour nodes. Each node in the tree is connected to zero or one input gates and zero or one output gates. Each gate in the tree has one or more inputs and a single output. A gate represents a relationship between its input node(s) and its output node. There are three types of behaviour nodes present in the fault tree: the top event node, intermediate fault nodes, and basic fault nodes. The only difference between intermediate fault nodes and basic fault nodes is that intermediate fault nodes have an input gate, whereas basic fault nodes are leaves of the tree. Henceforth both intermediate fault nodes and basic fault nodes will be simply called fault nodes.

Fault nodes represent behaviours of the entire system. These are composed of behaviours of each of the individual components. The top event node represents the hazard under investigation. This hazard is specified as a temporal logic proposition.

Comparing Faults in Hierarchical Developments: In the previous section, it was explained that a development consists of a series of system models, with each successive model describing the system in greater detail. The faults associated with later models will thus be more detailed than those of earlier models. It is necessary for the methodology to provide a way for faults in different system models to be compared. In particular, we wish to know whether an abstract fault represents essentially the same behaviour as a detailed fault or combination of faults.

More precisely, the question which must be asked is whether there is a preorder or refinement relation between the abstract fault and the detailed faults.

The notion of refinement of faults here differs from the way it is usually treated in the refinement literature. Typical refinement seeks to preserve properties related to safety or correctness, but accepts refined behaviours which are more deterministic. In this case, it is instead necessary for the refinement relation to preserve those properties of a behaviour which make it hazardous.

Finding the correct relation to use is not trivial. Informally, it is necessary for the “lifted" or more abstract fault to be equivalent in some way to the refined fault. Equivalence, though, is a somewhat vague concept, depending on the way in which systems are observed. Glabbeek (ref. 21) examines 155 separate equivalences, and notes that even this is not an exhaustive selection.

There is probably no equivalence relation which is ideal for every given system and hazard. Accordingly, this paper does not mandate a particular relation as an integral part of the methodology. In fact, as with the choice of modelling

language, the methodology is independent of the choice of equivalence relation. As a practical issue, the equivalence relations available to the user will be restricted by the choice of process algebra and the tool support available.

AND-Gates: AND-gates represent combinations of faults. Each input of an AND-gate will be a behaviour of the system exhibiting a single fault. That is, the behaviour is composed of behaviours of individual subsystems, where one subsystem is exhibiting a fault behaviour, and all other subsystems are exhibiting their normal behaviour. The output of an AND-gate is a behaviour of the system exhibiting multiple component faults. That is, the behaviour is composed of behaviours of individual components, where more than one component is exhibiting a fault behaviour, and all other components are exhibiting their normal behaviour. All of the input faults, and no other faults, are represented.

The input and the output nodes of an AND-gate are behaviours at the same level of abstraction. They are composed of the same components, and have the same alphabets. The only difference is that each of the inputs has only one component exhibiting a fault behaviour, whereas the output has as many components exhibiting a fault behaviour as it has inputs. Since each component can only exhibit a single fault behaviour at a time, there will be a constraint that prevents two inputs of the same AND-gate from representing different failure modes of the same component.

OR-Gates: OR-gates represent fault abstraction. Intuitively, OR-gates “lift" faults from one level of abstraction to another. In the same way that each system model is more detailed than the previous system model, behaviours at the inputs of an OR-gate are more detailed versions of the behaviour at the output. Note that because OR-gates link behaviours at different levels of abstraction, the two behaviours will not usually have the same alphabet. The output is equivalent to the input once any extra input events have been hidden. The meaning of “equivalent" for ORgates linking fault nodes refers to a testable equivalence relation.

Since the top event node contains a temporal logic property, whilst fault nodes contain processes, it is evident that the gate connected to the input of the top event will have different semantics to gates which connect fault nodes to fault nodes. The top event is a specification, and is therefore more abstract than the fault nodes near the top of the tree. Hence, the input to the top event cannot be an AND-gate, as AND-gates connect behaviours at the same level of abstraction. The input to the top event is thus always an OR-gate. The inputs to this OR-gate are behaviours which satisfy the top event.

Automating Fault Tree Generation

Based on the methodology above, we have produced a prototype tool, *Eucalypt*, for automating fault tree generation. Eucalypt makes use of the Concurrency Workbench for the New Century (ref. 22). In our approach, the fault tree analysis itself is an algorithm which translates a system model into a fault tree, as shown in Figure 3. Each development step maps directly into a portion of the fault tree. First, the complete system model is produced, with normal and fault behaviours specified for each component, and the hazard formulated.

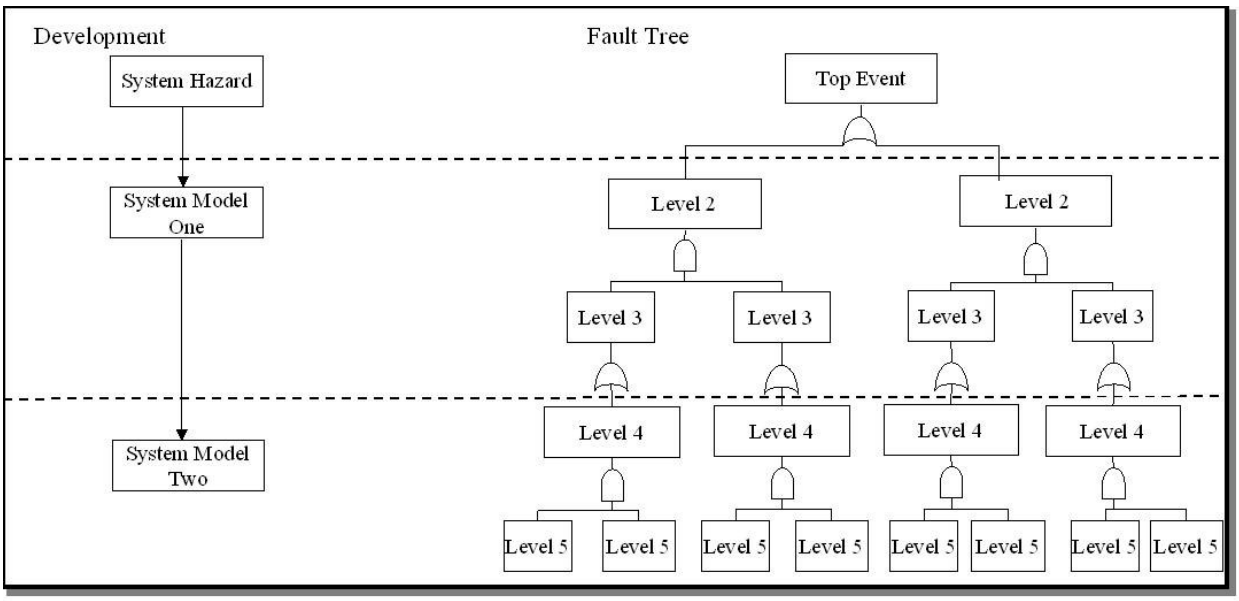

Figure 3 – Relationship between a System Model and a Fault Tree

Next, the most abstract system model is considered. Each possible fault behaviour of the system is compared to the top event. Where a behaviour satisfies the top event, it is linked to the top event via the top OR-gate. At this stage, each of these behaviours may incorporate multiple faults. The next layer of fault tree divides these into behaviours containing single component faults, linked to the composite faults via AND-gates.

Each successive development step is then analysed, by considering all of its possible behaviours. Behaviours which cannot cause the top event are discarded. For each remaining behaviour, an equivalent behaviour is found in the previous layer of the fault tree, and the two nodes are linked by an OR-gate. AND-gates are then produced to decompose the fault into individual subordinate component faults. If a behaviour satisfies the top event, but is not equivalent to any behaviour of the previous development step, this is evidence that the preceding development step did not completely represent all of the relevant faulty behaviours. The system model is then adjusted, and the analysis restarted.

Once the most concrete development step has been analysed, the fault tree generation process is complete.
Gates which have only one input may be trimmed from the tree for conciseness, or retained for clarity.

Figure 3 shows the relationship between a hierarchical development and the resulting fault tree. The top node of the tree corresponds to the hazard. For each system model in the development, there are two rows of behaviour nodes, linked alternately with OR-gates and AND-gates. An OR-gate links behaviours from different models, whilst an AND-gate groups faults from within a model into a single behaviour.

Conclusion

The underlying hypothesis of this paper is that in reasoning about the behaviour of complex systems in the presence of failures, it is possible and useful to consider faults as behaviours of the system components, rather than simply as discrete events. The paper explores this idea as it relates to modelling behaviours, comparing faults, and structuring fault trees. This avoids one of the main recurring problems with fault tree analysis of software, and design failures more generally, namely that a systematic failure is not a discrete event.

This approach also allows exploration of interesting behaviour, rather than simply examining combinations of failure events. Most early fault tree generation techniques were applied specifically to chemical plants or electric circuits. Whilst the methodology presented here is not unique in being more generally applicable, there are some specific types of problems to which it is better suited than other existing techniques. These include:

- Systems involving complex interaction of components, including features such as synchronisation and resource sharing;
- systems involving cycles and command loops, where an indefinite number of events may occur between the first failure and the realisation of a hazard;
- systems where design errors are a significant cause of faults;
- systems where failures can cause qualitative changes in the behaviour of a component, rather than total cessation of function or quantitative aberrations.

The primary application domain for such systems is real-time embedded control systems. This domain includes industrial control systems, weapon systems, and many transport systems.

The methodology has been implemented as a prototype tool called “Eucalypt". Eucalypt is the first automatic fault tree generator to make use of model checking. Most existing methodologies rely on a backtracking algorithm from a hazard location or event. Backtracking has considerable difficulty with loops and recursion. More importantly, backtracking techniques require that hazards be specified as incorrect values at particular points, whereas Eucalypt can express hazards more generally as abstract properties of the entire system. Eucalypt has been applied to the analysis of a naval weapon system, a robot-arm safety cell, and a railway signalling system.

References

1. Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. IEEE Computer, 26(7) pp 18 -- 41, July 1993.

2. European Space Agency. Ariane 5: Flight 501 failure. Report by the enquiry board. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html

3. Presidential Commission on the Space Shuttle Challenger Accident. Report to the president. http://history.nasa.gov/rogersrep/genindex.htm, June 1997.

4. Peter G. Neumann. Computer Related Risks. ACM Press, New York, 1995.

5. Charles Perrow. Normal Accidents: Living with High Risk Technologies. Basic Books, New York, 1984.

6. N.H. Roberts, W.E. Vesely, D.F. Haasl, and F.F.Goldberg. Fault Tree Handbook. Systems and Reliability Research Office of U.S. Nuclear Regulatory Commission, January 1981.

7. W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie. Fault tree analysis, methods and applications -- a review. IEEE transactions on reliability, R-34(3) pp 194-203, August 1985.

8. David F. Haasl. Advanced concepts in fault tree analysis. In System Safety Symposium. University of Washington, 1965.

9. J. L. Recht. System safety analysis: The fault Tree. National Safety News, April 1966.

10. Jerry B. Fussell, Gary J. Powers, and R. G. Bennetts. Fault trees -- a state of the art discussion. IEEE Transactions on Reliability, Volume R-23(1):51 -- 55, April 1974.

11. Nancy G. Leveson. SAFEWARE: System Safety and Computers. Addison-Wesley, Reading, Massachusetts, 1995.

12. R. E. Barlow and H. E. Lambert. Introduction to fault tree analysis. In Richard E. Barlow, Jerry B. Fussell, and Nozer Singpurwalla, editors, Reliability and Fault Tree Analysis. Society for Industrial and Applied Mathematics, 1975.

13. J. B. Fussell, W. B. Vesely, and J. D. Clement. Elements of fault tree construction - a new approach. Transactions of the American Nuclear Society, Volume 15(2), November 1972.

14. Gary J. Powers, Frederick C. Tompkins, and Steven A. Lapp. A safety simulation language for chemical processes: A procedure for fault tree synthesis. In Richard E. Barlow, Jerry B. Fussell, and Nozer Singpurwalla, editors, Reliability and Fault Tree Analysis. Society for Industrial and Applied Mathematics, 1975.

15. S. L. Salem, G. E. Apostolakis, and D. Okrent. A new methodology for the computer-aided construction of fault trees. Annals of Nuclear Energy, 4(9-10), 1977.

16. Giuseppe Reina and Giuseppe Squellati. L.A.M. technique: Systematic generation of logical structures. In G. Apostolakis, S. Garribba, and G. Volta, editors, Proceedings of the NATO Advanced Study Institute on Synthesis and Analysis Methods for Safety and Reliability Studies 1978, New York, 1980. Plenum Press.

17. P.K. Andow. Difficulties in fault-tree synthesis for process plant. IEEE Transactions on Reliability, R-29(1):2 -- 8, April 1980.

18. Yiannis Papadopoulos, John McDermid, Ralph Sasse, and Gunter Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Reliability Engineering and System Safety, 71(3):229{247, 2001.

19. C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

20. J. Rasmussen. Information Processing and Human-Machine Interaction. North-Holland, New York, 1986.

21. Rob J. van Glabbeek. The linear time – branching time spectrum II. In International Conference on Concurrency Theory, pages 66 -- 81, 1993.

22. Rance Cleaveland, Tam Li and Steve Sims. The Concurrency Workbench of the New Century Users Manual, SUNY at Stony Brook, 2000.

Biography

A. J. Rae, Information Technology and Electrical Engineering, University of Queensland, Brisbane 4072, Australia, telephone -- +61 7 3365 1651, facsimile -- +61 7 3365 4999, email – arae@itee.uq.edu.au

Andrew is a research officer at the University of Queensland. He has previously been employed at the Laboratory for Computer Science, MIT, where he researched the safety of radiation therapy machines, and by the Australian Department of Defence, where he researched the safety of naval platforms.

Professor P. A. Lindsay, Information Technology and Electrical Engineering, University of Queensland, Brisbane 4072, Australia, telephone -- +61 7 3365 2005, fax -- +61 7 3365 4999, email – pal@itee.uq.edu.au

Peter joined the University of Queensland in 1991 after holding academic and research positions at the University of New South Wales, the University of Manchester and the University of Illinois at Urbana-Champaign. He has more than sixteen years experience in formal aspects of systems and software engineering. He is co-author of two books on formal specification and verification of software systems, and over 50 refereed papers. In recent years he has contributed to a number of safety-critical applications in the areas of defence, aerospace and transport.