Fault Tree Analysis with Bayesian Belief Networks for Safety-Critical Software

Chris Hobbs, Kernel Developer
QNX Software Systems
chobbs@qnx.com
Abstract

The flexibility of Bayesian Belief Networks makes them particularly suitable for presenting a quantified safety case incorporating hard and soft evidence. This paper describes their application to one component of an overall safety assessment of the QNX Neutrino microkernel.

Introduction

When implementing a safety- or mission-critical application, it is essential to be able to present the argument that it meets the system requirements of availability and reliability; that is, how often a system returns a response, and how often the returned response is correct. Typically this argument is supported by a combination of:

- hard and soft evidence
- \textit{a priori} and \textit{a posteriori} evidence

Bayesian Belief Networks (BBNs) provide a tool for incorporating these types of evidence and providing quantitative results.

A full safety case contains evidence from many sources: the development team size, the skills of the developers, the development practices applied, etc. This paper describes one sub-network of a full safety case: that describing the failure analysis of the system. For a broader view of the use of BBNs for presenting safety cases, see papers such as Bouissou \textit{et al} listed under Further Reading.

The remainder of this whitepaper:

- describes the difference between a Fault Tree Analysis and a Failure Mode, Effects and Criticality Analysis. It justifies the former as more appropriate for a system such as the QNX Neutrino microkernel — section 2.
- outlines the main characteristics of a Fault Tree and the analysis that can be performed on it — section 3.
- describes the concept of a BBN and its advantages when building a Fault Tree — section 4.
- gives an insight into the application of a BBN to the QNX Neutrino microkernel — section 5.

\footnote{Evidence argued from cause to effect, and from effect to cause.}
Failure Analysis

Two techniques are commonly employed for assessing the risks associated with the use of systems in safety- or mission-critical applications: Failure Mode, Effects and Criticality Analysis (FMECA) and Fault Tree Analysis (FTA). These techniques differ in that FMECA is an inductive analysis of system failure, starting with the presumed failure of a component and analysing its effect on the stability of the system: “What would happen if valve A sticks open?” In contrast, FTA is a deductive analysis, starting with potential or actual failures and deducing what might have caused them: “How could a deadlock in the application occur?”

Either of these techniques can be greatly assisted by the use of a Bayesian Belief Network, but this paper deals explicitly with an FTA carried out by QNX on its Neutrino microkernel. In this case a FTA could provide more focused information than a FMECA because the Neutrino microkernel has a long history and extensive field usage. Failure modes are therefore known.

The Fault Tree

Fault trees encapsulate the concept that the failure of a (sub)system can be caused by the failure of lower-level (sub)systems. Typical types of combination are:

- X fails if both leaf Y and leaf Z fail (Y and Z may be identical units, either of which can carry the system on its own if required).
- X fails if either leaf Y or leaf Z (or both) fails (Y and Z may be units that act serially, and the failure of either breaks the chain).
- X fails if any two of leaves Y, Z and T fail.

The example in Figure 1 illustrates the basic concepts of a fault tree:

- The system will fail if both failure 3 and failure 2 occur.
- Failure 2 will occur if either a failure at leaf C or a failure at leaf D (or both) occurs.
- Failure 3 will occur if either failure 1 or a failure at leaf E (or both) occurs.
- Failure 1 will occur if both failure at leaf A and a failure at leaf B occur.

Figure 1 — A very simple fault tree. Failures are numbered (1, 2, etc.), while letters identify leaves (A, B, etc.).
Once the tree is drawn, the minimal cut sets can be identified. A cut set is a set of leaves where failure of every leaf would cause the system to fail. In Figure 1, a trivial cut set is \( \{ A, B, C, D, E \} \) because, if these leaves all failed, the system would fail. A minimal cut set is a cut set where, if any element were to be removed, it would no longer be a cut set.

For example, in Figure 1 the minimal cut sets are \( \{ E, C \} \), \( \{ E, D \} \), \( \{ A, B, C \} \), \( \{ A, B, D \} \). The failures listed in any one of these sets will cause a system failure. Removing any element from one of these sets would change that set so that a failure of all leaves in the set would not result in a system failure. Figure 1 is, of course, trivially simple and, for realistic trees, computer programs are needed to identify minimal cut sets.

**The Bayesian Belief Network**

In theory, given a Fault Tree, failure rates and failure distributions can be associated with each leaf in a BNN and a computer program can then consolidate this information into a failure rate and failure distribution for an entire system. In practice, however, quantifying the leaf failures for an entire system is difficult or, in many cases, impossible.

*Figure 2 — A BBN for Figure 1. Without additional information, all components are assumed to have a 50/50 chance of having failed.*
There are several reasons why it may not be possible to determine the failure patterns accurately, but one particular issue is that, in many instances, better failure information may be available for an entire subsystem than for some (or any) of its components. For instance, in Figure 1 failure information may be available for failure 3 and failure at leaf E, but this information may not be available for failures at leaf A and leaf B. In this case, the tree could be transformed to make failure 3 into a leaf without substructure. However, this change would lose information and make the reuse of items in different subtrees impossible, placing significant constraints on the way in which the tree can be created.

Bayesian networks allow for this difference in failure information by accepting evidence for the failure rate of any node, then using Bayes' theorem (see appendix) to calculate the a posteriori probabilities of the failure rates of the sub-elements: reasoning from effect to cause.

Using one of the graphical tools available for entering and analysing BBNs, it is easy to translate the structure of Figure 1 into a BBN, as illustrated in Figure 2. This BBN shows the BBN before any values for failure rates have been entered. In the figure, True is used to mean that the component or (sub)system has failed within the first $N$ hours of operation (where $N$ is...
chosen appropriately: e.g., $10^9$). Since we have provided the tool with no better values for the chances of leaf (component) failure, we can see from Figure 2 that the tool has assumed that there is a 50% probability of the component having failed. The tool then wraps up the failure values to produce a probability of failure for the whole system$^2$.

Making the assumption that we have "soft" values for the failure probability of subsystem 3, we can add those values to the model. As shown in Figure 3, the tool then recalculates not only the new value for the entire system but also applied Bayes' Theorem to calculate the values for subsystem 1 and components A, B and E. Note that the tool still has no better values for components C and D than 50%.

**Application to the QNX Neutrino Microkernel**

While the trivial example shown in Figures 2 and 3 is useful to illustrate the technique of using a BBN to create a Fault Tree, this example is much smaller than the Fault Tree for a microkernel such as QNX Neutrino. Creating a realistic Fault Tree for QNX Neutrino was carried out in several steps:

1. We clearly identified the ways in which QNX Neutrino can fail.

   Once initialisation is complete, QNX Neutrino is an event-handling system: it remains quiescent until an event (interrupt, exception or kernel call) occurs and then handles the event in an expeditious manner before returning to its quiescent state. Fundamentally, there are three ways in which such an event handler can fail:

   a. It can fail to respond to the event. This category includes silently ignoring the event or crashing before completing the event handling.

   b. It can respond to the event incorrectly, for example, scheduling a thread to run when a higher priority thread is also ready would fall into this category.

   c. It can respond correctly but corrupt its own internal state so that a subsequent event could potentially be handled incorrectly. Typically, such a corruption would lead to a subsequent failure to respond or incorrect response, but the failure has, in fact, occurred at the time the state was corrupted.

2. Guided by product history over the period from 2002 to 2009, we created a Fault Tree of conditions that, in practice (rather than theoretically) contribute to each failure type.

3. We expressed the Fault Tree as a Bayesian network so that we could incorporate soft evidence about field failure rates, and calculate the resulting post-probabilities.

---

$^2$ Figures 2 and 3 were drawn with the AgenaRisk tool. This tool uses Bayesian networks to model, analyse and predict risks. See http://www.agenarisk.com for more information.
4. We combined reports of failures in the field with field usage figures to estimate the failure rates to be used in the Fault Tree.

Once the model was complete, we carried out a sensitivity analysis to find the values to which the final result was most sensitive. We then refined these values and repeated the calculation.

The result of this analysis was an estimate, based on justifiable assumptions, of the level of availability of the QNX Neutrino microkernel that could be compared, for example, with the requirements of Safety Integrity Levels 1, 2, 3 and 4 of IEC61508.

Statistical Analysis

The methodology described above rests on the applicability of handling software failures statistically. It has been argued that this treatment is inappropriate because, unlike hardware, software does not “wear out”: all failures are design failures, and the software life cycle does not, therefore, follow the “bathtub curve”.

The arguments below provide brief justification for treating software faults statistically.

Bathtub curve

Software failure rates do in fact follow the conventional bathtub curve. Everyone is familiar with the high failure rate of software when it is first released and unanticipated usage patterns uncover latent faults. Once these failures have been fixed, the software settles down for a period of time until changes in it (patches, enhancements, etc.) and in its environment (operating system changes, faster processors, etc.) cause the failure rate to start rising again. This failure-stability-failure pattern is very much like the conventional bathtub curve.

Theoretical foundations

Criticisms of the theoretical underpinnings of the statistical model can be answered by arguing, as do Littlewood et al (see Further Reading below), that the random nature of demands provides a genuine statistical element to software failures. That is, if the complete, multi-dimensional input space\(^3\) (or “demand space”) were known, then the sequence of program invocations could be visualised as a “walk” through that multi-dimensional space. Some walks lead to faults (Heisenbugs\(^4\)) invoking errors and thereby creating failures. The very same point in the input space could be reached as part of a different trajectory and, because of the nature of Heisenbugs, not cause a failure. Since the walks that consumers make through the input space are unknown and, to all intents and purposes are random, the sequence of failures is random and can be analysed statistically. This is particularly true of a

---

\(^3\) The space with dimensions defined by the input variables.

\(^4\) A Heisenbug is a bug that appears because of some complex interaction (typically timing interaction) and is normally non-reproducible. A Bohrbug is a bug that causes a fault every time the program receives the same input. Bohrbugs tend to be fixed before product shipment; Heisenbugs are sometimes never fixed.
product like the QNX Neutrino microkernel in an symmetric multiprocessing (SMP) environment.

**Conclusion**

Fault Tree Analysis is a technique especially applicable to a mature product, such as the QNX Neutrino microkernel, where field usage figures and problem reports exist. Using a Bayesian Belief Network to express the Fault Tree allows us to incorporate both hard and soft evidence into our analysis of the product in a quantifiable way. We can then incorporate the results of this an analysis into a larger model, of which the Fault Tree forms just one component. This model expresses a full, quantified safety case for the product.

**Appendix — Bayes’ Theorem**

The Rev. Thomas Bayes published his famous theorem in the 18th century. If belief can be identified with probability, then the theorem allows reasoning from effect to cause as follows:

If X were true then Y would result. Y is actually true. This increases my belief in X by a certain amount.

Clearly, depending on the *a priori* unlikeliness of X, the amount by which X increases the belief may be very small or quite large. Smoking, for example, may cause dyspnoea (or shortness of breath) although many smokers do not have dyspnoea and dyspnoea has many other causes. If a patient presents himself to a doctor with dyspnoea, then the doctor's belief that the patient is a smoker will strengthen: reasoning from effect to possible cause.

More formally, given a hypothesis, h and some evidence e, Bayes' theorem states that

\[
P(b | e) = \frac{P(e | b) P(b)}{P(e | b) P(b) + P(e | \neg b) P(\neg b)}
\]

where \( P(X | Y) \) is the probability that X occurs given that Y has occurred and \( \neg X \) means "not X".

As a trivial example, assume that:

- \( h \) is the hypothesis that “It is raining at the moment”.
- \( e \) is the evidence that “I have just seen Chris with his umbrella”.

\[ ^5 \text{This is a perennial philosophical debate, the argument depending on one's interpretation of "probability".} \]
Assume that we know that:

- Chris carries his umbrella 60% of the time when it is raining ($P(e|b) = 0.6$).
- Chris carries his umbrella 30% of the time when it is not raining ($P(e|\neg b) = 0.3$).
- In the area where Chris lives it rains 20% of the time ($P(b) = 0.2$). This is known in the literature as the prior probability because it is a measure of the probability of the hypothesis before any evidence is considered.

Given these values, a person observing Chris in the street with his umbrella can calculate the probability that it is raining:

$$P(b|e) = \frac{P(e|b) \cdot P(b)}{P(e|b) \cdot P(b) + P(e|\neg b) \cdot P(\neg b)}$$

$$= \frac{0.6 \times 0.2}{0.6 \times 0.2 + 0.3 \times 0.8}$$

$$= 0.33$$

Note that this technique allows us to argue from effect (Chris is carrying his umbrella) to cause (It is raining).

**Further Reading**


About QNX Software Systems

QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry’s most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com