From Safety Analysis to Software Requirements

Kirsten M. Hansen, Anders P. Ravn *Member*, *IEEE*, and Victoria Stavridou, *Member IEEE*

Abstract—Software for safety critical systems must deal with the hazards identified by safety analysis. This paper investigates, how the results of one safety analysis technique, fault trees, are interpreted as software safety requirements to be used in the program design process. We propose that fault tree analysis and program development use the same system model. This model is formalized in a real-time, interval logic, based on a conventional dynamic systems model with state evolving over time. Fault trees are interpreted as temporal formulas, and it is shown how such formulas can be used for deriving safety requirements for software components.

Index Terms—Safety analysis, fault trees, requirements engineering, formal methods, temporal logic, real-time systems.

✦

1 INTRODUCTION

IN "Software Safety in Embedded Computer Systems" [14], Leveson advocates the use of system safety analysis techniques to derive system safety constraints which must be satisfied by software requirements. She further argues that the software requirements must be formalized in order to raise confidence in the verification. This paper extends these ideas by demonstrating how fault trees resulting from safety analysis can be interpreted directly as requirements. It provides a link between the worlds of the safety and software engineers, thus helping to reduce the errors occurring on the interface between the two engineering disciplines. We link fault tree analysis to program development, by requiring that both use the same system model. By using a common model, it is possible to use the results of the fault tree analysis directly, when specifying and designing the software. It is also possible to prove formally that a program is safe, i.e., that it does not cause the system to violate its safety requirements. The key point is to give a common interpretation to terms used in safety analysis and in requirements formulation; in other words to give a common semantic model.

A common framework is important whenever engineers from multiple disciplines need to work together. This is so since it is known that most system failures can be attributed to subtle design faults introduced because of a mismatch of assumptions and constraints originating from different system aspects. The digital flight control system (DFCS) part of the NASA advanced fighter technology integration (AFTI) F–16 program has highlighted these problems. According to NASA engineer Dale Mackall [19]:

> *Overall, the integrated DFCS provided many operational benefits. The hardware reliability of the complex system was excellent. However, the complexity of the system, coupled with the wide range of disciplinary engineers involved, caused numerous design oversights.*

The analysis of the DFCS test data has shown that the overwhelming majority of failure indications were not due to actual hardware failures but to design oversights concerning asynchronous computer operation. In particular, these failures were due to the lack of understanding of the interactions between the air data system, the redundancy management software and the flight control laws. In other words a missing common framework.

It is usual in engineering to use a mathematical theory to provide such a common framework for a given discipline or area. The basic framework we propose uses state variables, denoting functions of time, as in conventional dynamic systems theory [18]. Such models are known to engineers in general, often through their graphical representation by timing diagrams. The properties which we can model, are thus relations among time varying states. To specify such relations we use a real-time interval logic, the *duration calculus* [33], [10], which has been used successfully for requirements specification and design [26], [24], [25]. Other temporal and real-time logics such as [11], [20], [21] could be used in a similar manner. Whereas such logics are well established in formal specifications of software, they may be unfamiliar to other engineers. The underlying dynamic systems framework and the ability to illustrate duration calculus formulas by timing diagrams (cf. also graphical interval logic [4]) has, in our experience, helped to overcome this problem.

In our efforts to build a common semantic model for safety analysis and software requirements specifications, we have discovered that the accepted informal descriptions of fault tree gates are ambiguous, allowing several very different interpretations. For instance, the semantics of an AND-gate in the *Fault Tree Handbook* [32], is defined as:

- *K.M. Hansen is with the Danish National Rail Agency, ScanRail Consult, Pilestræde 60, 1, DK-1112 København K, Denmark.*
- *A.P. Ravn is with the Department of Information Technology, Technical University of Denmark, DK-2800 Lyngby, Denmark. E-mail: apr@it.dtu.dk.*
- *V. Stavridou is with the Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo , CA 94025. E-mail: victoria@csl.sri.com.*

Manuscript received 22 Feb. 1994; revised 1 Sept. 1997.
Recommended for acceptance by J. Knight.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106552.

"The output fault occurs only if all the input faults occur." But what exactly does this mean? Does it mean that all the faults have to occur at the same time, or does it mean that all faults have to occur, but that they need not overlap in time? And what is a fault? Clearly such uncertainty is not desirable when dealing with safety critical systems, reenforcing the motivation for formalization.

1.1 Related Work

There is an extensive literature on fault tree analysis and supporting tools, but only recently have there been attempts to relate it to software.

In [8] fault trees have been given a partial orders semantics and in [7] they have been given a petri net semantics. These two semantics are used to obtain a precise meaning of the results of a fault tree analysis, but it is not explained how they are used further on in software development.

In [1] fault trees have been assigned a modal μ-calculus semantics. This semantics differs from ours in that it defines a cause-effect relation between the input and output events of a gate, and in that it specifies that the output event does not necessarily occur if the input event occurs. During our work we defined a similar semantics. Later on we rejected this semantics, because it allows an optimistic interpretation of fault trees, which we do not find appropriate when building safety critical systems, see Section 4 for details. In [1] it is assumed that the fault tree analysis is performed in one system model (a safety model), and that the system development is performed in another system model. The model in which the semantics of fault trees is defined is used to validate the model in which the system development is performed in the sense that everything that is expressible in the safety model must also be expressible in the system model. An example is that if the safety model states that there is a train collision if there are two trains on a track segment, then the system model should be capable of expressing that there are two trains on a track segment. We have avoided this problem, as we require that the safety analysis and the system development use the same system models.

The static structure of systems is formalized using ordinary set theory in [16]. The system model defines a set of entities that denote components and a set of binary relations corresponding to physical dependencies among components. They also introduce events, where each event has an associated list of involved entities and a probability. The system model is structural and does not formalize dynamics, i.e., there is no temporal order associated with events. The events are used in constructing a fault tree, where each node is labeled by some event. Semantic checks are used to eliminate cyclic occurrences of events and against missing dependencies from the system model. The semantics of fault trees is not compositional, because nodes are labeled independently, and thus it is is not clear how requirements are derived from the analysis. The system model idea can be merged with our framework giving structural properties in addition to the dynamic properties that we discuss. However, we think such an extended framework should use a compositional semantics.

Useful procedures for applying fault tree techniques in software development are discussed in [29]. That work, however, does not discuss semantics at all.

A different link between software and safety analysis uses safety analysis techniques for assessing software [15], [14]. These techniques focus on analyzing eventuality properties of program executions by giving selected parts of programs a fault tree semantics. This approach is different from ours in the sense that it uses fault tree analysis on program code, whereas we use the fault tree analysis to derive the safety requirements which a program should fulfill. It is thus a program verification technique and not a requirements specification step.

In [3] the analysis of a program entering a certain unwanted state is extended to cover failures in the form of low voltage, radiation, etc., The analysis is based on representing weakest precondition predicates as fault trees. The approach is informal in the sense that fault trees are not assigned a formal semantics and in the sense that no formal relation is defined between the weakest precondition predicates and the fault trees.

1.2 Overview

We begin by introducing fault trees and their syntax in Section 2. In Section 3, we introduce the duration calculus and justify the choice of an interval logic. We use the logic to give fault trees formal semantics in Section 4. In Section 5, the semantics is used to relate fault tree analysis to software requirements and development. We conclude with examples and a discussion in Sections 6 and 7, respectively.

2 Fault Trees

Fault tree analysis [32] is a deductive safety analysis technique which is applied during the design phase. It is a top-down approach whose input consists of knowledge of the system's functions as well as its failure modes and their effects. The result of the analysis is a set of combinations of component failures that can result in a specific malfunction. The approach is graphical, constructing fault trees using standardized symbols.

A fault tree has a root which represents a total or catastrophic failure. For each catastrophic failure, the system is analyzed to reveal the possible causes of this failure. This analysis is used to populate the lower levels of the fault tree. Each subtree may be regarded as a fault tree for a component constituting a system of its own. The extent of the analysis, i.e., which components are considered basic, depends on the abstraction level chosen.

A fault tree is not a model of all possible causes for system failure; but given a particular failure, it reveals the possible combinations of component failures that may lead to this failure. Fault tree analysis is basically a qualitative model, but it is also often used in probabilistic analysis. In this paper we are not concerned with the probabilistic extensions to the basic model.

A fault tree is constructed from a predefined set of symbols [32], shown in Fig. 1. There are several variations and extensions, but in this article we limit ourselves to the above symbols.

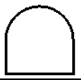
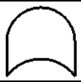
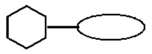
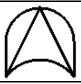
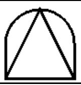
$\boxed{X}$	A node denoting a failure X, possibly resulting from a combination of basic failures (nodes).
	AND-gate—the failure in the top node occurs only when all the failures in the children nodes occur.
	OR-gate—the failure in the top node occurs only when one or more of the failures in the children nodes occur.
	INHIBIT-gate—the failure in the top node occurs only when both the failures in the child node occurs and the condition in the oval is true.
	EXCLUSIVE OR-gate—the failure in the top node occurs only when exactly one of the failures in the children nodes occurs.
	PRIORITY AND-gate—the failure in the top node occurs only when the failures in the children nodes occur in a left to right order.

Fig. 1. Fault tree symbols.

In fault tree terminology no distinction is made between faults, errors and failures, and in practice, the nodes of trees seem to contain a mixture of these. We use the terms fault, error and failure such that an error is the manifestation of a fault in the system, and a failure is the effect of an error on the service of the system [13], [12].

Fig. 2 is an example of a fault tree that relates to an electronically controlled combustion mechanism such as a gas burner. It is an extension of an example in [30]. It states that if a fire occurs, then either both an excess of gas and air have been present at the same time as an ignition has been attempted, or some of the cables have short-circuited. Further the fault tree states that if an excess of gas is present, then during a 30 sec period gas has leaked for more than 4 sec.

The problem with this tree is that it allows several different interpretations. What does the statement "Gas leaks for more than 4 sec" mean? Does this mean that gas should leak continuously for more than 4 sec, or does it mean that gas could leak for 2 sec, then stop leaking for 10 sec and then leak again for 3 sec? And if gas ignites, does air, gas, and ignition have to have been present *at the same time*? Our intuition says yes, because we know something about combusion. In Section 3 we present a real-time interval logic, the duration calculus [33], which we use to make such statements precise.

3 Duration Calculus

The formalization of fault trees is fairly straightforward in the case of the simple AND- and OR-gates. They correspond to the Boolean connectives of propositional logic. It is less obvious how to formalize the 'events' of the basic nodes. In some cases they correspond to state transitions, in other cases they denote state occurrence. They may also rely on some minimal or maximal time of occurrence. A common thread for in all these possible interpretations is that events are observed while time passes, i.e., over finite intervals of time, when certain state patterns occur, suggesting the use of a real-time, interval logic.

We introduce the duration calculus by formalizing the statements from the fault tree in Fig. 2: "Gas leaks for more than 4 sec" and "Observation interval less than 30 sec." We generalize the description of the calculus at the end of this section.

The first step in formalizing statements about a system is to construct a system model. As basis we use the *time-domain model* [18], where a system is described by a collection of *states* which are functions of *Time*. Here we take *Time* to be the non-negative reals, but discrete time domains could also be used. To formalize the first statement, that gas leaks for more than 4 sec, we use the following Boolean valued states

$$Gas, Flame : Time \rightarrow \{0, 1\}$$

which express the presence of gas and flame as functions of time. We assume that Boolean values are represented by 0 (*false*) and 1 (*true*).

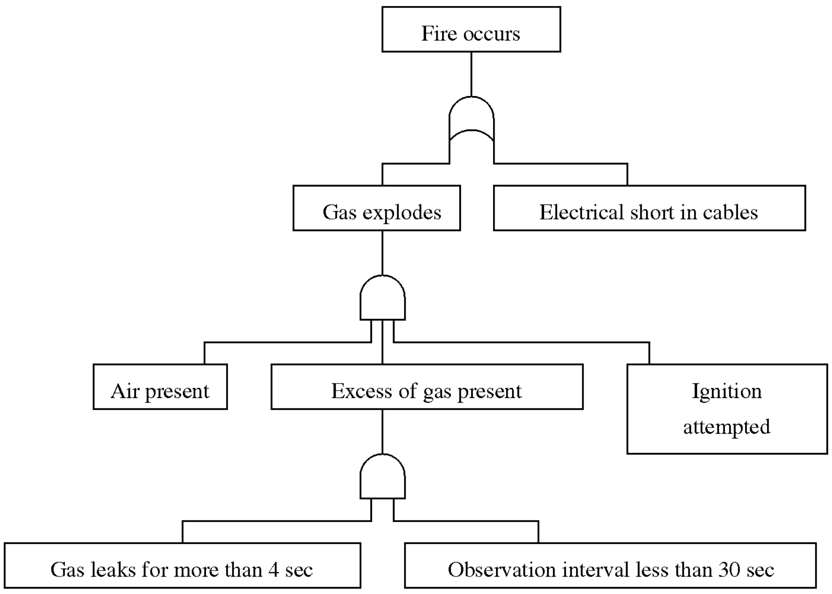

Fire occurs
Gas explodes
Electrical short in cables
Air present
Excess of gas present
Ignition attempted
Gas leaks for more than 4 sec
Observation interval less than 30 sec

Fig. 2. Fault tree for a gas burner.

Statements about a system are expressed by constraining states over time. We assume that a leak occurs whenever the state assertion $Gas \wedge \neg Flame$, i.e., the gas valve is open and there is no flame, is true. We abbreviate this composite state

$$Leak \overset{def}{=} Gas \wedge \neg Flame$$

When we consider a bounded time interval $[b, e]$, we can measure the *duration* of *Leak* within the interval by $\int_b^e Leak(t)\,dt$, cf. the timing diagram in Fig. 3. We introduce the notation $\int Leak$ for the duration of *Leak*. It denotes a function from intervals to reals; a real number for each particular interval. "Gas leaks for more than 4 sec" is thus written as

$$\int Leak > 4$$

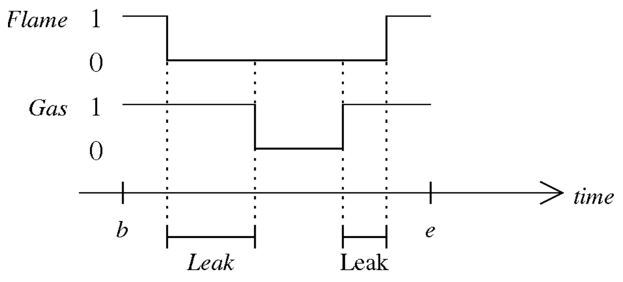


Fig. 3. Timing diagram for *Leak*.

The duration of the constant state 1, $\int 1$, is the length of the interval under consideration. We abbreviate $\int 1$ by ℓ. Thus, the fact that an interval is not longer than 30 sec is specified by the formula $\ell \leq 30$. This completes the formalization of the two nodes from the fault tree in Fig. 2.

Formulas may be combined by the usual logical connectives. Thus, we may combine the above formulas by Boolean connectives and obtain

$$(\ell \leq 30) \wedge (\int Leak > 4)$$

that is, the considered time interval is not longer than 30 seconds and gas leaks for more than 4 sec.

The property that a state, *Ignition*, holds throughout a nonpoint interval is defined by $\int Ignition = \ell \wedge \ell > 0$, abbreviated $\lceil Ignition \rceil$.

Subinterval properties are expressed by the binary "chop" operator (written ";") of interval logic. Given formulas D_1 and D_2, the formula $D_1 ; D_2$ holds for the interval $[b, e]$ just when this interval can be divided into an initial subinterval $[b, m]$ where D_1 holds and a final subinterval $[m, e]$ where D_2 holds.

"Somewhere" D, written $\Diamond D$, i.e., D occurs in some subinterval, is defined by the "chop" operator as $\Diamond D \overset{def}{=} true ; D ; true$. The dual to "somewhere" is "everywhere," written $\Box D$ and defined as $\neg \Diamond (\neg D)$.

A safety constraint S should hold for an arbitrary interval of the system lifetime. This can be expressed as: There is no subinterval for which the formula $\neg S$ holds. If we define S as

$$S \overset{def}{=} l \leq 30 \Rightarrow (\int Leak \leq 4)$$

then $\neg S = (\ell \leq 30) \wedge (\int Leak > 4)$ meaning that the observation interval is not longer than 30 sec and gas leaks for more than 4 sec.

The safety constraint for the gas-burner is thus $\neg \Diamond (\neg S)$ which is equivalent to $\Box S$.

3.1 Duration Calculus, Summary

The duration calculus is based on Moszkowski's (discrete time) interval logic (ITL) [21] which provides the basic binary modality "chop," written ';'.

3.1.1 Syntax

The syntax of duration calculus distinguishes (*duration*) *terms*, each one associated with a certain type, and (*duration*) *formulas*. Terms are built from names of elementary states like *Gas* and *Flame*, and *rigid variables* representing time independent logical variables, and are closed under arithmetic and propositional operators. Examples of terms are $\neg Flame$ and $Gas = 0$.

Terms of Boolean type are called *state assertions*. We use P to denote state assertions. For a state assertion P, the integral $\int P$ is called the *duration* because it measures the time P holds in the given interval.

Duration formulas are built from duration terms of Boolean type and are closed under propositional connectives, the chop connective, and quantification over rigid variables and variables of duration terms. We use D for a typical duration formula.

3.1.2 Semantics

The semantics of duration calculus is based on an *interpretation* $\mathcal{I}$ that assigns a fixed meaning to each state name, type and operator symbol of the language, and a time interval $[b, e]$. For given $\mathcal{I}$ and $[b, e]$ the semantics defines what domain values duration terms and what truth values duration formulas denote. For example, $\int P$ denotes the $\int_b^e P(t)\,dt$.

A duration formula D *holds* in $\mathcal{I}$ and $[b, e]$, abbreviated $\mathcal{I}, [b, e] \models D$, if it denotes the truth value *true* for $\mathcal{I}$ and $[b, e]$. D is *true* in $\mathcal{I}$, abbreviated $\mathcal{I} \models D$, if $\mathcal{I}, [a, b] \models D$ for every interval $[a, b]$. A *model* of D is an interpretation $\mathcal{I}$ which makes D true, i.e., with $\mathcal{I} \models D$. The formula D is *satisfiable* if there exists an interpretation $\mathcal{I}$ with $\mathcal{I} \models D$.

A *behavior* is an interpretation restricted to the names of the elementary states.

Duration calculus has a powerful proof system, cf. [25], which justifies its use in specifications. We use elementary properties in Section 4.

4 Fault Tree Semantics

In general, fault trees can be viewed as (temporal) logic formulas with uninterpreted basic symbols.

4.1 Leaves

In safety analysis terminology, the leaves in a fault tree are called events, often meaning the occurrence of a specific system state, but also used in the software engineering sense, meaning a transition between two states. In order to avoid misunderstandings we want to stress that whenever we use the term event in this article we mean a state transition (the software engineering interpretation of an event).

Our studies of fault trees have revealed that the contents of fault tree leaves depend much on the application; sometimes the leaves denote states and sometimes events. We, therefore, interpret a leaf node of a fault tree as a duration calculus formula. Such a formula may *for instance* be:

- the constants *true*, *false*
- occurrence of a state P, i.e., $\lceil P \rceil$
- occurrence of an event, i.e., a transition to state P: $\lceil \neg P \rceil ; \lceil \mathrm{P} \rceil$
- elapse of a certain time, i.e., $\ell \geq (30 + \epsilon)$, or
- a threshold of some duration, i.e., $\int P \leq 4 \times \epsilon$.

It is crucial that the safety engineer and the software engineer agree on the interpretation of the contents of leaves as formulas. This may for instance be done by interpreting the formulas as timing diagrams. Fig. 4 shows timing diagrams for some basic observations.

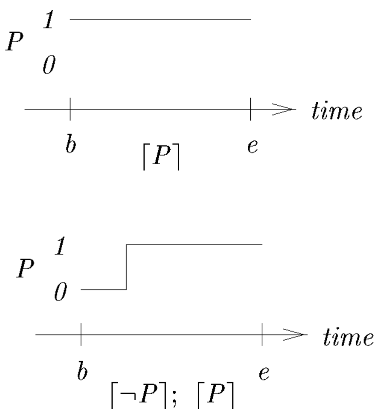


Fig. 4. Timing diagrams for some basic observations.

4.2 Intermediate Nodes

The semantics of intermediate nodes is defined by the semantics of the leaves, edges, and gates in the subtrees in which the intermediate nodes are roots. Intermediate nodes are merely names of the corresponding subtrees. In the following we define the semantics of intermediate nodes from the structure of the subtree.

4.3 Edges

Given interpretations of the leaves, we now consider the meaning of an intermediate node, A, connected to a node, B, by an edge, see Fig. 5. Assume that the semantics of B is B. We then define the semantics of A to be

$$A \overset{def}{=} B$$

i.e., as logical identity, meaning that the system failure A occurs when the failure B occurs. This semantics is pessimistic in the sense that it assumes that if something has the possibility of going wrong, then it does go wrong.

Informal readings of fault trees often state that it is not mandatory that A holds when B holds [32], [30], which is formalized as $A \Rightarrow B$. This semantics allows an optimistic interpretation of fault trees in the sense that a system failure may be avoided if the operator intervenes fast enough, has enough luck, etc. We do not think that speed, luck, and the like should be design parameters in safety critical systems, and therefore we have rejected this semantics.

Another issue is whether A and B occur at the same time or if there is a necessary delay from the occurrence of B to the occurrence of A, formally: $\lceil B \wedge \neg A \rceil ; \lceil B \wedge A \rceil$. Often there will be such a delay, but we have refrained from modeling it, as this again would give the impression that once B has occurred there is a possibility that A can be prevented.

4.4 Gates

We now consider the semantics of intermediate nodes connected to other nodes through gates.

4.4.1 AND

In the fault tree in Fig. 6 assume that the semantics of $\mathrm{B}_1, \ldots, \mathrm{B}_n$ is $B_1, \ldots, B_n$. We then define the semantics of A to be

$$A \overset{def}{=} B_1 \wedge \ldots \wedge B_n$$

i.e., A holds iff B_1 to B_n hold simultaneously.

We have considered a more liberal interpretation of AND-gates in which B_1 to B_n need not hold simultaneously, namely $A = \Diamond B_1 \wedge \ldots \wedge \Diamond B_n$. This has been rejected since this formula "remembers any occurrence of a B_i," such that if B_2 becomes true 1 year after B_1, and B_3 becomes true 3 years after B_2, and …, then A holds. This is clearly not the intended meaning of an AND-gate.

4.4.2 OR

For the fault tree in Fig. 7 assume that the semantics of $\mathrm{B}_1, \ldots, \mathrm{B}_n$ is $B_1, \ldots, B_n$. We define the semantics of A to be

$$A \overset{def}{=} B_1 \vee \ldots \vee B_n$$

i.e., A holds iff either B_1 or … or B_n holds. This interpretation shows that an OR-gate at the top of a fault tree means single point failure. If just one of the formulas holds, the failure occurs.

4.4.3 INHIBIT

We consider INHIBIT-gates where the condition is *not* a probability statement. According to the fault tree handbook, [32], the fault tree in Fig. 8 reads: "If A occurs then B_1 has occurred in the past while condition B_2 was true." In a direct translation of this sentence, we interpret B_1 as an event (state transition) that must occur while the system is in state B_2 in order to make A happen. In our semantics we have however refrained from requiring that B_1 must be an event and that B_2 must be a state, as an interpretation equally well could be the occurrence of a state B_1 while another state B_2 was true. We therefore interpret an INHIBIT-gate as an AND gate with B_1 and B_2 as inputs, where B_1 and B_2 are duration formulas. Generally, if B_n is the condition, and $B_1, \ldots, B_{n-1}$ are children to the INHIBIT-gate, the semantics is given by

$$A \overset{def}{=} B_1 \wedge \ldots \wedge B_n$$

4.4.4 EXCLUSIVE OR

A fault tree with an EXCLUSIVE OR-gate is pictured in Fig. 9. According to the fault tree handbook, [32], this tree

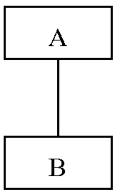


Fig. 5. Fault tree with no gates.

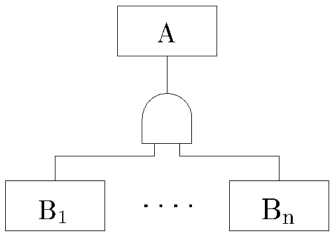


Fig. 6. Fault tree with AND-gate.

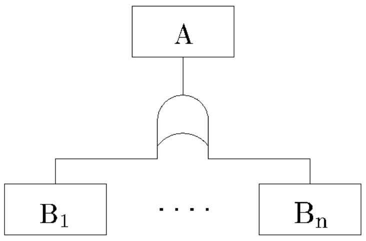


Fig. 7. Fault tree with OR-gate.

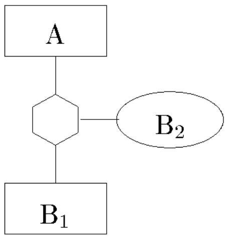


Fig. 8. Fault tree with INHIBIT-gate.

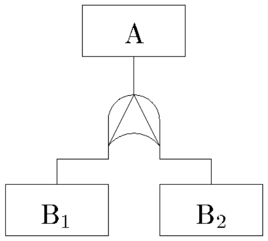


Fig. 9. Fault tree with EXCLUSIVE OR-gate.

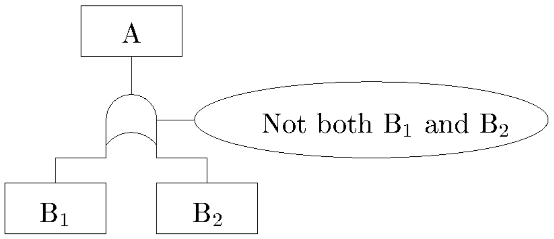


Fig. 10. Fault tree with OR-gate and a condition.

may be drawn as in Fig. 10, in which “Not both B_1 and B_2” is a necessary condition for A to hold. As for the INHIBIT-gate we interpret the condition “Not both B_1 and B_2” as a leaf which should also hold. By interpreting “Not both B_1 and B_2” as $\neg(B_1 \wedge B_2)$, we obtain the semantics $A \overset{def}{=} (B_1 \vee B_2) \wedge \neg(B_1 \wedge B_2)$, which may be rewritten to

$$A = (B_1 \wedge \neg B_2) \vee (\neg B_1 \wedge B_2)$$

This generalizes to

$$\begin{aligned} A \overset{def}{=} & (B_1 \wedge \neg(B_2 \vee \ldots \vee B_n)) \\ & \vee \\ & \vdots \\ & \vee \\ & (B_n \wedge \neg(B_1 \vee \ldots \vee B_{n-1})) \end{aligned}$$

4.4.5 PRIORITY AND

A fault tree with a PRIORITY AND-gate is pictured in Fig. 11. The informal semantics states that A occurs just when all of the children nodes occur in a left-to-right order. Assuming that $B_1, \ldots, B_n$ have the semantics $B_1, \ldots, B_n$ we, therefore, define the semantics of A to be

$$A \overset{def}{=} B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \ldots \wedge \Diamond B_n) \ldots)$$

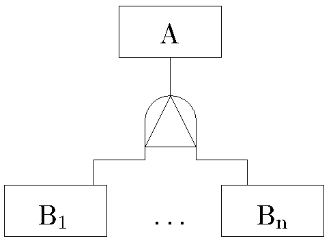


Fig. 11. Fault tree with PRIORITY AND-gate.

4.5 Trees

The semantics of a fault tree is determined by the semantics of the leaves, the edges, and the gates, such that the semantics of intermediate (not leaves) nodes are given by the semantics of the leaves, edges, and gates in the subtrees in which the intermediate nodes are roots.

The above procedure assigns semantics to fault trees in a compositional style. The meaning of a composite tree is given by a temporal formula denoting the meaning of the subtrees connected to the gate, while leaves are assigned a formula independent of their position in the tree.

5 Software Safety Requirements

Traditionally fault trees are used to analyze existing system designs with regard to safety. Instead of first developing a design, and then performing a safety analysis, we propose that the design and the safety analysis should proceed concurrently, thereby making it possible to let the fault tree analysis influence the design. In order to do this, the fault tree analysis and the system design must use the same system model. Given a common model, the system safety requirements may be deduced from the fault tree analysis. Safety requirements, derived this way, can be used during system development in order to validate the design, but they can also be used in a constructive way by influencing the design. We illustrate this below.

For each fault tree in which the root is interpreted as S, the system should be built such that S never occurs, i.e., the safety commitment which the system should implement is

$$\Box \neg S$$

If we have n fault trees in which the roots are interpreted as $S_1, \ldots, S_n$, the safety commitment which may be deduced from these fault trees is

$$\Box \neg S_1 \wedge \ldots \wedge \Box \neg S_n$$

i.e., the system should ensure that no top event in any fault tree ever holds. This corresponds to combining the trees by an OR-gate.

5.1 Deriving Component Requirements

If the fault tree contains gates, the derived specifications depend on the type of gates. Here we only consider AND-, OR-, and PRIORITY AND-gates, as fault trees containing other gates may be expressed in terms of these.

5.2 AND-Gates

Recall that the fault tree in Fig. 6 has the semantics $A = B_1 \wedge B_2 \wedge \ldots \wedge B_n$ and assume that the safety commitment is $\Box \neg A$. This safety commitment corresponds to specifying that the components never satisfy their duration formulas at the same time, i.e.,

$$\Box \neg(B_1 \wedge B_2 \wedge \ldots \wedge B_n)$$

One way to implement this is to implement the stronger formula

$$\Box \neg B_1 \vee \Box \neg B_2 \vee \ldots \vee \Box \neg B_n$$

i.e., to design at least one of the components such that it always satisfies its local safety commitment.

Often, the software engineer does not control all the input components to an AND-gate. For such components a safe approach is to assume the worst case, namely that the component is in a critical state and thereby contributes to violation of the safety commitment. Let us for instance assume in the case of the fault tree in Fig. 6, that the first component is uncontrollable. The worst case is that the component satisfies B_1, i.e., that

$$\Box \neg(\mathit{true} \wedge B_2 \wedge \ldots \wedge B_n)$$

meaning that the software engineer has to implement

$$\Box \neg(B_2 \wedge \ldots \wedge B_n)$$

One should, at some point, arrive at a conjunction of B_is which can be used in the design. Otherwise, we must conclude that the system is inherently unsafe. If the design relies on the absence of only one B_i, it is a design which is vulnerable to single point failures.

5.3 OR-Gates

The fault tree in Fig. 7 has the semantics $A = B_1 \vee \ldots \vee B_n$. For the system to satisfy the safety commitment $\Box \neg A$, the software engineer must implement

$$\Box \neg(B_1 \vee \ldots \vee B_n)$$

or equivalently

$$\Box \neg B_1 \wedge \ldots \wedge \Box \neg B_n$$

This formula expresses that the system only satisfies its safety commitments if all its components satisfy their local safety commitments.

Now suppose that the software engineer cannot control the first component, i.e., whether that component satisfies B_1 or not, is outside the scope of the design of the program. Making the safe choice of B_1 being *true* causes $\Box \neg B_1$ to be *false* which trivially implies that the safety commitment is violated. Making a tacit assumption of B_1 being *false* is very poor judgment, which essentially ignores the results of safety analysis.

The only reasonable option is to weaken the requirements specification. We *assume* that the behavior of the first component never satisfies B_1, i.e., that $\Box \neg B_1$ is *true*. To make the design team as a whole aware of this assumption, we add it to the environment assumptions. So, if the requirements were given in an assumption commitment style by a formula, $\mathit{Asm} \Rightarrow \mathit{Com}$, before this design step, we have the assumptions $\mathit{Asm} \wedge \Box \neg B_1$ afterwards. The specification of the requirements $\mathit{Asm} \Rightarrow \mathit{Com}$ has thus been weakened to $\mathit{Asm} \wedge \Box \neg B_1 \Rightarrow \mathit{Com}$, and the software engineer should alert the appropriate persons to the fact that the system requirements have been weakened.

Many design errors are located on interfaces. Having an explicit list of assumptions and adding to this list as the system development progresses, makes the interfaces clearer and reduces the likelihood of errors.

5.4 PRIORITY AND-Gates

The fault tree in Fig. 11 has the semantics $A = B_1 \wedge \Diamond (B_2 \wedge \Diamond (B_3 \wedge \ldots \wedge \Diamond B_n) \ldots)$. If the safety commitment is $\Box \neg A$, the software engineer must implement

$$\Box \neg(B_1 \wedge \Diamond (B_2 \wedge \Diamond (B_3 \wedge \ldots \wedge \Diamond B_n) \ldots))$$

This may either be done by making the implementation such that the B_is do not occur in the specified order or such that one of the B_is does not occur at all, i.e.,

$$\Box \neg B_1 \vee \Box \neg B_2 \vee \ldots \vee \Box \neg B_n$$

If one of the B_is, e.g., B_1 is uncontrollable, the worst case is that it does not satisfy its local safety commitment, i.e., that B_1 is true. The software engineer therefore assumes that B_1 is true and attempts to make the design such that

$$\Box \neg (B_2 \wedge \Diamond (B_3 \wedge \ldots \wedge \Diamond B_n) \ldots)$$

holds.

6 Example

In Section 6, we illustrate the formalization of fault trees and the derivation of safety requirements by analyzing the hazards of a railway interlocking system. The main task of an interlocking system is to prevent trains from colliding and derailing while allowing train movements. In Fig. 12 we present a fault tree for train collision. The fault tree has been developed stepwise until some implementable nodes were reached. The underlying model is presented in full in [9]. The parameters in the predicates in the nodes of the fault tree denote the station topology, *tpo*, the station state, *sst*, and the state of the interlocking system, *intlck*.

A station topology consists of the placement of the track segments, the points,[1] and the signals. The state of a station is given by the state of the track segments, i.e., whether there is a train within a given track segment or not, by the state of the points, i.e., the branches in which the points have control, and by the signal aspects: “stop” or “drive.”

The state of an interlocking system is determined by the train routes currently set, i.e., the track segments reserved for the trains.

6.1 Safety Commitment

The safety commitment is that there should never be a collision

$$Safe\_com \overset{def}{=} \Box \neg Collision(tpo, sst, intlck)$$

Collision is true if there is more than one train within one track segment. The safety commitment may, therefore, be implemented by allowing at most one train on the station at a time. However, with the amount of train traffic today, this is not a desirable solution. Instead we look at the driving possibilities at stations.

6.2 Fault Tree Analysis

Two trains may end up within the same track segment, if it is possible for the trains to drive to the track segment either legally, i.e., respecting the stop signals, or illegally, i.e., not respecting the stop signals. Given a station topology and a fixed station state, it is possible to deduce the legal driving possibilities for each train at the station. We call the track segments to which a train may move legally, for the area for that train. Two trains may collide if their area have common track segments.

$$\begin{aligned} &Collision(tpo, sst, intlck) \overset{def}{=} \\ &\quad Overlap\_areas(tpo, sst, intlck) \\ &\quad \vee Signal\_bypass(tpo, sst) \end{aligned}$$

A train passes a stop signal, if at first the train is on a track segment in front of the signal, and then it is on the track segment after the signal, while the signal shows “stop” continuously.

$$\begin{aligned} &Signal\_bypass(tpo, sst) \overset{def}{=} \\ &\quad \exists sig \in tpo.signals, train \in sst.train \cdot \\ &\qquad \lceil Train\_before\_signal(train, sig, tpo, sst) \\ &\qquad \wedge Signals\_stop(sig, sst) \rceil \\ &\qquad ; \lceil Train\_after\_signal(train, sig, tpo, sst) \\ &\qquad \wedge Signals\_stop(sig, sst) \rceil \end{aligned}$$

6.3 Safety Requirements

The fault tree structure says that the safety commitment

$$\Box \neg Collision\ (tpo, sst, intlck),$$

i.e., that there must never be a collision, must be implemented by

$$\begin{aligned} &\Box \neg\ Overlap\_areas\ (tpo, sst, intlck) \\ &\wedge \Box\ \neg\ Signal\_bypass\ (tpo, sst) \end{aligned}$$

So far the Danish National Rail Agency have chosen not to implement $\Box \neg Signal\_bypass$ in interlocking systems. At some places it is implemented in a separate train protection system, and at other places it is only implemented in regulations [31] stating that trains must not pass stop signals. In either case, it has to be made explicit that $\Box \neg Signal\_bypass$ is not implemented in the interlocking system, by adding it to the assumptions about the environment, i.e.,

$$Asm \overset{def}{=} \Box \neg Signal\_bypass(tpo, sst)$$

The safety requirement deduced so far from the fault tree is therefore

$$\begin{aligned} &Safe \overset{def}{=} \Box \neg Signal\_bypass(tpo, sst) \\ &\Rightarrow \Box \neg Overlap\_areas(tpi, sst, intlck) \end{aligned}$$

It turns out that the area concept restricts the train traffic too much, as it assumes that trains may drive in any direction. If we instead grant each train a route which it must use, there may be more trains at a station.

6.4 Fault Tree Analysis

Using the train route concept, i.e., a sequence of track segments reserved for a particular train, two trains may drive to the same track segment either if one of the trains is not on its train route, or if both trains are on train routes, and it still is possible for the trains to drive to a common track segment.

$$\begin{aligned} &Overlap\_areas(tpo, sst, intlck) \overset{def}{=} \\ &\quad Outside\_route(sst.intlck) \\ &\quad \vee To\_comm\_track(tpo, sst, intlck) \end{aligned}$$

where

$$\begin{aligned} &To\_comm\_track(tpo, sst, intlck) \overset{def}{=} \\ &\quad On\_routes(sst, intlck) \\ &\quad \wedge Drive\_to\_comm\_track(tpo, sst, intlck) \end{aligned}$$

It is possible for two or more trains to drive to a common track segment, if the train routes overlap, if one of the trains reverses, or if one of the train routes is erroneous.

1. American: Switches

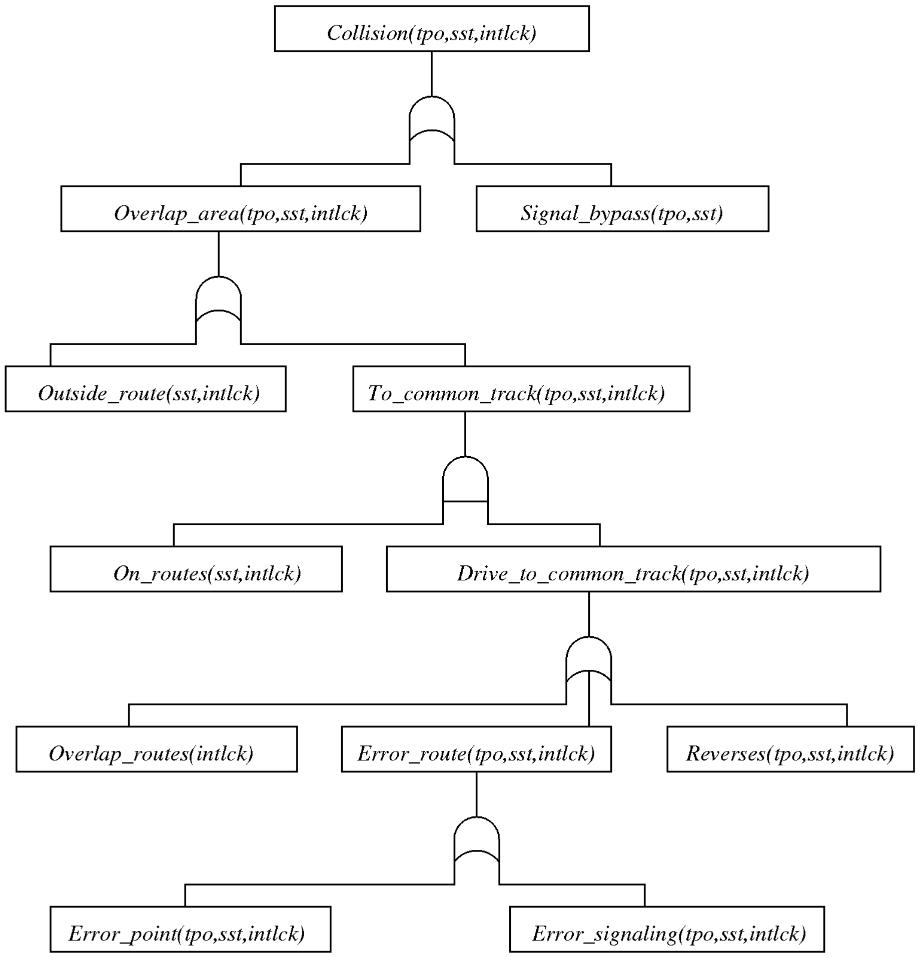

Collision(tpo,sst,intlck)
Overlap_area(tpo,sst,intlck)
Signal_bypass(tpo,sst)
Outside_route(sst,intlck)
To_common_track(tpo,sst,intlck)
On_routes(sst,intlck)
Drive_to_common_track(tpo,sst,intlck)
Overlap_routes(intlck)
Error_route(tpo,sst,intlck)
Reverses(tpo,sst,intlck)
Error_point(tpo,sst,intlck)
Error_signaling(tpo,sst,intlck)

Fig. 12. Fault tree for collision.

$$\begin{aligned}&\mathit{Drive\_to\_common\_track}(tpo, sst, intlck) \overset{def}{=}\\&\quad \mathit{Overlap\_routes}(intlck)\\&\quad \vee\ \mathit{Reverses}(tpo, sst, intlck)\\&\quad \vee\ \mathit{Error\_route}(tpo, sst, intlck)\end{aligned}$$

Two train routes overlap if they have a common track segment.

A train reverses if for a pair of track segments on a train route, one of the two following sequences of observations is made regarding the train:

	First segment	Second segment
Observation 1	*train*	*train*
Observation 2	*train*	*–train*

or

	First segment	Second segment
Observation 1	*train*	*–train*
Observation 2	*train*	*–train*

If a train is not on a train route, we cannot observe whether it reverses or drives forward, as we do not know in which direction it is supposed to move.

$$\begin{aligned}&\mathit{Reverses}(tpo, sst.intlck) \overset{def}{=}\\&\exists train \in sst.train,\ \exists(t1, t2) \in tpo.tracks \cdot\\&\quad ((\lceil \mathit{On\_track}(train, t1, sst)\\&\quad\ \ \wedge \mathit{On\_track}(train, t2, sst)\rceil\\&\quad ;\lceil \mathit{On\_track}(train, t1, sst)\\&\quad\ \ \wedge \neg \mathit{On\_track}(train, t2, sst)\rceil\\&\quad) \vee (\lceil \mathit{On\_track}(train, t1, sst)\\&\quad\ \ \wedge \neg \mathit{On\_track}(train, t2, sst)\rceil\\&\quad ;\lceil \neg \mathit{On\_track}(train, t1, sst)\\&\quad\ \ \wedge \neg \mathit{On\_track}(train, t2, sst)\rceil\\&\quad))\wedge\lceil \mathit{In\_intlck}(train, (t1, t2), \mathrm{int}\, lck)\rceil\end{aligned}$$

Finally, a train route is erroneous, if there is a point on the train route which has control in the other branch than the one through which the train route extends, or if there is not a stop signal between the front end of the train and the end of the train route.

$$\begin{aligned}&\mathit{Error\_route}\ (tpo, sst, intlck) =\\&\quad \mathit{Error\_point}\ (tpo, sst, intlck)\\&\quad \vee\ \mathit{Error\_signaling}\ (tpo, sst, intlck)\end{aligned}$$

Combining the above definitions according to the fault tree, we obtain

$$\begin{aligned} &Collision(tpo, sst, intlck) \overset{def}{=} \\ &\quad Signal\_bypass(tpo, sst) \\ &\quad \vee\, Outside\_route(sst, intlck) \\ &\quad \vee\, (\, On\_routes(sst, intlck) \\ &\qquad \wedge\, (\, Overlap\_routes(intlck) \\ &\qquad\quad \vee\, Reverses(tpo, sst, intlck) \\ &\qquad\quad \vee\, Error\_point(tpo, sst, intlck) \\ &\qquad\quad \vee\, Error\_signaling(tpo, sst, intlck)\,)) \end{aligned}$$

6.5 Safety Requirements

Previously we ended up with the safety requirement

$$\begin{aligned} &\Box \neg Signal\_bypass\ (tpo, sst) \\ \Rightarrow\ &\Box \neg Overlap\_areas\ (tpo, sst, intlck) \end{aligned}$$

Following the fault tree structure, $\Box \neg Overlap\_areas$ must be implemented by

$$\begin{aligned} \Box \neg(\ & Outside\_route(sst, intlck) \\ \vee\ & To\_common\_track(tpo, sst, intlck)\,) \end{aligned}$$

being equivalent to

$$\begin{aligned} &\Box \neg Outside\_route(sst, intlck) \\ \vee\ &\Box \neg To\_common\_track(tpo, sst, intlck) \end{aligned}$$

As *Outside_route* is a leaf, and as it is input to an OR-gate, it either has to be implemented, OR it must be justified that all trains are always on train routes. The Danish National Rail Agency requires that this is implemented by the operator of the interlocking system assigning each train a train route as it approaches a station (we do not consider shunting trains). As this is not implemented explicitly in the interlocking system, it is an assumption which must be ensured by the operator.

$$Asm' \overset{def}{=} \Box \neg Outside\_route(sst, intlck)$$

Further, $\Box\ \neg To\_common\_track$ must be implemented by

$$\Box \neg\begin{pmatrix} On\_routes(sst, intlck) \\ \wedge\, Drive\_to\_common\_track(tpo, sst, intlck) \end{pmatrix}$$

which again must be implemented by

$$\begin{aligned} \Box \neg(\ & On\_routes(sst, intlck) \\ & \wedge\, (\ Overlap\_routes(intlck) \\ & \qquad \vee\, Reverses(tpo, sst, intlck) \\ & \qquad \vee\, Error\_route(tpo, sst, intlck)) \end{aligned}$$

This is equivalent to

$$\begin{aligned} &\Box \neg\begin{pmatrix} On\_routes(sst, intlck) \\ \wedge\, Overlap\_routes(intlck) \end{pmatrix} \\ \wedge &\Box \neg\begin{pmatrix} On\_routes(sst, intlck) \\ \wedge\, Reverses(tpo, sst, intlck) \end{pmatrix} \\ \wedge &\Box \neg\begin{pmatrix} On\_routes(sst, intlck) \\ \wedge\, Error\_route(tpo, sst, intlck) \end{pmatrix} \end{aligned}$$

Here, $\Box\ \neg(On\_routes \wedge Overlap\_routes)$ is a safety commitment which must be implemented. Strictly speaking this safety commitment may be implemented by $\Box\ \neg On\_routes$. Such an implementation would however violate the assumption $\Box\ \neg Outside\_route$ and therefore it is not acceptable. So, we must design the algorithms for setting and releasing train routes in such a way that two train routes never overlap.

Currently, in the Danish National Rail Agency, $\Box\ \neg(On\_routes \wedge Reverses)$ is implemented in regulations [31], stating that trains are not allowed to reverse on train routes without special permission. If we follow the tradition of the Danish National Rail Agency and do not implement this in the interlocking system, we have to make explicit that our design relies on it being implemented elsewhere, i.e., the assumption

$$\begin{aligned} &Asm'' \overset{def}{=} \\ &\Box \neg\begin{pmatrix} On\_routes(sst, intlck) \\ \wedge\, Reverses(tpo, sst, intlck) \end{pmatrix} \end{aligned}$$

Finally, still using the fault tree structure, we have that $\Box\ \neg(On\_routes \wedge Error\_route)$ must be implemented by

$$\begin{aligned} \Box \neg(\ & On\_routes\ (sst, intlck) \\ & \wedge\, (Error\_point\ (tpo, sst, intlck) \\ & \qquad \vee\ Error\_signaling(tpo, sst, intlck))) \end{aligned}$$

Composing the above deductions we get that the safety commitment is

$$\begin{aligned} &Safe\_com \overset{def}{=} \\ &\Box \neg(\ On\_route(sst, intlck) \\ &\qquad\quad \wedge\, Overlap\_routes(intlck) \\ &\qquad \wedge\, (\ Error\_point(tpo, sst, intlck) \\ &\qquad\qquad \vee\, Error\_signaling(tpo, sst, intlck))) \end{aligned}$$

i.e., each train should be assigned a train route, and the train routes should be made in such a way that they do not overlap and that all points and signals on the routes are in states so that the trains cannot leave the train routes.

The assumptions which we have made so far are

$$\begin{aligned} Asm \overset{def}{=}\ & \Box \neg Signal\_bypass(tpo, sst) \\ & \wedge \Box \neg Outside\_route(sst, intlck) \\ & \wedge \Box \neg\begin{pmatrix} On\_routes(tpo, sst, intlck) \\ \wedge\, Reverses(tpo, sst, intlck) \end{pmatrix} \end{aligned}$$

i.e., trains do not pass stop signals, all trains drive on train routes, and trains do not reverse on train routes. The assumptions must be considered together as otherwise the assertion $\Box\ \neg(On\_routes \wedge Reverses)$ may be satisfied by $\Box\ \neg On\_routes$, thereby violating the assumption $\Box\ \neg Outside\_route$. Likewise as explained above, the assumptions and the safety commitments have to be considered together.

The deduced safety requirements are

$$Safe \overset{def}{=} Asm \Rightarrow Safe\_com$$

This is what a programed interlocking system must ensure.

7 Concluding Remarks

We have linked one safety analysis technique, fault tree analysis, to requirements specification so that software safety requirements can be derived directly from the system safety requirements. In the development of safety critical systems, this means that the software may be proven to satisfy the system safety requirements. The link is based on a dynamic systems model where states are functions of continuous time and the behavior of a system is defined by duration formulas that constrain the behavior within a bounded time interval. Fault tree nodes denote such formulas while gates denote formulas which express the temporal, causal ordering of events in nodes. Such an interpretation would also be possible and desirable with other temporal or real-time logics. The crucial point is, that the logic is capable of expressing both the semantics of the intermediate events, based on the structure of the fault tree, and the semantics of the leaves.

A very interesting opportunity exists where the software is implemented in a logic language. The semantics of fault trees may then be given in that logic, possibly using explicit encoding of time for temporal properties. It may then be feasible to check the consistency of systematically derived requirements directly against the program using automated tools.

Our work was originally inspired by the way that the former Danish State Railways developed requirement specifications for interlocking systems. They first performed a hazard analysis of the system, corresponding to deriving the roots of the fault trees, and next they negated the hazards thereby obtaining the safety requirements for the interlocking systems.

In practice, fault trees are developed after the system has been designed. Our work here shows how fault tree analysis and system design can be made to interact much more effectively. The method we propose enables feedback from the formulation of the software safety requirements to the system design. A good example of how this might work in practice, is the case of the software having to cope with single point failures as identified in Section 5; when such a software requirement is derived, a prudent systems architect might choose to modify the design, perhaps by adding some form of redundancy.

Recently, standards have been developed in the area of safety critical systems. These standards either advocate, or require, that formal methods are used in program development, and that safety analyses must be performed on the system, e.g., [2] for railway applications and [28], [27] for military applications. The work presented here integrates the standards in the areas of safety analysis and program development in a well defined, constructive way.

Our work has not yet been applied in full scale industrial practice, although it has inspired the derivation of software safety properties in one case. However, our results are currently used by an industrially led research project [6] on a real avionics system in order to arrive at a machine aided, stable, commercially viable software risk assessment methodology.

We have extended the ideas presented in this paper in [9] where we also have shown how to derive safety requirements from event trees [23] and cause-consequence diagrams [22].

One direction for further work is to extend the interpretation of fault trees to probabilistic reasoning. An initial step in that direction is the probabilistic version of the duration calculus [17], and the fault tree to Markov chain conversion [5]. The probabilistic duration calculus is based on discrete parameter Markov chains, and allows a designer to calculate the probability of the occurrence of a certain behavior within a given time. This would, in principle, allow us to interpret the probability figures that are often added to fault trees.

ACKNOWLEDGMENTS

This work was partially supported by the British Council, the Danish National Railway Agency, and the Commission of the European Communities (CEC) under the ESPRIT program Basic Research Action Project No. 7071: “ProCoS: Provably Correct Systems.”

REFERENCES

[1] G. Bruns and S. Anderson, “Validating Safety Models with Fault Trees,” J. Górski, ed., *SAFECOMP’93, Proc. 12th Int’l Conf. Computer Safety, Reliability and Security*, pp. 21–30. Springer-Verlag, 1993.

[2] CENELEC working group SC 9XA Communication, signaling and processing systems of Technical Committee CENELEC TC 9X, *European Standard Railway Applications: Software for Railway Control and Protection Systems*, prEN 50128:1995 E edition.

[3] S.J. Clarke and J. McDermid, “Software Fault Trees and Weakest Preconditions: A Comparison and Analysis,” *Software Eng. J.*, July 1993.

[4] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna, “A Graphical Interval Logic for Specifying Concurrent Systems,” *ACM Trans. Software Eng. and Methodology*, vol. 3, no. 2, pp. 131–165, Apr. 1994.

[5] J. Dugan, S. Bavuso, and M. Boyd, “Fault Trees and Markov Models for Reliability Analysis of Fault Tolerant Digital Systems,” *Reliability Eng. and System Safety*, vol. 39, pp. 291–307, 1993.

[6] B. Dutertre and V. Stavridou, “Formal Requirements Analysis of an Avionics Control System,” *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 267–278, May 1997.

[7] J. Górski, J. Magott, and A. Wardzinski, “Modelling Fault Trees Using Petri Nets,” Technical Report RR-126/95, The Franco-Polish School of New Information and Communication Technologies, 1995.

[8] J. Górski and A. Wardzi*acuten*ski, “Deriving Real-Time Requirements for Software from Safety Analysis,” *Proc. Eighth Euromicro Workshop Real-Time Systems*, pp. 9–14. IEEE CS Press, 1996.

[9] K.M. Hansen, “Linking Safety Analysis to Safety Requirements. Exemplified by Railway Interlocking Systems,” PhD thesis, Dept. of Information Technology, Technical Univ. of Denmark, Build. 344, DK-2800 Lyngby, Denmark, 1996.

[10] M.R. Hansen and Z. Chaochen, “Duration Calculus: Logical Foundations,” *Formal Aspects of Computing*, 48 p. 1997, to appear.

[11] F. Jahanian and A.K-L. Mok, “Safety Analysis of Timing Properties in Real-Time Systems, *IEEE Trans. Software Eng.*, vol. 12, no. 9, pp. 890–904, Sept. 1986.

[12] J.C. Laprie, “Dependable Computing: Concepts and Terminology,” D.R. Avresky, ed., *Hardware and Software Fault Tolerance in Parallel Computing Systems*. Chester, U.K.: Ellis Horwood, 1992.

[13] P.A. Lee and T. Anderson, *Fault Tolerance, Principles and Practice.* Springer-Verlag, 1990.

[14] N.G. Leveson, “Software Safety in Embedded Computer Systems,” *Comm. ACM*, vol. 34, no. 2, pp. 34–46, 1991.

[15] N.G. Leveson, S.S. Cha, and T.J. Shimeall, “Safety Verification of Ada Programs Using Software Fault Trees,” *IEEE Software*, pp. 48–59, July 1991.

[16] S. Liu and J.A McDermid, “A Model-Oriented Approach to Safety Analysis Using Fault Trees and a Support System,” *J. Systems Software*, vol. 35, pp. 151–164, 1996.

[17] Z. Liu, A.P. Ravn, E.V. Sørensen, and C. Zhou, “A Probabilistic Duration Calculus,” H. Kopetz and Y. Kakuda, eds., *Responsive Computer Systems*, vol. 7 of *Dependable Computing and Fault-Tolerant Systems*, pp. 29–52. Springer Verlag, 1993.

[18] D.G. Luenberger, *Introduction to Dynamic Systems. Theory, Models & Applications*. John Wiley & Sons, 1979.

[19] D.A. Mackall, “Development and Flight Test Experiences with a Flight-Crucial Digital Control System,” Technical Report Techni-

cal Paper 2857, NASA, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, Calif., 1988.

[20] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer-Verlag, 1992.

[21] B. Moszkowski, “A Temporal Logic for Multi-Level Reasoning about Hardware. *Computer*, vol. 18, no. 2, pp. 10–19, 1985.

[22] D.S. Nielsen, “The Cause/Consequence Diagram Method as a Basis for Quantitative Accident Analysis. Technical Report Risö-M-1374, Electronics Dept., Risø, 1971.

[23] Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission, Washington, D.C., NUREG/CR-2300, *PRA Procedures Guide*, 1982.

[24] E.-R. Olderog, A.P. Ravn, and J.U. Skakkebæk, “Refining System Requirements to Program Specifications,” C. Heitmeyer and D. Mandrioli, eds. *Formal Methods in Real-Time Systems*, Trends in Software-Engineering, ch. 5, pp. 107–134. John Wiley & Sons, 1996.

[25] A.P. Ravn, “Design of Embedded Real-Time Computing Ssystems,” Technical Report ID-TR:1995-170, Dept. of Computer Science, Technical Univ. of Denmark, DK-2800 Lyngby, Denmark, 1995.

[26] A.P. Ravn, H. Rischel, and K.M. Hansen, “Specifying and Verifying Requirements of Real-Time Systems,” *IEEE Trans. Software Eng.*, vol. 19, no. 1, pp. 41–55, Jan. 1993.

[27] *Safety Management Requirements for Defence Systems, Defence Standard 00-56*. Kentigern House, Glasgow G2 8EX, Dec. 1996.

[28] *Procurement of Safety Critical Software in Defence Equipment,* Interim Defence Standard 00-55, Kentigern House, Glasgow G2 8EX, Aug. 1997.

[29] S. Subramanian, R.V. Vishnuvajjala, R. Mojdebakhsh, W.T. Tsai, and L. Elliott, “A Framework for Designing Safe Software Systems,” *Proc. 19th Int’l Computer Software and Applications Conf., COMPSAC’95*, pp. 409–414. IEEE CS Press, 1995.

[30] J.R. Taylor, “A Background to Risk Analysis,” vol. 2. technical report, Risø, Elektronics Dept., Risø National Laboratory, Dk-4000 Roskilde, Denmark, 1979.

[31] Trafiksikkerhed, Trafikstyring, DSB. *Sikkerhedsreglement af 1975 (SR)*, 1975, in Danish.

[32] U.S. Nuclear Regulatory Commission, *Fault Tree Handbook*, NUREG-0492, Washington, D.C., Jan. 1981.

[33] C. Zhou, C.A.R. Hoare, and A.P. Ravn, “A Calculus of Durations,” *Information Proc. Letters*, vol. 40, no. 5, pp. 269–276, Dec. 1991.

Kirsten M. Hansen holds an MSc degree (1989) and a PhD degree (1996) in software engineering from the Technical University of Denmark. Since 1995 she has been employed at the Danish National Rail Agency working with safety validation of interlocking systems.

Anders P. Ravn holds an MSc degree in computer science and mathematics (1973) from the University of Copenhagen and was awarded the DrTech degree by the Technical University of Denmark in 1995. He is a reader in the Department of Information Technology at the Technical University of Denmark. His research interests center on application of formal methods in software engineering for embedded real-time systems. He is a member of the IEEE and the IEEE Computer Society.

Victoria Stavridou holds a BSc degree in electronic computer systems and an MSc degree in the assessment of computer aided logic design, both from the University of Salford, U.K. as well as a PhD degree in equational specification and verification of digital systems from the University of Manchester. She was a reader in computer science at Queen Mary and Westfield College, University of London, until she recently joined the Computer Science Laboratory at SRI International, Menlo Park, California, as a senior computer scientist. Her research interests include safety critical systems, formal methods, and dependability. She has written extensively in these areas. She is a member of the IEEE and the IEEE Computer Society.