

IEEE Transactions on Software Engineering, 1998.

From Safety Analysis to Software Requirements

**Korsten M. Hansen,
Anders P. Ravn,
Victoria Stavridou**

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

<http://dslab.konkuk.ac.kr>

Contents

1. Introduction
2. Fault Trees
3. Duration Calculus
4. Fault Tree Semantics
5. Software Safety Requirements
6. Example
7. Concluding Remarks

1. Introduction

- Nancy Leveson advocates that
 - the use of system safety analysis techniques to derive system safety constraints which must be satisfied by software requirements.
 - the software requirements must be formalized in order to raise confidence in the verification.
- This paper extended these ideas by
 - demonstrating how fault trees resulting from safety analysis can be interpreted directly as requirements.
 - linking 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.

1. Introduction

- Common semantic model = common framework
 - using state variables, denoting functions of time, as in conventional dynamic system theory
 - The properties which we can model are relations among time varying states.
 - To specify such relations we use a real-time interval logic, the duration calculus.
- But, engineers are unfamiliar with formal specification.
- The underlying dynamic systems framework and the ability to illustrate duration calculus formulas by timing diagrams has helped to overcome the problem.

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.
- A partial order semantics of fault trees: [8]
- A petri net semantics of fault trees: [7]
- A modal μ -calculus semantics of fault trees: [1]
- A formalizing of structure of systems using ordinary set theory: [16]
- Fault trees as a program verification(assessing) techniques: [14,15]

2. Fault Trees

- Fault tree analysis [32] is a deductive safety analysis technique which is applied during the design phase.
 - 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 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.

2. Fault Trees



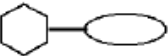


X	A node denoting a failure X, possibly resulting from a combination of basic failures (nodes).
	AND-gate—the failure in the top node occurs only when all the failures in the children nodes occur.
	OR-gate—the failure in the top node occurs only when one or more of the failures in the children nodes 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.

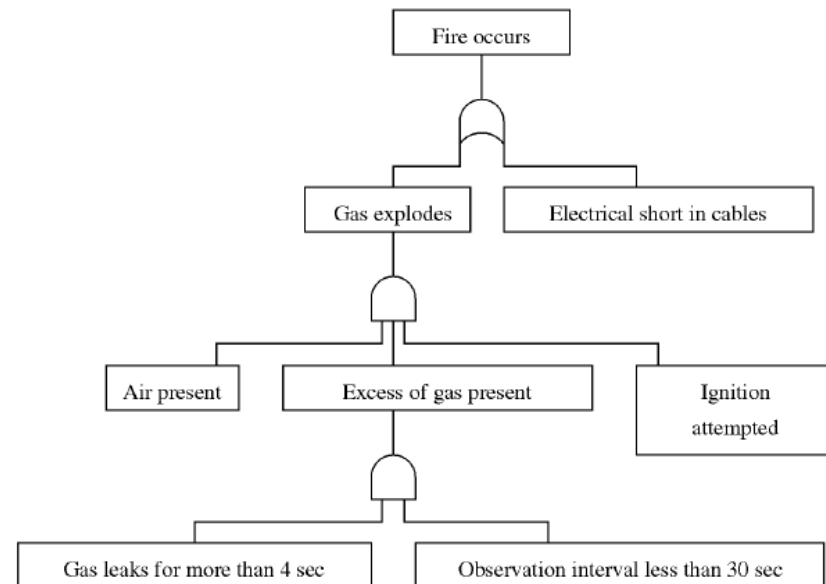


Fig. 2. Fault tree for a gas burner.

- The problem with this tree is that it allows several different interpretations.
- Section 3 presents a real-time interval logic, the duration calculus [33], to make such ambiguous interpretations precise.

3. Duration Calculus

- Formalization of fault tress
 - AND- , OR- gates : fairly straightforward – Boolean connectivities of Propositional logic
 - Events : not obvious
 - In some cases, correspond to
 - state transition
 - state occurrence
 - time of occurrence
 - A common thread in this paper 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.
 - Using the duration calculus

3. Duration Calculus

- The first step in formalizing statements about a system is to construct a system model.
- Time-domain model [18]
 - A system is described by a collection of states which are functions of Time.
- To formalize the first statement, that gas leaks for more than 4 sec, we use the following Boolean valued states,
$$\text{Gas, Flame} : \text{Time} \rightarrow \{0, 1\}$$
which express the presence of gas and flame as functions of time.

3. Duration Calculus

- Statements about a system are expressed by constraining states over time.

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

- When we consider a bounded time interval $[b, e]$, we can measure the duration of Leak within the interval by

$$\int_b^e Leak(t) dt$$

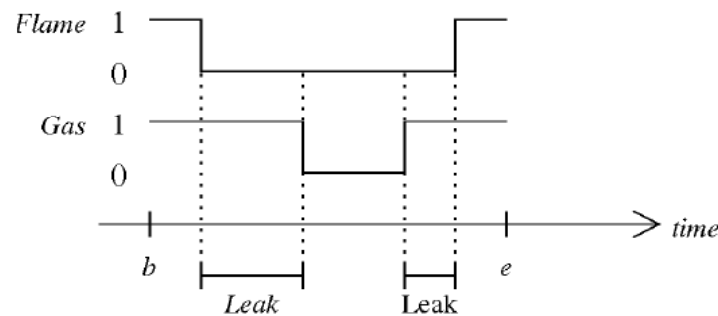


Fig. 3. Timing diagram for *Leak*.

3. Duration Calculus

- Gas leaks for more than 4 sec : $\int Leak > 4$
- $\int 1 = 1$, abbreviated by l .
- "The considered time interval is not longer than 30 seconds and gas leaks for more than 4 sec."
 $(l \leq 30) \wedge (\int Leak > 4)$
- $\lceil Ignition \rceil : \int Ignition = l \wedge l > 0$
- Subinterval property: $D_1 ; D_2$
- Somewhere : \diamond , Everywhere : \square

3. Duration Calculus

- 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."

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

$$\neg S = (l \leq 30) \wedge (\int Leak > 4)$$

- 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 $\square S$.

3.1 Duration Calculus, Summary

- Refer to [10], [33]

4. Fault Tree Semantics

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

4.1 Leaves

4.2 Intermediate Nodes

4.3 Edges

4.4 Gates

4.5 Trees

4.1 Leaves

- The leaves in a fault tree are called events, but
 - in safety analysis, often meaning the occurrence of a specific system state
 - in software engineering, meaning a transition between two states
 - we use the term event in this article, we mean a state transition (the software engineering interpretation of an event).
- We interpret a leaf node of a fault tree as a duration calculus formula.
 - 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 : $\lceil \neg P \rceil; \lceil P \rceil$
 - elapse of a certain time, i.e., $\int (30 + \varepsilon)$
 - a threshold of some duration, i.e., $\int P \leq 4 \times \varepsilon$

4.1 Leaves

- 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.

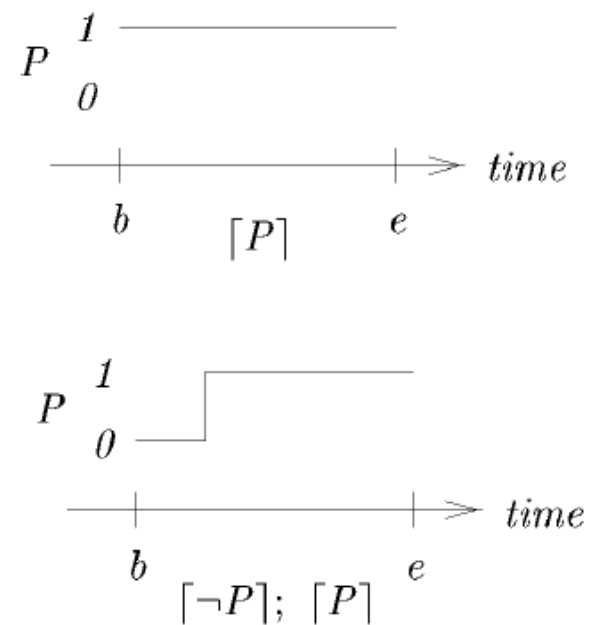


Fig. 4. Timing diagrams for some basic observations.

4.2 Intermediate Nodes

- Intermediate nodes are merely names of the corresponding subtrees.
- 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.

4.3 Edges

- We then define the semantics of A to be

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

- as logical identity, meaning that the system failure A occurs when the failure B occurs. (pessimistic interpretation)

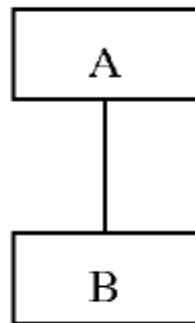


Fig. 5. Fault tree with no gates.

- Optimistic interpretation :

$$A \Rightarrow B$$

- A system failure may be avoided, if the operator intervenes fast enough, has enough luck, etc.

4.4 Gates

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

- AND

- $A \stackrel{def}{=} B_1 \wedge \dots \wedge B_n$

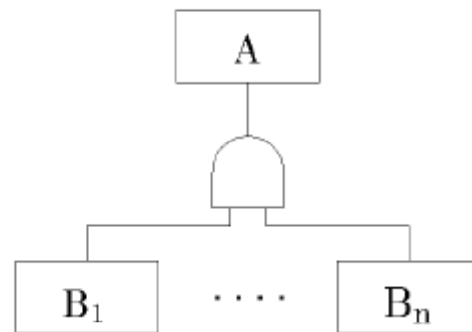


Fig. 6. Fault tree with AND-gate.

- OR

- $A = B_1 \vee \dots \vee B_n$

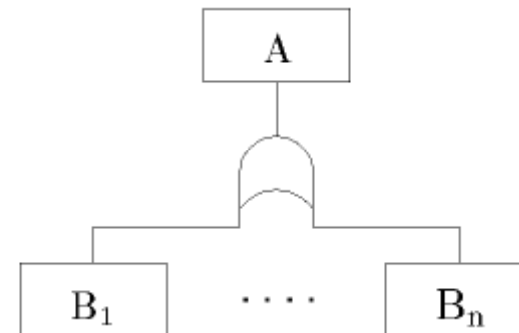


Fig. 7. Fault tree with OR-gate.

4.4 Gates

- INHIBIT

- $A \stackrel{def}{=} B_1 \wedge \dots \wedge B_n$
- B_n : a condition

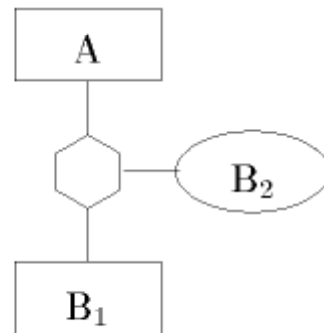


Fig. 8. Fault tree with INHIBIT-gate.

- EXCLUSIVE OR

- $A \stackrel{def}{=} (B_1 \wedge \neg(B_2 \vee \dots \vee B_n))$
 \vee
 \vdots
 \vee
 $(B_n \wedge \neg(B_2 \vee \dots \vee B_{n-1}))$

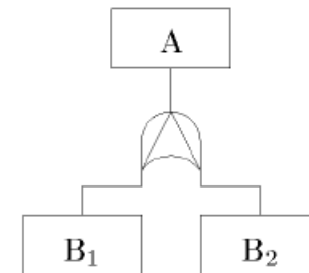


Fig. 9. Fault tree with EXCLUSIVE OR-gate.

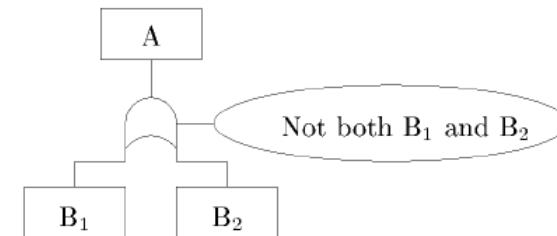


Fig. 10. Fault tree with OR-gate and a condition.

4.4 Gates

- PRIORITY AND
 - $A = B_1 \wedge \diamond(B_2 \wedge \diamond(B_3 \wedge \dots \wedge \diamond B_n) \dots)$

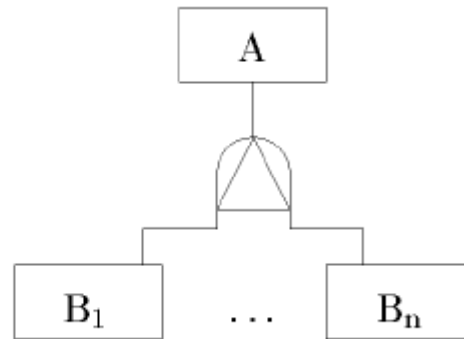


Fig. 11. Fault tree with PRIORITY AND-gate.

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

- 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.

5. Software Safety Requirements

- For each fault tree in which the root is interpreted as S , the safety commitment which the system should implement is

$$\square \neg S$$

- If we have n fault trees, the safety commitment is

$$\square \neg S \wedge \dots \wedge \square \neg S$$

i.e., the system should ensure that no top event in any fault tree ever holds.

5.2 AND-Gates

- The semantics is $A = B_1 \wedge \dots \wedge B_n$

- The safety commitment is $\neg A$
 - corresponds to specifying that the components never satisfy their duration formulas at the same time, i.e.,

$$\square \neg (B_1 \wedge B_2 \wedge \dots \wedge B_n)$$

- One way to implement this is

$$\square \neg B_1 \vee \square \neg B_2 \vee \dots \vee \square \neg B_n$$

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

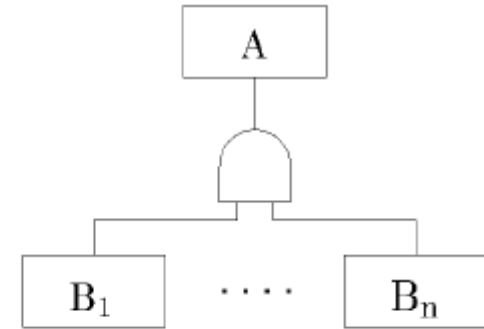


Fig. 6. Fault tree with AND-gate.

5.2 AND-Gates

- 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.

- If B_1 is uncontrollable,

$$\square \neg(true \wedge B_2 \wedge \dots \wedge B_n) \equiv \square \neg(B_2 \wedge \dots \wedge B_n)$$

- Software engineer should arrive at a conjunction of B_i s which can be used in the design.
- Otherwise, we must conclude that the system is inherently unsafe.

5.3 OR-Gates

- The semantics is $A = B_1 \vee \dots \vee B_n$
- The safety commitment is $\neg A$
 - expresses that the system only satisfies its safety commitments if all its components satisfy their local safety commitments.

$$\square \neg (B_1 \vee B_2 \vee \dots \vee B_n) \equiv \square \neg B_1 \wedge \square \neg B_2 \wedge \dots \wedge \square \neg B_n$$

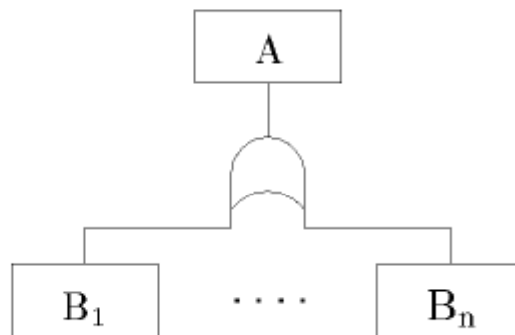


Fig. 7. Fault tree with OR-gate.

5.3 OR-Gates

- 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.
- To make the design team as a whole aware of the assumption that $\Box \neg B_1$ is *true*.
 - $Asm \Rightarrow Com$ has been weakened to
 - $Asm \wedge \Box \neg B_1 \Rightarrow Com$
 - The software engineer should alert the appropriate persons to the fact that the system requirements have been weakened.

5.4 PRIORITY AND-Gates

- The semantics is $A = B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \dots \wedge \Diamond B_n) \dots)$
- The safety commitment is $\neg A$

$$\begin{aligned} \Box \neg (B_1 \wedge \Diamond(B_2 \wedge \Diamond(B_3 \wedge \dots \wedge \Diamond B_n) \dots)) \\ \equiv \Box \neg B_1 \vee \Box \neg B_2 \vee \dots \vee \Box \neg B_n \end{aligned}$$

- This may either be done by making the implementation such that
 - the B_i s do not occur in the specified order or
 - one of the B_i s does not occur at all, i.e., that B_1 is holds.
- If one of the B_i is uncontrollable,
 - the same as the previous interpretation.

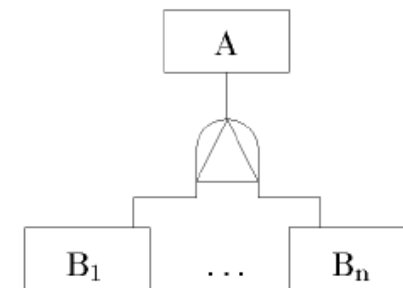


Fig. 11. Fault tree with PRIORITY AND-gate.

6. Example

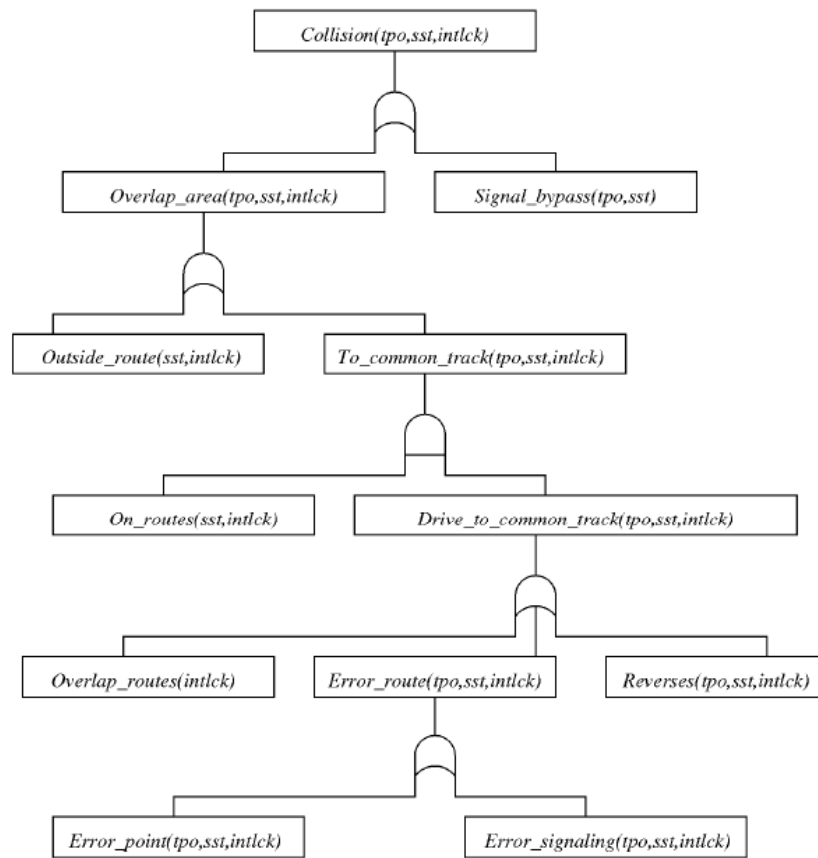


Fig. 12. Fault tree for collision.

- $Safe \stackrel{def}{=} Asm \Rightarrow Safe_com$
- $Asm \stackrel{def}{=} \square \neg Signal_bypass(tpo, sst)$
 $\wedge \square \neg Outside_route(sst, intlck)$
 $\wedge \square \neg (On_routes(tpo, sst, intlck)$
 $\wedge Reverses(tpo, sst, intlck))$
- $Safe_com \stackrel{def}{=} \square \neg (On_route(sst, intlck)$
 $\wedge Overlap_routes(intlck)$
 $\wedge (Error_point(tpo, sst, intlck)$
 $\vee Error_signaling(top, sst, intlck)))$

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.

7. Concluding Remarks

- 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.
- In the development of safety critical systems, this means that the software may be proven to satisfy the system safety requirements.
- We have extended the ideas presented in this paper in [9] where we also have shown how to derive safety requirements from event trees and cause-consequence diagrams.