A Model-Oriented Approach to Safety Analysis Using Fault Trees and a Support System

Shaoying Liu
Hiroshima City University, Japan

John A. McDermid
University of York, United Kingdom

Fault-tree technique has been used in industry for safety analysis of safety critical systems for decades. It can be used for analyzing the safety of both software and hardware. However, there are many problems with ensuring the internal consistency and validity of fault trees constructed because of the absence of appropriate approaches for using fault tree techniques. This article describes a model-oriented approach for safety analysis using fault trees. It advocates that a safety analysis of a system should start with a systematic study of the physical model of the system and, as a result, construct a system safety model based on the physical model. Further safety analysis of the system — based on the system safety model by constructing fault trees — is carried out. The internal consistency and consistency with the system safety model of these fault trees must be ensured. A prototype called FTSS (Fault Tree Support System) has been implemented by the ASAM (A Safety Argument Manager) project to support this approach, and its functionality is described. © *1996 by Elsevier Science Inc.*

1. INTRODUCTION

Safety is a very important property of a safety critical system whose failure may lead to grave dangers to human life and property. It is a measure of the continuous delivery of service free from occurrences of catastrophic failures. Safety analysis of such a system has been recognized as a compulsory process for ensuring its safety in deployment (Roberts, 1981; Leveson and Harvey, 1983a, 1983b; HSE, 1987).

Fault-Tree Analysis has been the most popular technique for safety analysis of overall systems (Alesso and Benson, 1980; Pate-Cornell, 1984; Villemeur, 1992). It may be applied both to hardware and software (Perkusich et al., 1994; Zhuang and Xie, 1994). In the application of hardware safety analysis, fault trees are traditionally derived directly, based on the physical model of the system concerned (a physical model of a physical system includes the structure, operation principle, properties of components and relationships between components of the physical system), which reflects potential causes of some failure ("top event") (Lambert, 1973). Each event in such a fault tree describes some failure involving some physical components, and high-level events can be caused by various combinations of lower-level events, with the logical connectives that are called AND gate, OR gate, EXCLUSIVE-OR gate, PRIORITY-AND gate (which generates a true output only if the input events are true in the order specified in an associated 'condition') and INHIBIT gate (which generates a true output if some input event in the system is true, and some external 'conditional event' has occurred). Leveson and her colleagues were the first to apply fault-tree technique to the safety analysis of software (Leveson and Harvey, 1983a, 1983b). The goal of software safety is the avoidance of system safety failures that are caused by a software error or are detected and handled by software procedures. A

Address correspondence to Dr. Shaoying Liu, Hiroshima City University, Japan.

J. SYSTEMS SOFTWARE 1996; 35:151–164
© 1996 by Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010

0164-1212/96/$15.00
SSDI 0164-1212(95)00094-H

which may result in casualties or serious consequences (Leveson and Harvey, 1983a). The aim of safety analysis of software is therefore to find all those errors which can cause safety failures. In this application, fault trees are derived directly from the software (programs) based on the semantics of each statement (e.g., sequential statement, conditional statement, iteration statement, etc.).

The common characteristic of these two kinds of applications of fault-tree technique is to construct fault trees directly, based on the physical model (hardware or software). However, this approach may cause the constructed fault trees to be unsatisfactory. The reasons for this limitation are threefold.

The first is a lack of understanding of the physical model. The physical model of a real safety critical system is usually very complex, and the task of constructing a safety case is to map the physical model to safety requirements. Therefore, without a systematic study of the physical model to understand various kinds of relationships between many different components of the physical system, it is very difficult to construct a satisfactory safety case expressed by fault trees.

The second reason is the absence of guidelines for constructing fault trees based on a physical model. That is, how and why a fault tree is built, in general, based on a physical model, is not yet clear. There might be particular rules for a particular physical model, but for a general approach of safety analysis using fault trees, it will be very helpful if there are general guidelines to follow. In this case, when a safety case is undertaken, a particular rule of constructing fault trees can be produced by applying the general guidelines to a particular model.

The third reason is the lack of a firm foundation to check whether the constructed fault trees are consistent with the knowledge that the physical model represents. A safety case expressed by fault trees for a real safety critical system is usually huge. It is therefore necessary to check whether this safety case violates the knowledge which the physical model represents. However, without expressing this knowledge precisely and properly, it will be difficult to carry out consistency checking, especially when this checking is expected to be performed automatically by a computer system.

In order to overcome these difficulties, we propose a *model-oriented approach for safety analysis* (MOASA for short) using fault trees in this paper. It advocates that safety analysis of a system should start with a systematic study of the physical model of the system and, as a result, construct a system safety model that contains the basic knowledge concerning the safety of the physical system. Then, further safety analysis of the system is done by constructing fault trees based on the system safety model, and the consistency of these fault trees with the system safety model must be ensured.

This approach is derived from our research and experience on the ASAM[1] project and has been exercised with a case study (McDermid and Liu, 1993). A fault tree support system (FTSS) has been prototyped using the Object-Oriented Common Lisp (CLOS) on Macintosh. This prototype uses ONTOS object oriented database for storing system safety models and safety cases.

The remainder of this paper is organized as follows. Section 2 describes the strategy of MOASA. Section 3 discusses the principles of building a system safety model from a physical model. Section 4 addresses the issue of how to construct a fault tree to represent a safety case. Section 5 describes the criteria for checking the consistency of fault trees against the system safety model. Section 6 describes the functionality of FTSS. Finally, Section 7 presents the conclusions and points out further research.

2. STRATEGY OF MOASA

In MOASA, a safety analysis includes three steps: the construction of a system safety model, the construction of safety cases, and consistency checking of the safety cases. This process is expressed in Figure 1.

The construction of a system safety model should start with a careful study of a physical model that is an abstract representation of a physical system. A physical system can be both hardware, such as an aircraft, a nuclear power station, or a hospital, etc., and software such as a Pascal program. A physical model is usually expressed in a language (e.g., an aircraft is described by a design graph and the associated document of the design; a program is expressed in a program language and its formal semantics). The aim of the construction of a system safety model based on a physical model is to discover as much knowledge that is useful for safety analysis as possible (e.g., structure of the physical model, relationships between components of the physical model, etc.), to express this knowledge precisely and properly, and to provide a firm foundation

[1] A Safety Argument Manager. This is a three-year project sponsored by SERC and DTI.

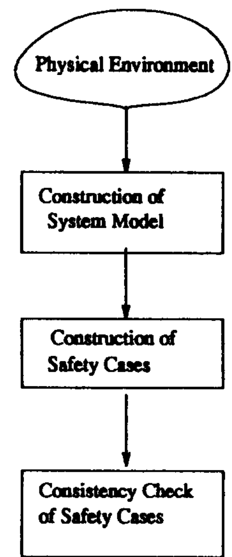

Figure 1. Process of safety analysis using MOASA.

for the construction and consistency checking of safety cases. The contents and structure of a system safety model will be described in next section.

Since a system safety model is expected to represent all the basic knowledge useful for safety analysis of the system concerned (e.g., relationships between the components of an aircraft, possible events happening on an aircraft, etc.), further safety analysis should be carried out based on this system safety model. As a result of this analysis, safety cases expressed by fault trees are produced. In fact, a safety case is a particular kind of safety knowledge of a physical system but focuses on the causal relationships between events happening on the physical system or components of the physical system.

Therefore, a constructed safety case must be consistent with the system safety model so that the safety knowledge does not violate the background knowledge of a physical system. For example, if a safety case of an aircraft describes a fact "high temperature of the engine" is caused by the event "cooling system is broken", while in the system safety model there is no relationship between the engine and the cooling system described, then this safety case will be inconsistent with the system safety model. This inconsistency does not necessarily mean the safety case is not reasonable but may indicate that the system safety model does not capture enough information useful for further safety analysis. Therefore, if such an inconsistency occurs, both the system safety model and the safety case must be examined, and the system safety model may be enriched with new information involved in the safety case.

Three issues especially important in this approach include how to build a system safety model, how to derive a safety case from this system safety model, and how to check whether a safety case is internally consistent and consistent with this system safety model. They are discussed in the following sections, respectively.

3. CONSTRUCTION OF SYSTEM SAFETY MODEL

We want a system safety model to store sufficient basic knowledge concerning the safety of a physical system that should be part of the system model of the physical system. By system model we mean a database for storing various kinds of pieces of knowledge about the physical system, including some relevant and some other irrelevant to the safety of the physical system. In this section, we try to work out a general guideline to produce a system safety model from a physical model that may be applied to any particular case. To do this, we first need to formally define the concept of system safety model.

3.1 Definition of System Safety Model

Definition 3.1 [system safety model]. A system safety model is a quadruple: (E_n, R, E_v, P), where E_n is the collection of all entities, R is a set of binary relations between entities, E_v is a set of events, and P is a map from E_v to $[0,1]$.

An entity represents an object in the real world (e.g., a person, a computer, a cup, etc.). It usually has attributes to describe its interesting details (e.g., name, age, sex, etc.). Each attribute is a value of some type (e.g., natural number, real number, boolean number, etc.). The set of entities E_n in a system safety model must include all the components, substances, and materials, etc., associated with the physical system when it is in operation and can usually be obtained from a physical model of the physical system because the physical model is normally a representation of a real physical system.

A binary relation in R is a collection of pairs, each of which consists of two entities. Since the relations existing in the system safety model are expected to contribute to safety case construction, they must reflect the relationships between components (entities) of the corresponding physical system,

based on which causal relationships between events involving these components can be described. Experience suggests that the following relations are necessary to construct from a physical system (hardware or software) if applicable:

1. *Physical Connection* relation
2. *Logical Connection* relation
3. *Contain* relation
4. *Control* relation
5. *Input* relation (materials or information)
6. *Output* relation (materials or information)
7. *Get-Information-From* relation
8. *Process* relation (process materials or information)
9. Other relations

Two entities e_1 and e_2 have a *Physical Connection* relation if they are connected physically in a physical system, have a *Logical Connection* if they are connected logically, have a *Contain* relation if e_1 contains e_2, have a *Control* relation if e_1 controls e_2, have an *Input* relation if e_1 is an input of e_2, have an *Output* relation if e_1 is an output of e_2, have a *Get-Information-From* relation if e_1 gets information from e_2, have a *Process* relation if e_1 processes e_2.

An event in E_v represents a condition that might cause failure of the system concerned (e.g., the engine of an aircraft is overheated). It consists of three components: name, content, and a set of entities. The name is an identifier used for identifying the event. The content is a statement to express what the event is. It must state precisely what the fault is and should be as complete a description as required for a third part to understand what is happening. The set of entities presents all the entities involved in the event (e.g., 'engine' and 'aircraft' are two entities involved in the event: the engine of an aircraft is overheated). Let $e \in E_v$, we use *e.name*, *e.content*, and *e.entities* to represent the name, content, and the set of entities of the event e.

Experience suggests that the content of an event is usually derived either from the historical experience of the same class systems (e.g., when a new aircraft is designed, an event may be: the engine fails in operation. The idea of this event comes from the historical experience of some aircraft crash due to the failure of engine) or from some scientific principles (e.g., the idea of the event: cracks in the pistons in an internal combustion engine is caused by overheating, could come from the relevant material science).

The map P records the probabilities of events in E_v. The probability of an event indicates the degree of the possibility to which the event happens. It is usually derived from the statistics based on historical experience of the system under consideration.

In order to help obtain the safety knowledge accurately and efficiently from the expertise of physical systems, we provide a Schema Editor in our support system to allow the user to define and edit entity types and relationship types, to define the attributes for each, and to draw entity-relationship diagrams. A finished schema document is transformed into the internal representation stored in the system safety model.

3.2 Examples

An example is given below to help illustrate the principle of constructing a system safety model from a physical model.

Figure 2 presents a physical model describing a control mechanism of a drain valve in a PES (Programmable Electronic Systems). This PES is for controlling and protecting a plant manufacturing explosive pentaerythritol tetranitrate (PETN). This substance is made by the nitration of pentaerythritol (PE) in a batch process. The reaction is exothermic and must be controlled to prevent an excessive temperature rise that would cause decomposition of the PETN and the production of toxic fumes. If the condition is not brought under control, there is a risk of fire that might spread to parts of the building containing finished PETN.

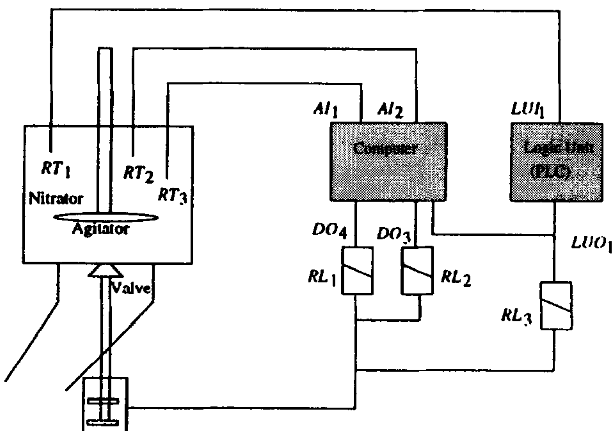

Figure 2. Control mechanism of drain valve. (*AI* = Analogue input to computer; *LUI* = Logic unit input; *DO* = Digital output to computer; *RL* = Resistance logic; *RT* = Resistance thermometer.

J. SYSTEMS SOFTWARE 155
1996; 35:151-164

A predetermined quantity of nitrating acid is run from an acid head tank into a nitrator vessel. A preweighed quantity of PE is then fed into the vessel from a weigh hopper by a vibrating feeder. The vessel is cooled by the circulation of chilled water on its outside surface, and the contents are continuously stirred (the whole model of this system is presented in McDermid and Liu, (1993) which is not necessary for our purpose of demonstrating how to build a system safety model in this example).

After a present time, and provided that the temperature has fallen below a present value, the contents of the nitrator are discharged through the drain valve and diverted to the nitration filter for further processing.

The control computer receives signals from plant sensors (e.g., RT_1 and RT_2, etc.) and computer control outputs accordingly, but where a sensor is monitoring a critical parameter, the sensor is duplicated, and the signal from the duplicate is fed to a Programmable Logic Controller (PLC). Operations, such as valve open, depending on the critical parameters, are controlled mainly by the control computer supplemented by back-up controller PLC.

The failure of the valve open at a proper time may cause the nitrator to overflow, which is dangerous to the whole system. Understanding how this failure can happen is part of the whole safety requirements analysis for the whole PES system.

In order to do this, we first build a system safety model $L_m = (E_n, R, E_v, P)$ from the physical model given in Figure 2 and the operation principle of the PES system described previously as follows:

1. $E_n = \{AI_1, AI_2, LUI_1, LUO_1, DO_3, DO_4, RL_1, RL_2, RL_3, RT_1, RT_2, RT_3, Computer, Logic\text{-}unit, Nitrator, Agitator, PE, Valve\}$. This entity set includes all the components (e.g., RT_1, RT_2, etc.), substance (e.g., *PE*), and information (e.g., DO_3, DO_4, etc.) which are used in the physical model.
2. $R = \{$*Physical-Connection, Contain, Control, Input, Output, Process, Get-Information-From*$\}$, where each of these relations is constructed as follows:

   - *Physical-Connection* $= \{(RT_1, Computer), (RT_2, Computer), (RT_3, Logic\text{-}unit), (Agitator, PE), (Nitrator, Valve), (Computer, RL_1), (Computer, RL_2), (Computer, RL_3), (Logic\text{-}unit, RL_3), (RL_1, Valve), (RL_2, Valve), (RL_3, Valve)\}$
   - *Contain* $= \{(Nitrator, PE)\}$.
   - *Control* $= \{(Computer, RL_1), (Computer, RL_2), (Logic\text{-}unit, RL_3), (RL_1, Valve), (RL_2, Valve), (RL_3, Valve)\}$.
   - *Input* $= \{(AI_1, Computer), (AI_2, Computer), (LUI_1, Logic\text{-}unit), (LUO_1, Computer)\}$.
   - *Output* $= \{(DO_3, Computer), (DO_4, Computer), (LUO_1, Logic\text{-}unit)\}$.
   - *Process* $= \{(Computer, AI_1), (Computer, AI_2), (Logic\text{-}unit, LUI_1), (Nitrator, PE)\}$.
   - *Get-Information-From* $= \{(RT_1, Nitrator), (RT_2, Nitrator), (RT_3, Nitrator)\}$.

   These relations records all the relationships between entities in E_n. They show, for example, RT_1 is physically connected to *Computer*, *Nitrator* contains PE, *Computer* controls RL_1 and RL_2, AI_1 is an input to *Computer*, DO_3 is an output of *Computer*, *Computer* processes AI_1 and RT_1 gets information from *Nitrator* and so on.
3. $E_v = \{(e_1$, '*Valve* fails to open', $\{Valve\})$,
   $(e_2$, 'RL_1 is out of control', $\{RL_1\})$,
   $(e_3$, 'RL_2 is out of control', $\{RL_2\})$,
   $(e_4$, 'RL_3 is out of control', $\{RL_3\})$,
   $(e_5$, 'RL_1 crashes', $\{RL_1\})$,
   $(e_6$, '*Computer* is out of control', $\{Computer\})$,
   $(e_7$, 'RL_2 crashes', $\{RL_2\})$,
   $(e_8$, 'RL_3 crashes', $\{RL_3\})$,
   $(e_9$, '*Logic-unit* is out of control', $\{Logic\text{-}unit\})$,
   $(e_{10}$, 'RT_1 is out of control', $\{RT_1\})$,
   $(e_{11}$, 'RT_2 is out of control', $\{RT_2\})$,
   $(e_{12}$, '*Computer* crashes', $\{Computer\})$,
   $(e_{13}$, 'RT_1 crashes', $\{RT_1\})$,
   $(e_{14}$, 'PE is high', {PE}),
   $(e_{15}$, 'RT_2 crashes', $\{RT_2\})$,
   $(e_{16}$, 'RT_3 is out of control', $\{RT_3\})$,
   $(e_{17}$, 'RT_3 crashes', $\{RT_3\})$
   This event set records for all possible events happening to the entities given in E_n. Each event is represented by a triple, $(e_1$, '*Valve* fails to open', $\{Valve\})$ for example, where e_1, '*Valve* fails to open' and $\{Valve\}$ are the name, content, and entity set of this event, respectively.
4. $P = \{(e_1, 0.02), (e_2, 0.02), (e_3, 0.02), (e_4, 0.02), (e_5, 0.04), (e_6, 0.09), (e_7, 0.04), (e_8, 0.05), (e_9, 0.03), (e_{10}, 0.01), (e_{11}, 0.01), (e_{12}, 0.05), (e_{13}, 0.01), (e_{14}, 0.12), (e_{15}, 0.01), (e_{16}, 0.01), (e_{17}, 0.01)\}$.

This map P records the probabilities of all the events given in E_v (e.g., the event with the name e_1 happens with the probability 0.02). Precisely, the P is not a map from E_v to [0, 1], but a map from the set of names of the events in E_v to [0, 1]. Because the name of an event in E_v is unique, it can represent the corresponding event. Therefore, there is no problem for us to understand P as a map from events to probabilities.

Note that not all pieces of the knowledge represented by a system safety model is necessarily used when a fault tree is constructed, but some fault tree describing a particular safety case may use some part of the knowledge and some other fault trees may use some other parts of the knowledge.

3.3 Internal Consistency of System Safety Models

The internal consistency of a system safety model is defined as follows:

Definition 3.2 [internal consistency of system safety model]. Let $L_m = (E_n, R, E_v, P)$ be a system safety model. If it satisfies the conditions:

- $\forall_{e \in E_v} \cdot e.entities \subseteq E_n$
- $E_v = dom(P)$
- $\forall_{r \in R} \cdot dom(r) \cup rng(r) \subseteq E_n$,

then the system safety model L_m is internally consistent, where $dom(P)$ and $dom(r)$ denote the domains of the map P and the relation r, respectively, $rng(r)$ represents the range of r.

The internal consistency of a system safety model guarantees three things. The first is that all the entities of any event in E_v belong to E_n. The second is that P maps every event in E_v to its probability (that is, every event must be given a probability). The third is that all entities occurring in every relation in R belong to E_n. The reason for guaranteeing these three conditions for a system safety model is that a system safety model should be a close system in the sense that every entity associated with events or relations recorded in the system safety model must exist in the system safety model. In this case, a system safety model expresses a consistent and sensible knowledge of the corresponding physical system (e.g., without recording '*Valve*' as an entity in E_n, it is hard to understand whether the event '*Valve* fails to open' is sensible within the system safety model).

Applying this definition to the system safety model defined in Section 3.2, we can see that this system safety model is internally consistent.

4. CONSTRUCTION OF FAULT TREES

Since a system safety model is the result of careful study of a physical model, it usually records sufficient knowledge of the physical system which is useful for further safety analysis (safety case construction).

Fault trees, which express safety cases of a physical system, should be built based on the system safety model (but not depend on it completely because some more knowledge must be obtained from the expertise of this physical system when a safety case is built). Before describing the principle of constructing fault trees based on a system safety model, we first need to define fault trees and to introduce the graphical notation which is used in fault trees supported by the system FTSS.

4.1 Definition of Fault Trees

Definition 4.1 [fault tree]. A fault tree is a triple: (F_e, F_g, f), where F_e is the collection of events, F_g is the collection of gates, and f is a map from the union of F_e and F_g to its power set, defined as follows:

$f: F_e \cup F_g \rightarrow {}_2(F_e \cup F_g)$.

Every event in a fault tree should come from the system safety model because the system safety model is expected to be the basis for the construction of safety cases (further safety analysis). Syntactically, there are several different graphical notations to express different kinds of events. These notations include *circle*, *diamond*, *oval*, *rectangle*, and *house*, and they represent BASIC EVENT, UNDEVELOPED EVENT, CONDITIONING EVENT, INTERMEDIATE EVENT, and EXTERNAL EVENT, respectively. The detailed description of these notations is described in Figure 3.

The four components: name, content, set of entities, and the probability of each event is presented in the associated graphical notation. Note that these four components are usually inputed by means of a human-computer interface supported by FTSS. They should be the same as those of the same event in the corresponding system safety model but might be different due to input errors or other reasons (e.g., the person constructing the fault tree and the person building the system safety model are different, and their knowledge about a same event might also

J. SYSTEMS SOFTWARE 1996; 35:151-164

	Event Symbols	
	BASIC EVENT	A basic initiating fault requiring no further development.
	UNDEVELOPED EVENT	An event which is not further developed either because it is of insufficient consequence or because information is unavailable.
	CONDITIONING EVENT	Specific conditions or restrictions that apply to any logic gate (used primarily with PRIORITY AND and INHIBIT gates).
	INTERMEDIATE EVENT	A fault event that occurs because of one or more antecedent causes acting through logic gates.
	EXTERNAL EVENT	An event which is normally expected to occur
	Gate Symbols	
	AND	Output fault occurs if all of the input faults occur
	OR	Output fault occurs if at least one of the input faults occurs.
	EXCLUSIVE OR	Output fault occurs if exactly one of the input fault occurs.
	PRIORITY AND	Output fault occurs if all of the input fault occur in a specific sequence (the sequence is represented by a CONDITIONING EVENT drawn to the right of the gate).
	INHIBIT	Output fault occurs if the (single) input fault occurs in the presence of enabling condition (the enabling condition is represented by a CONDITIONING EVENT drawn to the right of the gate.

Figure 3. Fault tree symbols (a).

be different). In addition to *e.name*, *e.content*, and *e.entities*, we use *e.probability* to represent the probability of the event e. Note that we do not favor formal notation to describe the four components because otherwise, it would increase the difficulty in readability of safety cases.

A gate in a fault tree is a logical connective. There are several different graphical notations to denote different gates. They are called AND, OR, EXCLUSIVE-OR, PRIORITY-AND, and INHIBIT. The detailed explanation of these gates are presented in Figure 3. Furthermore, in order to allow a big fault tree to be drawn on separate pages, two transfer symbols are necessary. They are called TRANSFER IN and TRANSFER OUT, and are explained in Figure 4.

The map f in a fault tree describes a graphical connection relationship between events and gates. For $a, b \in F_e \cup F_g$, if $a \in f(b)$, then we call a a child node of b and b a parent node of a. In this case, both a and b can be either an event or a gate. An example of a fault tree will be given when the guideline of constructing fault trees is discussed below.

	Transfer Symbols	
	TRANSFER IN	Indicates that the tree is developed further at the occurrence of the corresponding TRANSFER OUT (e.g. on another page).
	TRANSFER OUT	Indicates that this portion of the tree must be attached at the corresponding TRANSFER IN.

Figure 4. Fault tree symbols (b).

4.2 Guidelines for Constructing Fault Trees

The process of constructing a fault tree concerning the safety of a physical system should start with the examination of the relation set R and the set of events E_v in the corresponding system safety model. The reason for this is because the relation set R is expected to describe all the possible physical relationships (e.g., RT_1 *Physical-Connection Computer*, *Nitrator Contain* PE, etc.) and logical relationships (*Computer Control* RL_1, RL_3 *Control Value* etc.), and the set of events E_v is expected to represent all the possible events involving the entities which might be used in relations in R. The task of constructing a fault tree is to work out how a top event is caused by other events (i.e., causal relationship between different events). During this course, the person who is in charge of the construction of the fault tree may need to get some knowledge from the expertise of the physical system. Careful choice of the top event is important to the success of the construction of a safety case. If it is too general, the analysis becomes unmanageable, if it is too specific the safety case does not provide a sufficiently broad view of the event. In fact, if sufficient information (i.e., the immediate cause of each event) can be obtained, then several total or partial ordering of the events can be generated, which can help select an appropriate top event.

The steps to construct a fault tree are as follows:

1. Ensure the boundaries of the system being considered are defined.
2. Ensure the "limit of resolution" to be used in the construction is defined. This is the lowest level that will be considered.
3. Establish the "top event" to be analyzed.
4. Determine the immediate causes of the "top event". These are not the basic causes but immediate causes. The immediate cause concept allows a fault tree to be developed step by step during its construction.
5. Examine the causes leading to the event in question and establish the type of gate (or basic event) required.
6. Repeat step 5 until the "limit of resolution" required is reached, and each branch of the tree terminates in basic events. If the "limit of resolution" is not reached, the branch should terminate in a diamond event.

Let us take the system safety model given in Section 3.2 as an example to demonstrate how a fault tree is constructed based on it. The physical model given in Figure 2 describes a system of processing PE (Pentaerythritol) in the Nitrator. When the process is finished, it is tested by the three resistance thermometer RT_1, RT_2, and RT_3 and decided by the computer and the logic-unit; the computer and the logic-unit will then output the signals DO_3, DO_4, and LUO_1 to RL_1, RL_2, and RL_3 to open the valve underneath the nitrator. The aim of constructing a fault tree is to analyze why the valve fails to open when the process is finished. To construct the fault tree, we first examine the relations in R built in the corresponding system safety model to find which entities have the relations possibly affecting the valve's ability to open (e.g., the entities RL_1, RL_2, and RL_3 have the relations *Physical-Connection* and *Control* with the valve). Then search the set of event E_v to see whether there is any event to express the failure of opening the valve. Finally, consult the relevant expertise of the physical system to obtain more knowledge about the causal relationship between this top event and other events, and continue this process until all leaf events are basic events that are represented by either a diamond or a circle. As a result of this process, a fault tree is constructed, as shown in Figure 5, Figure 6, and Figure 7, where N, C, E, and P in each event node indicate the name, content, set of entities, and the probability of the event following, correspondingly.

According to this fault tree, the top event is identified by e_1 and describes the fact "*Valve* fails to open" with the probability 0.02. The entity involved is *Valve*. This event is caused by one of the three events identified by e_2, e_3, and e_4, respectively, and these events are caused by other events in the fault tree. As this fault tree is quite straightforward, we do not explain it in detail here.

Note that there might be some event occurring in a fault tree that does not exist in the corresponding system safety model. This kind of event is usually produced during the consultation of the expertise of the physical system. In this case, the corresponding system safety model must be enriched with this new event and the associated entities.

5. CONSISTENCY OF FAULT TREES

There are two kinds of consistency to address. The first is the consistency of a fault tree with its syntax and semantics definition. We call this *internal consistency*. The second is the consistency with a corresponding system safety model. We call it *model consistency*. It is necessary to check whether a constructed fault tree satisfies both internal and model

J. SYSTEMS SOFTWARE
1996; 35:151-164

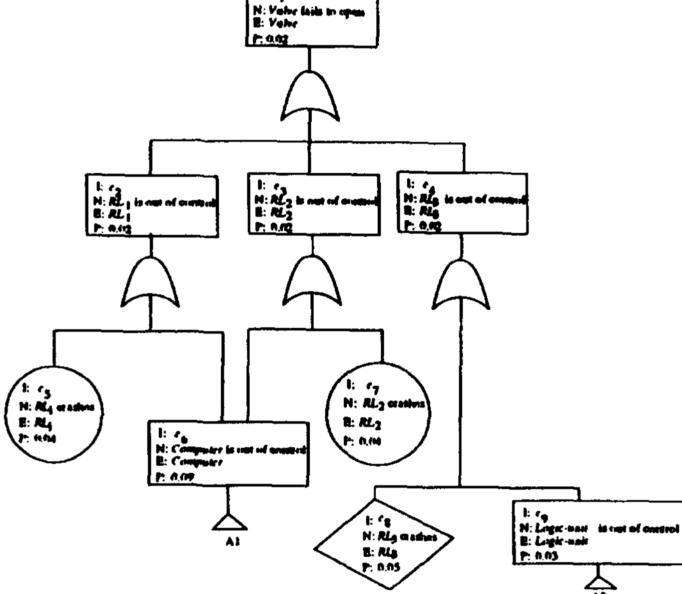

Figure 5. Fault tree about the control of drain valve (*a*).

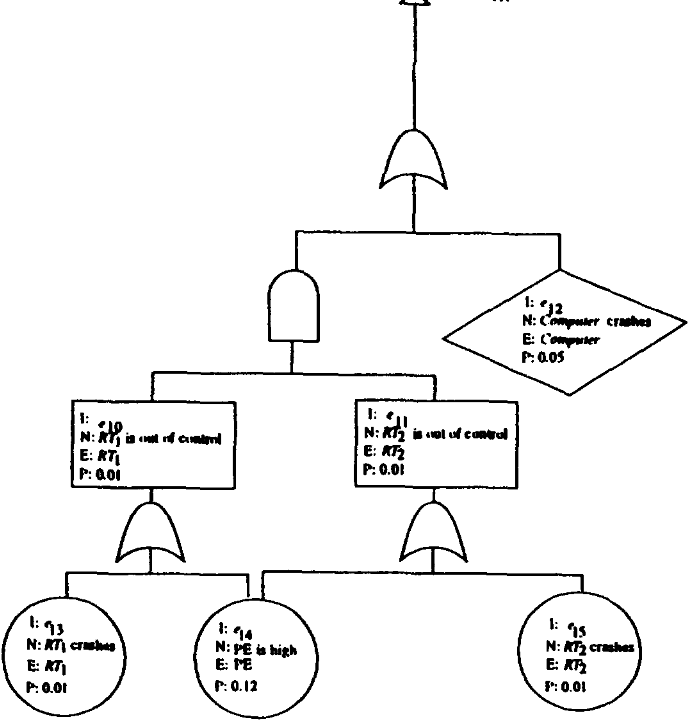

Figure 6. Fault tree about the control of drain valve (*b*).

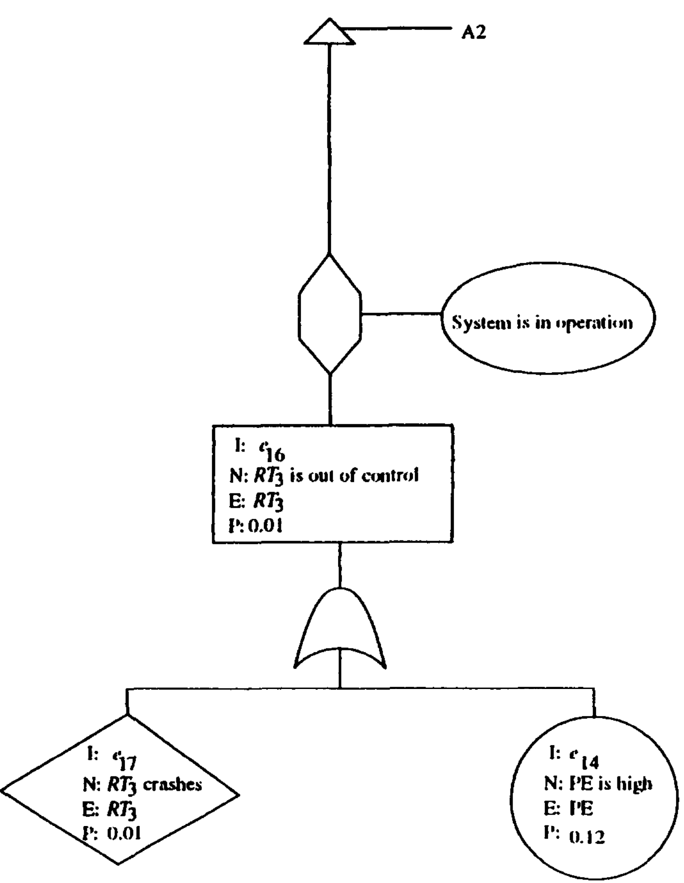

Figure 7. Fault tree about the control of drain valve (*c*).

consistency in order to ensure that the fault tree is valid.

5.1 Internal Consistency of Fault Trees

Internal consistency includes syntactic consistency and semantic consistency. Syntactic consistency requires that a fault tree satisfies the syntactic rules that are described in Section 4.1. Semantic consistency requires that a fault tree must not have any semantic contradictions. They are described in detail below.

5.1.1 Syntactic Consistency. To define it precisely, we need the following notation:

- e^s indicates the shape of the graphical representation of the event e, where s can be one of the shapes: c (circle), r (rectangle), h (house), d (diamond), o (oval). For example, e^c represents that the shape of the event e is a circle.
- $e = e_1$ for the two events e and e_1 means their four components are identical, correspondingly. That is, $e.name = e_1.name$, $e.content = e_1.content$, $e.entities = e_1.entities$ and $e.probability = e_1.probability$.

Definition 5.1 [syntactic consistency]. Let $T = (F_e, F_g, f)$ be a fault tree. T is syntactically consistent if it satisfies the following conditions:

1. $\forall_{e, e_1 \in F_e} \cdot e \neq e_1 \Rightarrow e.name \neq e_1.name$
2. $\exists!_{e_t \in F_e} \forall_{e \in F_e} \cdot e_t \notin f(e)$
3. $\forall_{g \in F_g} \cdot f(g) \neq \{ \}$
4. $\forall_{e \in F_e} \cdot e^d \vee e^c \Rightarrow f(e) = \{ \}$
5. $\forall_{e \in F_e} \cdot e^r \vee e^h \Rightarrow f(e) \neq \{ \}$

The syntactic consistency of a fault tree ensures that five conditions are satisfied. The first is that every event in a fault tree has a unique name. The second is that there exists a unique event which has no parent node (namely, top event). The third is that every gate must have child nodes. We do not require that a child node of a gate must be an event; it can be either an event or a gate because this kind of fault tree has been in use effectively in practice (McDermid and Liu, 1993). The fourth is that every circle and diamond shape of event must not possess any child nodes because they represent basic or undeveloped events. The last one is that every rectangle and house shape of event must have some child nodes as they represent nonbasic events.

Applying this definition to the fault tree presented in Figure 5, Figure 6 and Figure 7, we can see this fault tree satisfies the syntactic consistency.

5.1.2 Semantic Consistency. The purpose of checking the semantic consistency of a fault tree is to ensure that the events and gates are used correctly in the sense of logic. For example, the fault tree in Figure 8 is not semantically sensible because the content (as well as the set of entities and probability) of the event identified by e_1 is the same as that of the event identified by e_2 (which indicates that e_1 and e_2, in fact, represent the same event), but e_1 is caused by e_2 and e_3. In order to define the semantic consistency precisely, we first need to introduce several notions.

Definition 5.2 [path]. Let $T = (F_e, F_g, f)$ be a fault tree, $a, b \in F_e$ be two events. Then a path between a and b is a sequence of events $ax_1x_2 \ldots x_nb$ such that:

$$\left(\exists_{g_1 \in F_g} \cdot g_1 \in f(a) \wedge x_1 \in f(g_1)\right) \wedge \left(\exists_{g_2 \in F_g} \cdot g_2 \in f(x_1) \wedge x_2 \in f(g_2)\right) \wedge \cdots \wedge \left(\exists_{g_n \in F_g} \cdot g_n \in f(x_n) \wedge b \in f(g_n)\right).$$

For example, the following are some paths between the top event e_1 and the events e_5, e_{13}, and e_{17} in the fault tree presented in Figure 5, Figure 6, and Figure 7:

- $e_1e_2e_5$
- $e_1e_3e_6e_{10}e_{13}$
- $e_1e_4e_9e_{16}e_{17}$

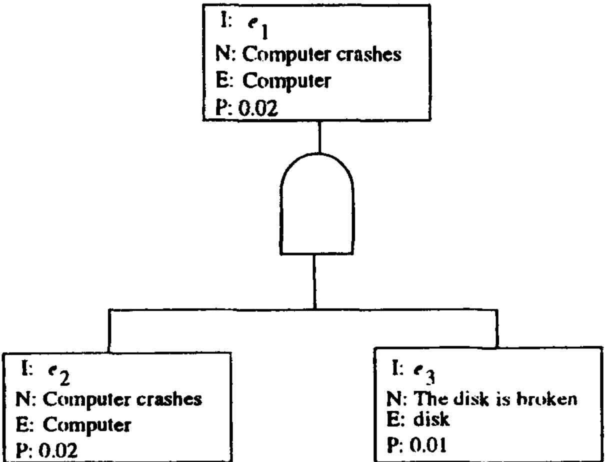

Figure 8. Semantically inconsistent fault tree.

Definition 5.3 [minimal cutset]. Let $T = (F_e, F_g, f)$ be a fault tree. A minimal cutset of T, denoted by M_c^T, is a set of events satisfying the following two conditions:

- $\forall_{e \in M_c^T} \cdot f(e) = \{\ \}$.
- All the events in M_c^T together cause the top event of T.

An algorithm producing all the minimal cutsets of a fault tree is given in (Roberts, 1981) and implemented using the Object-oriented Common Lisp in FTSS. For example, all the minimal cutsets of the fault tree given in Figure 5, Figure 6, and Figure 7 are as follows:

$$\{e_5\}, \{e_{13}, e_{14}\}, \{e_{13}, e_{15}\}, \{e_{14}\}, \{e_{14}, e_{15}\}, \{e_7\}, \{e_8\}, \{e_{17}\}.$$

Definition 5.4 [semantic consistency]. Let $T = (F_e, F_g, f)$ be a fault tree, $M_c^T = \{e_1, e_2, \ldots e_n\}$ be any minimal cutset of T, e_t be the top event of T. For any event e_i $(i = 1..n)$ in M_c^T, if in a path between e_t and e_i, say $e_tx_1x_2 \ldots x_me_i$, there does not exist any two events, say x_1 and x_2, to satisfy the condition:

$$x_1.content = x_2.content,$$

then we say the fault tree T is semantically consistent.

For example, the fault tree given in Figure 5, Figure 6, and Figure 7 satisfies the semantic consistency, but the fault tree in Figure 8 violates the semantic consistency.

5.2 Model Consistency

Model consistency of a fault tree requires that the knowledge concerning the safety of a system, which is expressed by a fault tree, must not contradict the knowledge represented by the system safety model. This is reasonable because the system safety model records the first-hand knowledge of a physical system and is a foundation of constructing fault trees. To define model consistency precisely, we need the notion of a child event.

Definition 5.5 [**child event**]. Let $T = (F_e, F_g, f)$ be a fault tree, a and b be two events in F_e. Then b is called a child event of a if

$$\exists_{g \in F_g} \cdot g \in f(a) \wedge b \in f(g).$$

We use $CHILDREN(e)^T$ to represent the collection of all child events of the event e in the fault tree T. Note that $CHILDREN(e)^T$ is empty if e is a

leaf event. If b is a child event of a, we call a a parent event of b.

Definition 5.6 [model consistency]. Let $T = (F_e, F_g, f)$ be a fault tree, $L_m = (E_n, R, E_v, P)$ be the associated system safety model. The fault tree T satisfies model consistency if

1. $\forall_{e \in F_e} \cdot e \in E_v$.
2. $\forall_{e \in F_e} \cdot e.entities \subseteq E_n$
3. $\forall_{e \in F_e} \cdot e.probability = P(e)$
4. $\forall_{e_1, e_2 \in F_e} \cdot ((e_2 \in CHILDREN(e_1)^T \Rightarrow (\exists_{en_1 \in e_1.entities, en_2 \in e_2.entities} \cdot \exists_{r \in R} \cdot ((en_1, en_2) \in r \vee (en_2, en_1) \in r)))$

Model consistency of a fault tree ensures that four conditions are satisfied by the fault tree. The first is that every event occurring in a fault tree must exist in the system safety model. The reason for this is that we want the system safety model to capture all the possible events concerned with the corresponding physical system and the safety case to reflect the causal relationships between them. The second is that all the entities involved in any event of a fault tree must exist in the system safety model. The third is that the probability of any event in a fault tree must be the same as that of the same event in the system safety model. The fourth condition is that for two events e_1 and e_2 in a fault tree such that e_2 is a child event of e_1, there must exist two entities involved in e_1 and e_2, respectively such that they have some relation existing in the system safety model. This is because every entity in the system safety model represents a component or substance used in the corresponding physical system, and it is usually impossible for two events to have a causal relationship if the entities involved in these two events have no relation at all.

Applying this definition to the fault tree presented in Figure 5, Figure 6, and Figure 7, and the corresponding system safety model described in Section 3.2, we can see that this fault tree satisfies the model consistency.

6. SUPPORT SYSTEM

FTSS (Fault Tree Support System) is a subsystem of SAM (Safety Argument Manager) (Forder et al., 1993). The purpose of this system is to support the Model-Oriented Approach for Safety Analysis introduced previously. In this section, we are not prepared to address the technique issues involved in the implementation of this system but briefly introduce its functionality.

FTSS provides five kinds of services that include *fault-tree drawing*, *internal consistency check*, *model consistency check*, *summary generation*, and *probability management.*

In the service of fault-tree drawing, a menu is provided to list all kinds of events (rectangle, circle, house, diamond, and oval) and gates (AND, OR, EXCLUSIVE-OR, PRIORIT-AND, and INHIBIT) for selection. When a user gives a command to the system for drawing a fault tree, a window will be available for the user to draw a fault tree in it. To draw a fault tree, the user first needs to draw event nodes and gate nodes, and then draw links between them. All of the edit functions including *cut*, *copy*, *paste*, and *move*, etc., are available for users to construct their satisfactory fault trees, and all the drawn fault trees are stored in the database of SAM.

In the service of internal consistency check, syntactic and semantic consistency check are available for applying to users' fault trees. Internal consistency check is implemented based on the rules given in Definition 5.1 and 5.4. When syntactic consistency check is selected, a window called *Report* then appears and the checking result is displayed in it. If there are syntactic errors displayed, each error can be referred to the corresponding graphical nodes (event or gate nodes) to help users locate errors in their fault trees. Semantic consistency check cannot be carried out unless no syntactic error is found. The same *Report* window is also used for displaying semantic errors.

When model consistency check is selected, the FTSS system will search both the fault tree to be checked and the corresponding system safety model stored in the database of SAM. According to the rule given in Definition 5.6, the system will display the checking result in the *Report* window which is either a confirmation message of no error being found, or a list of errors. Model consistency check must be preceded by internal consistency check.

A summary of a fault tree can be generated automatically if the user selects the item 'Summary' from the menu provided by FTSS. A summary of a fault tree T is a simplified fault tree consisting of only the top and leaf events of T as well as one OR gate and some AND gates. A fault tree and its summary are equivalent in the sense that both of them describe the same fact that the same combination of the leaf events cause the same top event. For example, the summary of the fault tree given in Figure 5, Figure 6, and Figure 7 is presented in Figure 9.

In the service of probability management, three concrete services are provided, which are: input probability, change probability, and calculate probability. A user can input and change probabilities of

J. SYSTEMS SOFTWARE 1996; 35:151-164

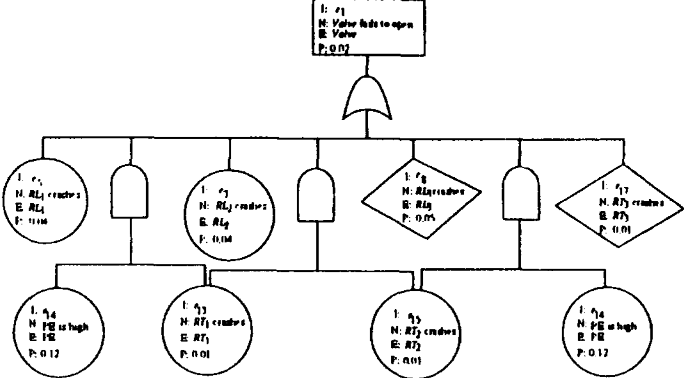

Figure 9. Summary of a fault tree.

all the events in his or her fault trees by means of a dialog mechanism and can order the system to calculate the probability of the top event of the fault trees based on the relevant probability theory (Pate-Cornell, 1984; Villemeur, 1992) and the probabilities of the leaf events of the fault trees.

7. CONCLUSIONS

This paper describes a model-oriented approach for safety analysis using fault trees and its support system. The strategy of this approach requires that safety analysis of a system should start with a systematic study of the physical model of the system concerned and, as a result, construct a system safety model. Then, further safety analysis of the system by constructing fault trees, based on the system safety model, is carried out. A principle of how to construct a system safety model from a physical model is described, and guidelines for constructing fault trees based on a system safety model are presented. In order to ensure that a system safety model and a constructed fault tree are valid, internal consistency of the system safety model, internal consistency, and model consistency of the fault tree, which are well defined in this paper, must be guaranteed. A fault-tree support system (FTSS), which has been prototyped on the ASAM project to support this approach, is described.

There are two limitations with this model-oriented approach for safety analysis and its support system. The first is that it cannot provide a precise way for constructing a system safety model from a physical model, although the structure of a system safety model hints of some guidelines. The difficulty of this limitation is how to know accurately all the possible events and with what probabilities events will happen to the physical system or its components. In fact, this should not only be the responsibility of this model-oriented approach, but also (mainly) the responsibility of safety engineers. Experience suggests that this knowledge can only be obtained by means of repeatedly studying the expertise of the physical system. However, an effective way of doing this needs to be found.

Second, the FTSS is now only a prototype (but a very useful one) and the efficiency needs to be improved. University of York and the Software Engineering Ltd. of York have started another project which aims to develop the present SAM (FTSS is part of SAM) prototype into a final product.

ACKNOWLEDGMENTS

We would like to thank all the colleagues in ASAM for their contributions to this research in various ways. Dr. Chris Higgins of the University of York participated in the implementation of the FTSS and had frequent discussions with us about many relevant issues. Justin Forder, Graham Storrs, Helen Tang, and Mark Howroyd of Logica Cambridge Limited, and Peter Fenelon of the University of York provided great help in the implementation of the FTSS. Finally, we would like to thank British SERC and DTI as well as Hiroshima City University of Japan for their financial support to this project and for improving this paper, respectively.

REFERENCES

Alesso, H. P., and Benson, H. J., Fault Tree and Reliability Relationships for Analyzing non Coherent Two-State Systems, *Nuclear Engineering and Design*, 309–20 (1980).

Forder, J., Higgins, C., McDermid, J. A., and Storrs, G., SAM—A Tool to Support the Construction, Review and Evolution of Safety Arguments, in *Proceedings of Safety-critical Systems Symposium 1993*, Bristol, 9th–11th February, 1993.

Health and Safety Executive, PES—Programmable Electronic Systems in Safety Related Applications, Crown copyright, pp. 80–99, 1987.

Lambert, H. E., Systems safety analysis and fault tree analysis, UCID-16238, 31, 1973.

Leveson, N. G., and Harvey, P. R., Analyzing Software Safety, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, 569–579 (September 1983).

Leveson, N. G., and Harvey, P. R., Software Fault Tree Analysis, *The Journal of Systems and Software*, 173-181 (1983).

McDermid, J. A., and Liu, S., A Case Study Using SAM—Safety Analysis of PES, submitted to the *Journal of High Integrity Systems*, 1993.

Pate-Cornell, M. E., Fault Trees vs. Event Trees Reliability Analysis, *Risk Analysis*, Vol. 4, No. 3, 177-186 (1984).

Perkusich, A., De Figueired, J. C. A., and Chang, S. K., Embedding Fault-Tolerant Properties in Design of Complex Software Systems, *Journal of Systems and Software*, 23-37 (April 1994).

Roberts, N. H., Vesely, W. E., Haas, V. F., and Goldberg, F. F., Fault tree handbook, Technical Report by Systems and Reliability Research Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission, Washington, D.C. 20555, January 1981.

Villemeur, A., *Reliability, Availability, Maintainability and Safety Assessment*, John Wiley and Sons Ltd, West Sussex, England, 1992, pp. 149-195.

Zhuang, W. J., and Xie, M., Design and Analysis of Some Fault-Tolerance Configurations Based on A Multipath Principle, *Journal of Systems and Software*, 101-108 (April 1994).