Fault Tree Analysis and Formal Methods for Requirements Engineering

by

Jianwen XIANG

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Kokichi FUTATSUGI

School of Information Science
Japan Advanced Institute of Science and Technology

Sept. 2005

Abstract

This thesis is concerned with fault tree analysis (FTA) and formal methods for more efficient and precise requirements engineering, i.e., how to develop the fault trees in a formal correct way, and how to combine FTA and formal methods in a consistent way to assist system analysis, design, and verification.

In this thesis, focused on the incorrectness problem of traditional fault trees, we first propose a new formal fault tree construction model based on monotonicity of temporal logic, and demonstrate how to formally model, specify, and verify a system based on the analyzing results of fault trees with observational transition system (OTS) of CafeOBJ (a wide spectrum specification language based on multiple logical foundations). And as a complement of theorem proving technique of CafeOBJ, we also discuss how to model-check OTSs with Maude (a sibling language of CafeOBJ). A new formal fault tree analysis is then proposed based on the OTS model in order to make the combination of safety analysis (FTA) and requirements analysis (formal system specification and verification with OTS/CafeOBJ) more consistent. Finally, we present some further discussion and analysis of formal fault tree semantics and demonstrate how to transform the results of fault tree analysis into formal system specifications by using the common signature and framework of OTS.

The technical contributions of our work are as follows.

- We carry out our study on a unified platform — OTS/CafeOBJ, which makes the combination of FTA and formal system specification and verification more smooth and consistent.
  - We identify decomposition of fault events as a core issue that guarantees the correctness of the fault trees, i.e., the sub events must formally result in their top event through the given logic gate. As for a solution, we propose a formal fault tree construction model based on temporal logic to guarantee the correctness of the fault trees.
  - We propose an approach to derive concrete requirements (safety assumptions and commitments) from FTA so as to guild and assist system design and verification;
  - We demonstrate how to write fault tree specification and realize automatic calculation of minimal cut sets of fault trees with term rewriting system (TRS) of CafeOBJ;
  - We demonstrate how to formally model, specify, and verify OTSs with CafeOBJ based on the analyzing results of FTA.

  - Most importantly, we further propose a novel formal fault tree analysis by introducing the basic concepts of OTS. The point is that, by using a common framework of OTS, it is possible to use the results of fault tree analysis directly, when specifying and verifying the system with OTS/CafeOBJ. Therefore, we build a common semantic model for safety analysis and software requirements specifications.

- In addition, as a complement of theorem proving technique provided by CafeOBJ, we also demonstrate how to model-check OTSs with Maude, which makes our method more complete.

Key Words: *Fault Tree Analysis, Formal Methods, Requirements Engineering, Theorem Proving, Model Checking, CafeOBJ, Maude*

Acknowledgments

The dream of every graduate student is to find his advisor a true mentor. Prof. Kokichi FUTATSUGI of JAIST (Japan Advanced Institute of Science and Technology) has been that, and much more. He has not only introduced me to the research in Formal Methods and Requirements Engineering, but also guided me during these years with constant encouragement and precious advice as a strict while generous and kindhearted teacher, allowing me to grow in my enthusiasm and confidence. This thesis would not have been possible without his invaluable support and guidance.

I wish to express my sincere gratitude to my sub-advisor Associate Professor Rene Vestergaard of JAIST for his insightful suggestions and comments while always in an interesting and vivid way, especially on the ropes of art of doing research and technical writing and presentation. Without his help and encouragement, this research would not finish successfully or just have come out to be insipid and indigested.

Associate Professor Kazuhiro OGATA of JAIST and NEC Software Hokuriku Ltd. (Japan), throughout my days at JAIST, has been a constant source of inspiration and support; whenever I have any question and obstacle, I can always get prompt advice and detailed comments from him, which has always gone beyond what I could ever hope. He has my deepest gratitude for his personal warmth and help through my study.

I am grateful to Professor Yoshiteru NAKAMORI of JAIST, who was my minor theme advisor on the study of Knowledge Management, Professor Yanxiang HE of Wuhan University of China, who was my former advisor when I studied at Wuhan University before coming to JAIST, Professor Koichiro OCHIMIZU of JAIST, Assistant Dr. Masaki NAKAMURA and Dr. Noriki AMANO of JAIST, and Judith A. STEEH, the head editor of JAIST for their kind help that they have always been ready to provide. I have greatly benefited from their valuable suggestions and comments.

The days would not have been the same without the friendship of and stimulating exchanges with Kazuki MUNAKATA (my tutor and first Japanese friend), Weiqiang KONG, Masahiro NAKANO, Jittisak SENACHAK, Dr. Takahiro SEINO, Daigo YAMAGISHI, Yoshiyuki ASABA, Bochao LIU, Yu JIN, Dr. Tieju MA, Chuan HE, Biao HUANG, Chen YU, to name only a few; their kindness and help have greatly smoothed and enriched both my study and life in JAIST. My days in JAIST would just have been boring, tedious, dull and lonely without sharing experiences in technical discussions, travelling, PC games, cooking, sports, movies, and fun-making with them. It is really a precious treasure with a lot of fun that I would like to cherish for ever. I would also like to devote my sincere thanks and appreciation to all of them, and my colleagues.

My parents have been an endless source of love, happiness, and support. They have devoted their whole love and unreserved support to my study, even when it meant long

years of separation. I also can't image my life without my elder sister who always supports me and takes care of my old parents for me. Their support and encouragement have made it possible for me to go a long way.

Finally, my parents-in-law have been another source of warmth, support, and encouragement. Their kindness and understanding have been the best gift (together with my wife) that I could have ever had. My wife, Jing TIAN, has been long years of my love, my joy, my strength, and my happiness, and for ever. This dissertation is dedicated to her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is concerned with fault tree analysis (FTA) and formal methods for more efficient and precise requirements engineering, i.e., how to develop the fault trees in a instructive formal way with respect to complex system analyses, and how to combine FTA and formal methods in a consistent way as for system analysis, design, and verification.

As we know, nowadays the high-flying development of information and software technologies helps us develop more complex and more powerful systems to improve our lives and to realize our dreams, such as exploring Mars; but on the other hand, more serious accidents have occurred than before because of the same reason, that is to say, even tiny flaw or mistake may cause fatal accidents, such as the space shuttle Columbia Tragedy in 2003. It is a crucial issue for both engineers and designers to check the correctness, consistency, and accuracy between system implementation and initial requirements, so as to guarantee that the important system safety properties are really preserved using formal methods.

Within the field of requirements engineering and system safety analysis, a number of useful techniques have been developed. Examples of them are fault tree analysis (FTA) [VGRH81], failure mode and effects analysis (FMEA) [Rei83], hazard and operability analysis (HAZOP) [FMNP85], and state machine hazard analysis (SMHA) [Lev87]. These methods have different coverage and validity, for instance, FTA is primarily a means for analyzing causes of hazards, while FMEA was developed by reliability engineers to permit them to predict equipment reliability, and it is a form of reliability analysis that focus on successful functions rather than hazards and risks. So in requirements engineering and system safety analysis, we often need to use several methods at the same time during the life of the project. But in this thesis, we focus on FTA and formal methods with CafeOBJ [FNT00, DF98] (a wide spectrum specification language based on multiple logical foundations), and concern ourselves with the *correctness* of the fault tree with respect to complex system analyses, and the combination of FTA and CafeOBJ as for more efficient requirements engineering. To this end, we proposed two formal fault tree construction models based on temporal logic and OTS, respectively, discussed how to write the formal specifications of the fault trees and realize automated logic deduction and calculation of minimal cut sets of the fault trees with term rewriting system (TRS) of CafeOBJ, and introduced how to formally model, specify, and verify the system as well as its important safety properties based on the analyzing results of FTA with observational transition system (OTS) of CafeOBJ.

In the following sections of this chapter, we will first briefly introduce the history and development of FTA, and then focus on the problems of traditional FTA, a survey of current formal works on FTA will be presented. Based on the above discussion and the problems of current formal FTAs, we will carry out our research motivations and targets. Finally we will present the organization of this thesis.

1.1 History and Development of Fault Tree Analysis

Fault Tree Analysis (FTA) is originally developed in 1961 by H. A. Watson at Bell Telephone Laboratories to evaluate the Minuteman Launch Control System for an unauthorized missile launch [WL61]. Boolean logic method had been used in FTA to analyze the relationships between events/hazards. Then, the engineers of Boeing Company developed the procedure further and became its foremost proponents. After that, it has been supported by a rich body of research and widely used in the aerospace, electronics, and nuclear industries. One of the important handbooks of FTA, "Fault Tree Handbook" was written by U.S. Nuclear Regulatory Commission (NRC) in 1981 to serve as a text for the system safety and reliability course [VGRH81]. It could be looked as the first combination of material on fault tree construction and evaluation.

FTA 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 (Boolean) logic symbols. An important concept of FTA is *Cut Sets*, that is, a set of basic events enough to cause the system to fail . The goal of the analysis is to find the *Minimal Cut Set*, which represents the basic events that will cause the system to fail and which cannot be reduced in number - that is, a cut set that does not contain another cut set .

FTA can be used both in *qualitative* and *quantitative* analysis. The purpose of qualitative analysis is to reduce the minimal cut sets; while the quantitative analysis of fault trees uses the minimal cut sets to calculate the probability of occurrence of the top event from the probability of occurrence of the basic events [HK00, Sho90] . Or in reverse, given the system reliability (the negation of the probability of occurrence of the fault event) and development budget/cost, calculate and maximize the reliability of each basic event [XFH03], which is a kind of software reliability allocation problem.

Although FTA was originally developed to analyze hardware hazards in general, it also provides a useful qualitative design aid for software systems as well [HH86] . Similar to the analysis of hardware systems in which the basic events are usually associated with physical component failures, the basic events of a software system represent software *modules* which might produce an incorrect result or accept an invalid input, or a basic event can represent the incorrect setting of an initialization parameter by a user or a buffer overflow [Lyu95]. It can help designers understand the potential failure modes of a software system, and provides a logical structure for the placement of exception tests.

On a lower level, FTA might also be applied to the *code/program statement* analysis to aid in safety validation. For instance, McIntee has shown how it can be applied to

assembly language programs [McI83], while Taylor [Tay82] and Leveson et al. [LH83] have demonstrated the technique on software written in higher-level languages, and some more recent works can be in [MJC+99] and [Liu00] with respect to Ada95 and SOFL (a formal specification language) respectively. These methods are called Software Fault Tree Analysis (SFTA) . The goal of SFTA is either to find paths through the code from particular inputs to the hazardous outputs, or prove that such paths do not exist using designed templates for program statements. In fact, SFTA is a graphical application of axiomatic verification where the postconditions describe the hazardous conditions rather than the correctness conditions [Lev91] .

In addition to the detailed analysis of single-version critical software, fault tree models are useful for analyzing the failure modes of *fault-tolerant* software systems. Here, the basic events are considered as fault activation scenarios that will result in system-level failures. Three qualitative fault tree models for distributed recovery block (DRB) [KW89], N-version programming (NVP) [AC77, CA78, Avi85], and N self-checking programming (NSCP) [LABK90, DB93] system respectively, together with a quantitative assessment of the probability of failure using a common set of assumptions and parameter values are discussed in [Lyu95, pages 627–645]. A more complete analysis of these three systems (DRB, NVP, and NSCP) that considers the dynamic reconfiguration of the system configuration in response to hard permanent faults by introducing and combining Markov model with FTA can be found in [DL95] .

1.2 Problems of Traditional FTA

The advantages associated with the use of fault trees are the graphical and mathematical foundations, which give rise to good qualitative and quantitative solution methods, and thus it provides a common framework for comparative analysis in requirements engineering [Lyu95]. However, like any other model or method, FTA also has several *limitations* as follows.

One criticism is that the fault trees are often constructed after the system has been well analyzed and designed, since to develop the correct and useful fault trees usually requires detailed knowledge of the system. For example, in an electrical system analysis, without a precise operation diagram of the system, it is impossible to develop the fault tree with respect to a specific hazard or undesired event. However, from the point of view of requirements engineering, it is more desired that FTA can be applied in the early design stages of the system life cycle, and thus the results of FTA can be used to assist the system design. This problem often happens in hardware system analyses using FTA, and some people think that it may be better to spend time ensuring that the design criteria have been incorporated rather than building fault trees in case we have already well defined all the operations and criteria of the system [Ham80]. One use for FTA in the design stage is that at the system level to trace system hazards to individual sub-systems or modules, such as FTA for software system analyses as we introduced in the last section.

Another criticism is that FTA is a simplified representation of a generally very complex process: Its relative simplicity can be deceptive [Lee80]. For example, the technique is

particularly suited to discrete events, such as a valve opening or closing, but time- and rate-dependent events, such as changes in critical process variables, degrees of failure (partial failure), and dynamic behavior are not so easily represented [ea83]. This is because FTA was originally developed for hardware system analyses, and its logic foundation is simple propositional logic. To solve this problem, a dynamic fault tree has been defined, which used a Markov chain for solution [DBB92].

Problems may also occur when time spans or chronological ordering of events should be specified in some realtime systems, since the fault tree is generally a snapshot of the state of the system at one point in time [Lev95]. As a treatment to this problem, some special gates such as INHIBIT and PRIORITY-AND have been proposed [VGRH81]. However, some researchers think that this complicates the evaluation of the tree, and somewhat negates one important advantage of FTA – the ease with which the trees can be read and understood and thus reviewed by domain experts and used by designers. And they think that if chronology is important, using a model and analysis technique that involves backward or forward search may be more appropriate than forcefitting this into a hierarchical, top-down modeling technique [Lev95].

Traditional FTA does not represent the transitions between states, and it is difficult to handle complex system states which consist of event combinations rather than component state itself [Div87]. This is because in traditional FTA, each node (event) was defined as a single system or component fault state (which seems enough for hardware system analyses), while in complex system analyses, an undesired event often consists of a combination of several *normal* states/events, it is difficult to decompose such event without the concept of state transition.

An important and related issue is that generally speaking, FTA does not provide precise formal semantics to verify the correctness of its descriptions. This situation stems from that FTA was originally developed as a graphical and informal method for the engineers to analyze hardware systems. The advantage of informal and graphical method is obvious as its disadvantage, that is, on one hand, it can quickly provide a good first-hand communication platform for the domain experts and designers, but on the other hand, without formal semantics supporting, it is quite difficult to persuade people to accept the correctness and completeness of the result. It seems to be a common problem as to the safety analysis techniques mentioned in the beginning of this chapter: As Leveson has said, "Perhaps the most important fact to keep in mind is that very little validation of any of these techniques has been done, and so all results should be treated with appropriate skepticism" [Lev95].

Further considering this problem, it results from the difference and complexity between hardware and software systems, that is, unlike hardware systems in which it is quite easy to directly decompose a system fault event into some component failures, a fault event in a software system are more difficult to deal with in case the fault event consists of complex logic formulas. If we still first decompose the fault event and construct the fault tree in an informal way (or by intuition), just like the hardware fault tree analysis, a subsequent stand-alone formal verification and refinement of the fault tree is unavoidable. It is a kind of patch work that seems easy to begin with, but ends up being much harder and more complicated. Even so, some new problems and unnecessary side conditions will be introduced as we have proved in this thesis.

Additional criticisms of FTA relate to its quantitative aspects. For example, common-cause failures may cause problems and lead to orders-of-magnitude errors in the calculated failure probability [McC81]. However, since the quantitative analysis of fault tree is not the main concern of this thesis, we did not discuss this issue in detail here.

1.3 Related Works on Formal Fault Tree Analysis

Focused on the limitations of traditional fault tree analysis mentioned in above section, recently several formal fault tree models have been developed, which can be classified into the following categories.

System model for safety and requirements analyses with FTA

As we discussed in the last section, FTA was often criticized because the fault trees were usually developed after the system design state, and thus the effect of FTA is doubted to some extent. To this end, Kirsten M. Hansen and Anders P. Ravn proposed a system model for fault tree analysis and program development [HR98]. The model was formalized in a real-time, interval logic, based on a conventional dynamic systems model with states evolving over time. Fault trees were interpreted as temporal formulas after giving different semantics to the constructs such as leaves, nodes and logic gates, and it was shown how such formulas can be used to derive safety requirements for software components.

In [HR98], Hansen et al. have shown how fault tree analysis and system design could be made to interact much effectively by a railway interlocking system example. The point is that, through the formulation of software safety requirements, FTA can provide a feedback to the system design, which also inspires our work in this thesis.

Dynamic fault tree models

The first dynamic fault tree model was proposed J. B. Dugan et al., which was served to analyze fault tolerant computer systems [DBB92]. In this model, fault tree has capability to model complex dynamic relationships between failures, such as failure sequences, functional dependencies, in which a Markov chain is used for solution. A software package for the analysis of dynamic fault tree models, called DIFtree was also developed by Dugan's group [DVG97].

Related work can also be found in [MDCS98], which further extended DIFtree analysis capability to model several different distribution of time to failure, including fixed probabilities (no time component), exponential (constant hazard rate), Weibull (time varying hazard rate), and log normal. It is an extension of dynamic fault tree modeling technique for the reliability analysis of computer based systems.

The formal semantics of dynamic fault tree was defined in [CSD00], in which Z specification language and denotational semantics techniques were used to represent the formal specification of dynamic fault trees.

Temporal semantics of fault trees

Traditional FTA was based on Boolean logic to represent the fault trees (logic gates and events), assuming that there is no time-delay between the inputs (sub-events) and output (event) of a logic gate. Such an assumption rarely causes problems in hardware system analyses, in which the flow usually is virtually instantaneous (such as an electric circuit). Another assumption is that once a fault occurs, then it will exist forever, i,e., considering all systems as nonrepairable. In other words, once the sub-events occur, then the output event will *always* hold.

However, in complex system analyses where the flow is not instantaneous, such assumptions may cause troubles or misunderstanding as we discussed in the last section. To this end, G. Bruns and S. Anderson proposed two other temporal semantics for gates of fault trees [BA93]. One took causality between the inputs and output of a gate to be only sufficient, not necessary or immediate, i.e., once the gate condition is satisfied, the gate output must *eventually* occur, using the temporal logic operator **Eventually** (which means some time in the future); The other treated causality as only necessary, that is, if the gate output happens, then the inputs must have happened some time in the past, using the temporal operator **Once** [1].

Another work can be found in [HR98] as we introduced above – system model for safety and requirements analyses with FTA. In [HR98], the fault trees were interpreted as temporal formulas after giving different semantics to the constructs, i.e., events and logic gates. However, since the main intention of Hansen's work was to show how such formulas can be used to derive safety requirements for software components, it did not consider the time-delay between the inputs and output of the logic gates. For example, a AND-gate was defined as in a form of $A \stackrel{def}{=} B_1 \wedge \ldots \wedge B_n$, where A and $B_i (i = 1, \ldots, n)$ denoted the output and input event, respectively. The temporal logic was mainly introduced to interpreted the events and the safety requirements of the trees [2].

In [STR02], G. Schellhorn, A. Thums and W. Reif further defined a subtle formal semantics of fault trees, which allowed cause-consequence relations among events in addition to Boolean decomposition. In this model, formal completeness and correctness conditions (corresponding to necessary and sufficient conditions in traditional FTA) for logic gates are given, using Interval Temporal Logic with continuous semantics. However, their method is still the same as traditional formal fault tree analyses, that is, first building the informal fault tree by intuition, then verifying the correctness and completeness conditions of the fault tree according to defined formal semantics.

Combination of FTA and other formal methods

In addition to providing different formal semantics and specifications of fault trees, researchers also tried to combine FTA with other formal methods, such as model checking and Petri-Nets, so as to formally verify the results of FTA and improve requirements analyses. Examples of them are as follows.

[1] In [BA93], it used the temporal operator **Prev** rather than **Once**, however, this is easy to cause confusing with the standard temporal logic operator **Previous**, which means in the previous state [MP95]. Therefore, here we revised the notation based on the explanation of [BA93].

[2] A special case is the PRIORITY AND-gate, in which the input events must occur in a left to right order so as to cause output event, here, temporal operator **Eventually** was introduced to represent the order of the input events as used in [HR98]

- Andreas Schäfer presented a semantics for fault tree analysis in the real-time interval logic Duration Calculus with Liveness (DCL) [sch03]. And the properties of the fault tree was checked automatically by model-checking of Phase Automata [Tap01, DT03]. In addition, an profit of the combination of FTA and model checking was shown in the case study of a single track line segment system, that is, instead of model-checking a complex top-event directly, FTA can be used as a decomposition method to find the manageable and basic events to check, which is more efficiently.

- Guy Helmer, and Johnny Wong et al. proposed an integration model of software fault tree analysis and Colored Petri Nets (CPNs) [HWS+00]. In this model, software fault trees (SFTs) was augmented with nodes in terms of trust, temporal, and contextual relationships to describe intrusions of an agent-based intrusion detection system (IDS), while CPNs for intrusion detection were built using CPN templates created from the augmented SFTs.

- Axel Lankenau and Oliver Meyer proposed a general verification approach for reactive systems, which was based on a CSP specification of a fault tree that observes the behaviour of the target system [LM99]. A template for the modeling of fault tree leaves and nodes was given, and it was demonstrated by a mobile robotics system. This work also demonstrated the benefit of the integration of informal requirements, the semi-formal fault tree technique and the formal verification using a model checking as for more efficient requirements engineering.

- Another related work is Axel van Lamsweerde's obstacle analysis of goal-oriented requirements engineering [vLL00, vL01], which uses a temporal logic variant of the regression procedure found in [vL91] as well as the tree structure to formally deduce the possible obstacles with respect to a specific goal. It can be viewed as a kind of goal-anchored obstacle tree analysis rather than standard fault tree analysis. The point is that it effectively avoids the incorrectness problem of traditional fault tree analysis thanks to the formal regression procedure [vL91], which inspires our work for the formal fault tree construction.

In short, the above formal FTAs provided valuable and useful experiences as for FTA and formal methods in requirements engineering, and some of them inspired our work in this thesis, such as the idea of deriving software requirements from safety analyses with FTA [HR98], the temporal logic semantics of fault trees [BA93, STR02], and the combination of FTA and formal verification techniques [sch03, HWS+00, LM99]. However, these works mainly focused on providing more precise formal semantics for fault tree constructors or different fault tree models, they seldom considered how to formally construct the fault tree in a deductive manner. The common method is to develop the formal model and the fault trees as separate documents. That is to say, building up the fault trees is driven by intuition, while the events and sub-events of a gate are formalized afterward with respect to the formal model [RST00]. This approach is effective for quickly constructing a fault tree, but the informal construction creates problems later, when verifying its correctness. And that is why a stand-alone formal verification and refinement of the fault tree is unavoidable in these methods.

As we discussed in the last section, It is a kind of patch work that seems easy to begin with, but ends up being much harder and more complicated. To this end, we bring out our research motivation and targets in the next section.

1.4 Research Motivation and Targets

As discussed above, FTA is widely used in requirements analyses as a safety analysis technique thanks to its qualitative and quantitative methods as well as rich graphical supports. However, with respect to complex system analyses, traditional FTA should be improved and augmented, such as introducing formal temporal semantics for the fault trees as some researchers have done. A neglected problem is that, most of the formal FTAs focused on providing more precise formal semantics for the fault tree constructors, such as nodes, leaves, and logic gates, while few of them paid attention to the formal fault tree construction.

In other words, the common sense is still regarding FTA as a semi-formal method, and thus separating fault tree construction and verification as separated processes. The advantage of this approach is obvious, that is, it can quickly construct the fault trees as a communication platform for the experts and designers. But it makes the formal verification of the fault trees much more difficult, just like a kind of patch work.

For example, in [STR02], Schellhorn et al. noticed that sometimes the correctness condition was unprovable, in that the sub-events could not formally result in the top event through a logic gate. They found that such gates typically represent design decisions, in which all states described by the sub-events were considered already faulty regardless of side conditions. This fact has led traditional formalization attempts of fault trees either to require only the completeness condition [HRS94] (i.e. neglect the correctness condition in this case), or make the side conditions explicit in the fault tree by introducing an INHIBIT-gate [STR02]. But in that case, finding and confirming such side conditions may become another problem, especially if the gate is in the bottom layer of a large and complex informal fault tree. Even so, another problem may appear, that is, how can we ensure such side conditions are really *necessary*?

In addition, as we know, a common roadblock to the adoption of formal methods in system analysis is the difficult and tedious work of logic proving and deduction. Providing a lightweight formal method to deduce/develop fault trees as well as their formal specifications with the support of an automated logic deduction mechanism is another important task we want to attack. The significance here is not only to provide relief for engineers from difficult and tedious logic proving and deduction; more importantly, the formal specification of the fault tree is also a very important formal document per se for the subsequent formal system modeling, specifying and verifying. From this point of view, an executable formal language is preferred than other notational formal languages, such as Z. And this is the reason why we choose CafeOBJ [FS95, FN97, DF98], an executable algebraic specification language which is a modern successor of OBJ [JWM+00, FGJM85], in our study.

Another important issue is how, after finding the basic fault events that will result in system failure, to ensure that such safety properties (the negations of the basic fault events) will be preserved in the system. In other words, can we formally prove that the basic fault events (related to design failures) will never occur in our revised system? This formal verification work is quite important since it can not only greatly and efficiently enhance system safety, but it also helps us pay more attention to other aspects or fault events of the system (the basic events related to physical failures and human errors); that

is, we can have a more definite object in view to allocate our resources more efficiently and reasonably in terms of different system safety properties. This issue has been referred partially in some previous formal FTAs, such as in [HR98], Hansen proposed the concepts of safety assumption and commitment to distinguish the basic events related to physical failures (human errors) and design failures, respectively, but it did not further discuss the formal verification of the design failures; and in [sch03, LM99], the combination of FTA and model-checking was proposed to verify the properties of fault trees, but they did not distinguish different fault events as for the whole safety requirements analyses.

Therefore, in the work described in this thesis, we concentrate our attention on the following topics (targets).

- Firstly, based on our previous work [XFH04b, XFH04a, XFH04c] and further comprehensive analysis of the problems caused by traditional methods, we further improve and complete our formal fault tree construction model based on monotonicity of temporal logic [MP95] as well as the idea of Lamsweerde's obstacle analysis [vLL00]. In our model, the correctness of the fault tree is guaranteed by the construction process itself, along with the sub-events are deduced recursively and inductively by finding hidden domain rules at the design stage, giving the designers the opportunity to revise their system design in a timely fashion.

- Secondly, we demonstrate how CafeOBJ [FS95, FN97, DF98], can be used as an integrated powerful tool with the formal fault tree model for requirements engineering: it is not only useful to write the formal fault tree specifications in which we encapsulate some automatic logic deduction functions into the system built-in modules , but is also powerful to formally model, specify, and verify a system as well as its important safety properties discovered in the fault tree model.

- Thirdly, as we know, with respect to finite system analyses, model-checking technique is easier and more practical than theorem proving technique. To this end, we demonstrate how to model-check OTSs with Maude as a complement of theorem proving technique of CafeOBJ.

- Last but not least, to make the combination of FTA and OTS/CafeOBJ more consistently and smoothly, we further propose a novel formal fault tree analysis model by introducing the basic concepts of OTS, which has the ability to represent transitions between states and deal with complex events that consists of a combination of events rather than component failures (one of the limitations of traditional FTA as we discussed in Section 1.2). An important motivation of this study is that, as we know, a common framework is important whenever engineers and designers from multiple disciplines need to work together, therefore to propose a formal fault tree analysis model by using the common framework of OTS is really desired, which can efficiently smooth the transformation between FTA and formal system specification and verification with OTS/CafeOBJ. From this point of view, it is more than an alternative (to the formal fault tree construction model based on temporal logic we proposed) but a significant improvement for the engineers (especially those who are not familiar with temporal logic).

1.5 Thesis Organization

The rest of this thesis is organized as follows.

- Chapter 2 provides some preliminaries on fault tree analysis, temporal logic, and CafeOBJ. It constitutes the rationale of this thesis and provides a quick reference for readers who are not familiar with these fields.

- Chapter 3 first deeply analyzes the origin and harm of traditional construction methods, i.e., the incorrectness problem, based on a radio-based crossing control system example. Then, it elaborates our formal fault tree construction method based on monotonicity of temporal logic in detail. Finally, it presents how to derive concrete safety assumption and commitment from FTA for the subsequent system formal specification and verification.

- Chapter 4 demonstrates how to use CafeOBJ as an integrated tool to implement the automated logical deduction and formal specification of fault tree construction, along with the formal modeling, specification, and verification of the system as well as its important safety properties. Model-checking OTSs with Maude is then discussed as a complement of theorem proving technique of CafeOBJ. A discussion and analysis of the combination of FTA and OTS/CafeOBJ(Maude) is presented at the end of this chapter.

- Chapter 5 further proposes a novel formal fault tree analysis model by introducing the basic concepts of OTS (called FTA/OTS), in which the formal semantics of events of fault trees are redefined, as well as the definition of transition rules and corresponding decomposition patterns of sub-events are provided.

- Chapter 6 presents some further discussions and analyses of formal fault tree semantics, and demonstrates how to transform the results of FTA/OTS into formal system specifications with OTS/CafeOBJ.

- Chapter 7 summarizes this thesis, and discusses some related and future works.

- Appendixes include the complete CafeOBJ and Maude codes as well as the experimental results discussed in Chapter 4 for reference.

Chapter 2

Preliminaries

2.1 Background of Fault Tree Analysis

Fault tree analysis [VGRH81] is a deductive safety analysis technique which is applied during the design phase. The technique was first developed in the 1960s to facilitate analysis of the Minuteman missile system [WL61] and has been supported by a rich body of research since its inception.

A fault tree analysis can be simply understood as an top-down analytical technique, whereby an undesired state of the system is specified (usually a state that is critical from a safety standpoint), and the system is then analyzed in the context of its environment and operation to find all credible ways in which the undesired *top event* can occur [VGRH81]. The top event is resolved into its constituent causes, connected by logic *gates* that denote the type of relationship of the input causes required for the output top event, such as AND, OR, and EXCLUSIVE OR. The causes are called *fault events* that are associated with component hardware or software failures, human errors, or any other related events which can result in the undesired top event. The fault events then should be further resolved until *basic events* are identified, which represent basic initiating faults requiring no further development/resolution. The extent of the analysis, i.e., which components are considered basic, depends on the abstraction level chosen.

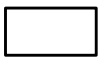
It should be noted that a fault tree is *not* a model of *all* possible system failures or all possible causes for system failure. A fault tree is developed from a top event which corresponds to a specific system failure mode, and the fault tree includes only the faults contributing to this top event. Therefore, in system analysis, we need to develop and analyze various fault trees with respect to different system failure modes.

Moreover, It is also important to point out that the faults of a fault tree are *not* exhaustive — they cover only the faults that are assessed by analysts, in other words, *the completeness of the faults depends on the understanding and analysis of the analysts.*

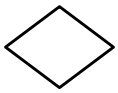
2.1.1 Fault Tree Symbols

A typical fault tree is composed of a number of standardized symbols, which is shown in Figure 2.1. There are several variations and extensions, but in this thesis we limit ourselves to the following symbols.

BASIC EVENT: A basic initiating fault requiring no further development.



INTERMEDIATE EVENT: An event that results from a combination of events through a logic gate.



UNDEVELOPED EVENT: An event which is not further developed either because it is of insufficient consequence or because information is unavailable.

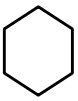
CONDITIONING EVENT: Specific conditions that apply to any logic gates(usually used with PRIORITY AND and INHIBIT gate).

AND-gate: The output fault in the top node occurs only when all the input children faults occur.

OR-gate: The output fault in the top node occurs only when one or more the input children faults occur.

PRIORITY AND-gate: The output fault in the top node occurs only when the input children faults occur in a left to right order

EXCLUSIVE OR-gate: The output fault in the top node occurs only when exactly one of the input children faults occurs.



INHIBIT-gate: The Output fault in the top node only occurs if the input children fault occurs in the presence of an enabling condition, i.e. a CONDITIONAL EVENT.

Figure 2.1: Fault tree symbols

2.1.2 Fault Tree Construction Fundamentals

In Section 2.1.1, we introduced the symbols which are used to construct the fault tree. In this section we discuss some fundamental concepts and rules for proper fault tree construction.

Fault Occurrence vs. Fault Existence

In the discussion of the fault tree gates in Section 2.1.1, we have spoken of the occurrence of faults. A fault may be repairable or not, depending on the nature of the system. If no repair, a fault that occurs will continue to exist. It is important to distinguish between the occurrence of a fault and its existence in a repairable system, but this distinction is of importance only in fault tree quantification [VGRH81]. And from the viewpoint of fault

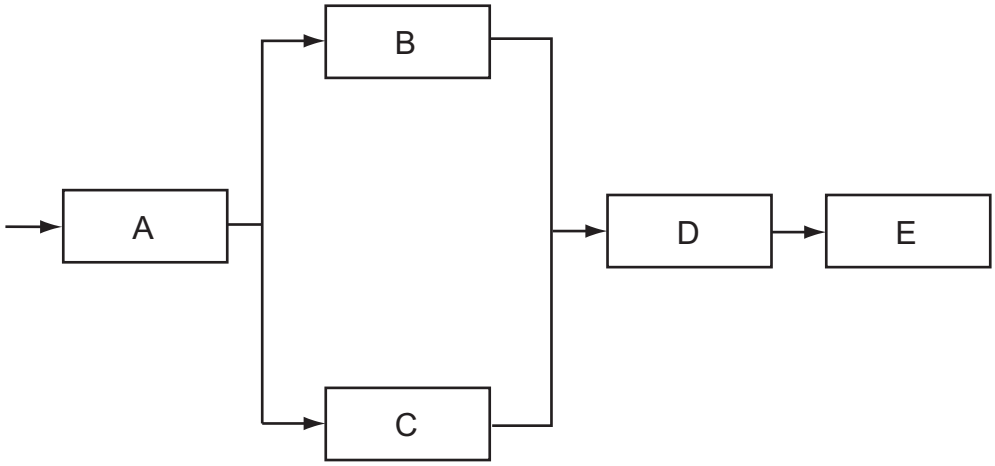


Figure 2.2: System illustrating "Immediate Cause" concept

tree construction, we need concern ourselves only with the phenomenon of occurrence, i.e., supposing all systems are nonrepairable.

Component Fault Categories

In hardware system analysis, it is useful for fault tree analyst to classify component faults into three categories: *primary*, *secondary* and *command* [VGRH81].

- A primary fault is any fault of a component that occurs in an environment for which the component is qualified; e.g., a pressure tank, designed to withstand pressures up to and including a pressure p_0, ruptures at some pressure $p \leq p_0$ because of a defective weld.
- A Secondary fault is any fault of a component that occurs in an environment for which it has *not* been qualified, i.e., the component fails in a situation which exceeds the conditions for which it has been designed; e.g., in the above example of the pressure tank, ruptures under a pressure $p > p_0$.
- In contrast to that the primary and secondary faults are generally component failures, a command fault is a fault with the *proper operation* of a component but at the *wrong time* or in the *wrong place*.

The Concept of "Immediate Cause"

In fault tree construction, an important concept of the fault resolution is to find the *immediate*, *necessary*, and *sufficient* causes for the occurrence of the fault event. It should be noted that these are not the *basic* causes of the event but the *immediate* causes or *immediate* mechanisms for the event [VGRH81].

The "immediate cause" concept is sometimes called the "Think Small" Rule because of the methodical, one-step-at-a-time approach, and the motivation is to ensure that no fault event in the sequence is overlooked.

As an example to further illustrate the "immediate cause" concept, consider the simple system in Figure 2.2, which is taken from [VGRH81].

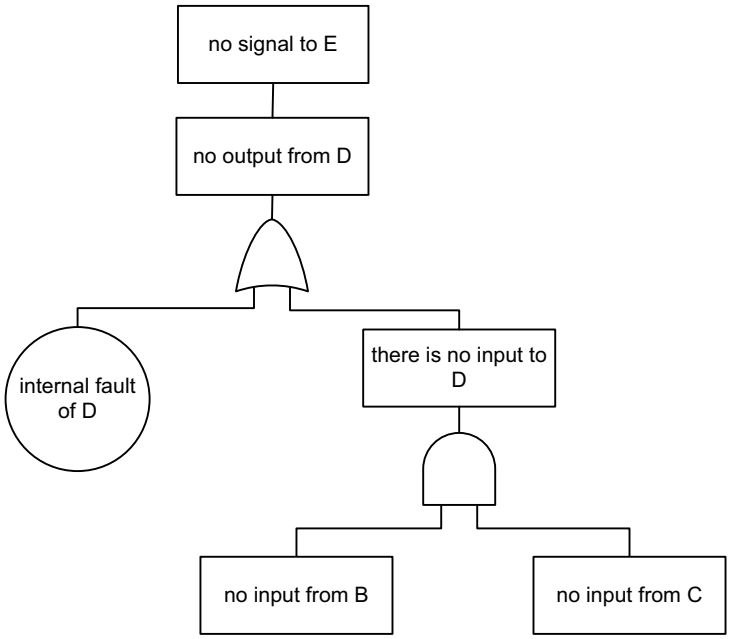


Figure 2.3: Fault tree for system illustrating "Immediate Cause" concept

The system is supposed to operate in the following way: a signal to A triggers an output from A which provides inputs to B and C. B and C then pass a signal to D which finally passes a signal to E. A, B, C, and D can be considered as dynamic subsystems. Furthermore, a signal from either B or C or both of them to D will trigger its output to E. Therefore, the system has redundancy in this portion. The system can be interpreted quite generally, such as an electrical system in which the subsystems are analog modules, or a piping system in which A, B, C, and D are valves, and so on.

If we Choose the top event as "no signal to E", and assume that there is no transmitting failures in the system. The immediate cause of the event, "no signal to E", is then defined as "no output from D". It should be noted that we can not directly list the event, "no input to D" as the immediate cause of "no signal to E", which violates the "one-step-at-a-time" approach, and some fault events will be overlooked.

Then, we focus on the sub-top event "no output from D", and its immediate causes are the *union* of the following two events:

(1) "There is an input to D but no output from D."

(2) "There is no input to D."

Notice here if we directly listed the event, "no input to D" as the immediate cause of "no signal to E" as mentioned above, then the event (1) is missed in this case.

Suppose our limit of resolution is the subsystem level, then event (1) (which can be rephrased as "D fails to perform its proper function due to some fault internal to D", or in short, "internal failures in D") is not analyzed further and constitutes a basic event of our fault tree. Analyzing event (2) and its immediate causes recursively in an analogous way, we derive a fault tree shown in Figure 2.3.

Basic Rules for Fault Tree Construction

In the beginning, the construction of fault trees was thought of as an art more than a science. But after a period of years practices, people have realized that successful fault

trees were all drawn in accordance with the following basic rules [VGRH81].

Rule 2.1 (Ground Rule I) *Write the statements that are entered in the event boxes as faults; state* precisely *what the fault is and when it occurs.*

Rule 2.2 (Ground Rule II) *If the answer to the question, "Can this fault consist of a component failure?" is "Yes," classify the event as a "state-of-component fault." If the answer is "No," classify the event as a "state-of-system fault."*

If the fault event is classified as "state-of-component," add an OR-gate below the event and look for primary, secondary and command modes. If the fault event is classified as 'state-of-system," look for the minimal necessary and sufficient *immediate* cause or causes.

Rule 2.3 (No Miracles) *If the normal functioning of a component propagates a fault sequence, then it is assumed that the component functions normally.*

Rule 2.4 (Complete-the-Gate) *All inputs to a particular gate should be* completely *defined before further analysis of any one of them is undertaken.*

The rule 2.4 states that the fault tree should be developed in levels, and each level must be completely analyzed before any consideration of its lower level. It helps the analyst develop the fault tree in a *methodical* way.

Rule 2.5 (No Gate-to-Gate) *Gate inputs should be properly defined fault events,and gates should* not *be directly connected to other gates.*

The "gate-to-gate" shortcuts may lead to confusion and may demonstrate that the analyst has an incomplete understanding of the system. Such attempt of shortcutting must be avoid during the analysis and construction process.

The above concepts and rules are useful and instructive for the proper fault tree construction, especially in hardware fault tree analysis. However, with respect to software and integrated system analysis, the situation is more complex and thus some new rules should be introduced, which we will discuss in Chapter 3.

2.1.3 Minimal Cut Sets and Minimal Path Sets

Analysis of a fault tree begins with an enumeration of the *minimal cut sets*, i.e., the smallest combinations of component failures which, if they all occur, will cause the top event to occur. A *cut set* is a combination (intersection) of primary events sufficient for the top event. A *minimal cut set* is a cut set with no redundant elements, that is, if any failures is removed from a minimal cut set then it is ceases to be a cut set [VGRH81, Lyu95].

In contrast, a complemental concept of FTA is *minimal path set*, which is a smallest combination (interaction) of basic events whose non-occurrence assures the non-occurrence of the top event [VGRH81, pages VII-15 – VII-20]. This concept was proposed from the point of view of reliability when we concerned with the prevention of the top event. The way to get the minimal path sets is easy and similar to the way of getting minimal cut sets: first *complement* the original fault tree (including all the events and

logic gates) by applying De Morgan's theorem of Boolean Algebra Laws and get a complemented tree, called the dual of the original fault tree or *dual fault tree* , then use the same method (see Algorithm 2.1) to calculate the minimal path sets. In case the minimal cut sets has already been calculated as a disjunctive normal form (DNF), just negating the DNF and transforming it into another DNF can also get the minimal cut sets.

By introducing the concepts of minimal cut sets and minimal path sets, it is convenient to discuss the (global) *correctness* and *completeness* of the fault trees. A fault tree is correct provided that for any minimal cut set, if all the primary events of the minimal cut set occur, then the top event must occur also. In contrast, a fault tree is complete provided that for any minimal path set, the non-occurrence of all the primary events of the minimal path set assures the non-occurrence of the top event. In this thesis, we also discuss (local) correctness condition and completeness condition of gate, which are used for defining the semantics of gates. The correctness condition of a gate states that the input events can logically imply the output event. And in reverse, the completeness condition of a gate states that the output event can logically imply the input events, i.e., all the causes (input events) has been listed. Therefore, the global correctness and completeness of fault tree depend on that all the local correctness and completeness conditions of gates of the fault tree can be proved, respectively.

Algorithm 2.1 (A top-down algorithm for determining the minimal cut sets)

1. Starts at the top event of the fault tree and constructs the set of cut sets by considering the gates at its lower level.
   - If a gate below the top event is an *OR*-gate, then split the input events into different cut sets, one containing each input to the *OR*-gate.
   - If it is a *AND*-gate, then list all the input events into one cut set.
2. With respect to each intermediate event of the cut sets, regard it as a top event and repeat the step 1 until the set of basic events is reached.
3. Finally, to get the minimum cut sets from the cut sets, we have to remove the redundancies according to the following rules.
   - If two cut sets are the same, then delete one;
   - If one cut set is the subset of another, the latter can be removed since it is not a minimal cut set;
   - If a cut set contains the same basic event more than once, then the redundant entries can be delete.

Actually, the algorithm for determining the minimal cut sets can be understood as a process of Boolean substitution and absorbtion if we use Boolean equations to represent the fault tree. First translate the top event to its equivalent Boolean equation, and then perform the Boolean substitution for the intermediate events repeatedly until the basic events are reached, finally use two Boolean laws, the distributive law and the law of absorption, to remove the redundancies [VGRH81, pages VII-15–VII-19].

For better understanding, Figure 2.4 shows an example fault tree whose cut set generation is shown in Figure 2.5. The undesired top event T occurs when both intermediate

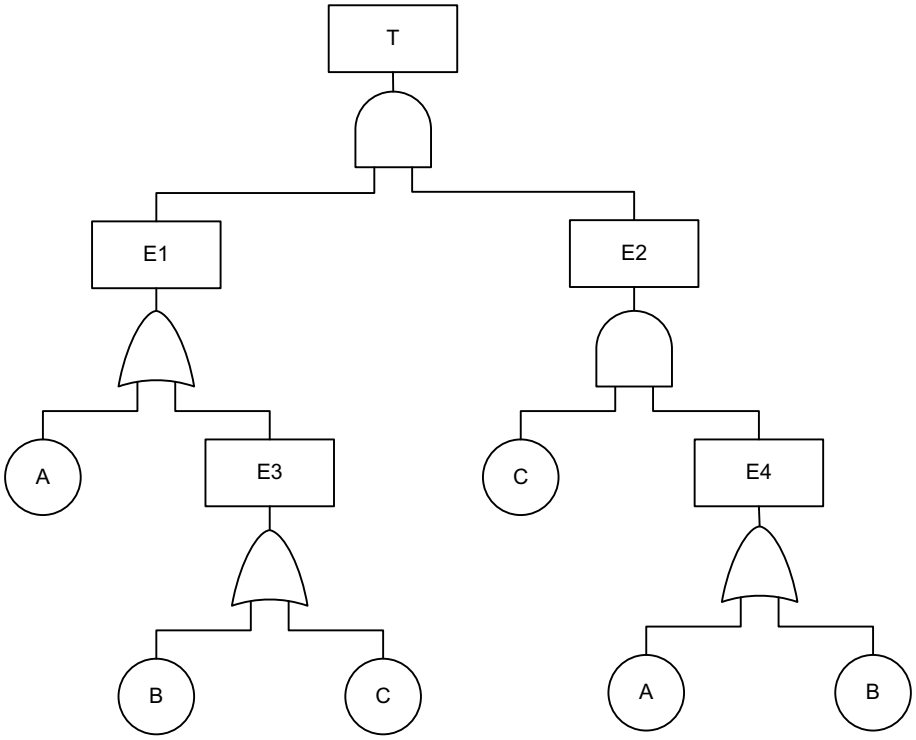


Figure 2.4: Example fault tree

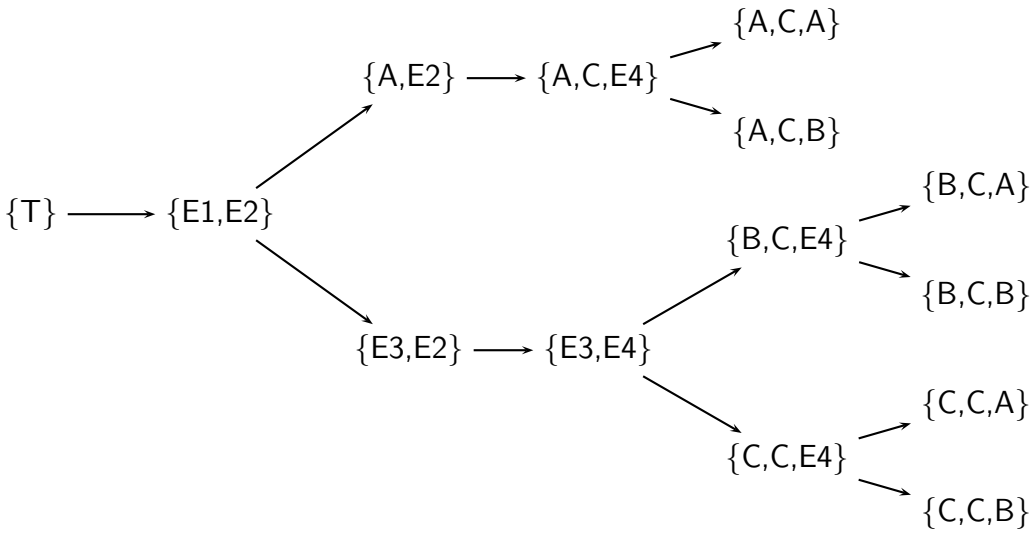


Figure 2.5: Cut set generation for a fault tree

events E1 and E2 occur, which are connected by a OR-gate and AND-gate respectively with other intermediate events or basic events. There are four intermediate events and three basic events in the fault tree, labeled E1 to E4, and A, B, and C respectively.

To get the minimal cut sets of Figure Figure 2.4, let's first analyze the fault tree from the top even T. Because T is resolved into its causes E1 and E2 through a AND-gate, it can be replaced with one set, {E1, E2}. Then we focus on E1, since it resolved into a basic event A and a intermediate event E3 through a OR-gate, we split the set of {E1, E2} into two sets, {A, E2} and {E3, E2}. Redo this process until all the basic events are reached, we get all the six cut sets of the fault tree, i.e. {A, C, B}, {A, C, B}, {B, C, A}, {B, C, B}, {C, C, A}, and {C, C, B} (the generation process of the cut sets is shown in Figure 2.5). Since there are some redundant entries in the resulting cut sets, we can use the absorption rules of Algorithm 2.1 to delete the redundant entries and cut sets, and finally we get two minimal cut sets of the fault tree, {A, C} and {B, C}.

2.1.4 Quantitative Analysis of Fault Tree

Once the minimal cut sets are obtained, probability analysis can be performed if quantitative results are desired. The quantitative results obtained from the evaluation include: (a) absolute probabilities, (b) quantitative importance of components and minimal cut sets, and (c) sensitivity and relative probability evaluations [VGRH81]. The quantitative evaluations usually are performed in a sequential manner, first determining the component failure probabilities/rates, then the minimal cut set probabilities, and finally the system, i.e., top event probability. If the failure rates are treated as random variables, then random variable propagation techniques can be used to estimate the variabilities in system results which result from the failure rate variation. But in this thesis, we limit ourselves in the qualitative analysis and the formal construction model of the fault trees, so here we just briefly introduce a simple algorithm to calculate the failure probability as follows.

Given the component failure probabilities, we can calculate the top event failure probability. The top event probability can be determined by the set of the minimal cut sets: the set of minimal cut sets denotes all the factors that will cause the system failure, thus the system failure probability is the probability of the union of the minimal cut sets.

$$P\left\{\bigcup_i C_i\right\} \tag{2.1}$$

where C_i is the minimal cut set. Since there always exist some intersections (same basic events) between the minimal cut sets, so the system failure probability is not equal to the sum of minimal cut set probabilities. If we simply add the minimal cut set probabilities together, the system failure probability would be overvalued because we count some basic events for several times.

To solve this problem, there are some methods to evaluate the Equation 2.1, the simplest way is including-excluding method, which is the general principle to calculate the probability of the union of two events:

$$P\{A \cup B\} = P\{A\} + P\{B\} - P\{A \cap B\} \tag{2.2}$$

So we get the refined algorithm below [Lyu95].

$$P\left\{\bigcup_{i=1}^{n} C_i\right\} = \sum_{i=1}^{n} P\{C_i\} - \sum_{i<j} P\{C_i \cup C_j\} + \sum_{i<j<k} P\{C_i \cap C_j \cap C_k\} \mp \cdots \pm P\left\{\bigcap_{i=1}^{n} C_i\right\} \quad (2.3)$$

In addition, with respect to software reliability allocation, a fault tree allocation model has been developed [XFH03]. It is a kind of inverse calculation of system failure probability, i.e., dealing with the setting of reliability goals for individual components such that a specified system reliability goal is met and the component goals are well balanced among themselves [MIO87].

2.2 Temporal Logic Foundations

The formal semantics of our formal fault tree construction model is based on Temporal Logic [MP92] with real-time restrictions [Koy92]. In this thesis, we will use the following classical temporal operators (shown in Figure 2.6).

Symbol	Name	Explanation
◯	*Next* Operator	in the next state
⊖	*Previous* Operator	in the previous state
◇	*Eventually* Operator	some time in the future
⧫	*Once* Operator	some time in the past
□	*Henceforth* Operator	always in the future
⊟	*Has-always-been* Operator	has always been in the past

Figure 2.6: Temporal logic operator symbols

For each of the operators and sub-formulas allowed in a temporal formula, we introduce a definition of its interpretation in a given model. This definition is based on the notion of a formula p holding at a time position j, $j \geq 0$, in a sequence σ, denoted by

$$(\sigma, j) \models p$$

Where $j \in T$, and T denotes a linear temporal structure assumed to be discrete in this thesis for sake of simplicity. The semantics of the above temporal operators are then defined as usual [MP92] as shown in Figure 2.7.

We will also use the standard logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), $\Rightarrow$ (strongly implies), $\Leftrightarrow$ (strongly equivalent), with

$(\sigma, j) \models \bigcirc p$ iff $(\sigma, j+1) \models p$
$(\sigma, j) \models \ominus p$ iff $(j > 0)$ and $(\sigma, j-1) \models p$
$(\sigma, j) \models \Diamond p$ iff $(\sigma, k) \models p$ for some $k, \geq j$
$(\sigma, j) \models \Diamond\!\!\!\!-\, p$ iff $(\sigma, k) \models p$ for some $k, 0 \leq k \leq j$
$(\sigma, j) \models \Box p$ iff $(\sigma, k) \models p$ for all $k, k \geq j$
$(\sigma, j) \models \boxminus p$ iff $(\sigma, k) \models p$ for all $k, 0 \leq k \leq j$

Figure 2.7: Semantics of temporal logic operators

$$p \Rightarrow q \text{ iff } \Box\,(p \rightarrow q)$$
$$p \Leftrightarrow q \text{ iff } \Box\,(p \leftrightarrow q)$$

Note thus that there is an implicit outer $\Box$-operator in every strong implication.

Moreover, there are some important *distribution properties* of the nonimmediate operators, i.e., excluding the next/previous operators. Some of these operators have a universal character while others have an existential character. For example, the unary operators $\Box$ and $\boxminus$ are universal, while the unary operators $\Diamond$ and $\Diamond\!\!\!\!-$ are existential. The universal operators are distributed over conjunction and universal quantifications, while the existential operators are distributed over disjunction and existential quantifications.

This is expressed by the following properties, stated for the future operators $\Box$ and $\Diamond$.

DP1. $\Box\,(p \wedge q) \Leftrightarrow (\Box\, p) \wedge (\Box\, q)$

DP2. $\Box\,(\forall u : p) \Leftrightarrow \forall u : \Box\, p$

DP3. $\Diamond\,(p \vee q) \Leftrightarrow (\Diamond\, p) \vee (\Diamond\, q)$

DP4. $\Diamond\,(\exists u : p) \Leftrightarrow \exists u : \Diamond\, p$

Some often used entailments are:

E1. $\Box\, p \Rightarrow p$

E2. $\boxminus\, p \Rightarrow p$

E3. $p \Rightarrow \Diamond\!\!\!\!-\, p$

E4. $p \Rightarrow \Diamond\, p$

And in requirements engineering we often need to introduce real-time restrictions. Bounded versions of the above temporal operators are therefore introduced, in the style advocated by [Koy92], such as:

$\Diamond_{\leq d}$ (some time in future within deadline d)
$\Box_{\leq d}$ (always in future up to deadline d)
$\boxminus_{\geq d}$ (always in the past over deadline d)

To define such operators, the temporal structure T is enriched with a metric domain D and a temporal distance function $dist : T \times T \rightarrow D$, which has all desired properties

of a metrics [Koy92]. A frequent choice is as follows.

T: the set of naturals
D: $\{d$: there exists a natural n such that $d = n \times u\ \}$, where u denotes some chosen time unit
$dist(j,k)$: $|k - j| \times u$

Multiple units can be used (e.g., second, day, week); they are implicitly converted into some smallest unit. The $\bigcirc$-operator (next-operator) then yields the nearest subsequent time position according to this smallest unit.

The semantics of the real-time operators is then defined accordingly, e.g.,

$(\sigma, j) \models \Diamond_{\leq d} p$ iff $(\sigma, k) \models p$ for some $k \geq j$ with $dist(j,k) \leq d$
$(\sigma, j) \models \Box_{\leq d} p$ iff $(\sigma, k) \models p$ for all $k \geq j$ such that $dist(j,k) \leq d$

An important property of temporal logic that we will use in our formal fault tree construction model, i.e., *monotonicity* [MP92, pages 202–204], is as follows.

Monotonicity

Let $\varphi(u)$ be a formula scheme with one or more occurrences of the sentence symbol u.

We define an occurrence of u to be *positive* (*of positive polarity*) in φ if it does not occur in a subformula of the form $p \leftrightarrow q$ and it is embedded in an even (explicit or implicit) number of negations. Note that an occurrence of u in p in the context $p \rightarrow q$ counts as an (implicit) negation, because $p \rightarrow q$ is equivalent to $(\neg p) \vee q$. Similarly, an occurrence of u in φ is defined to be *negative* (*of negative polarity*) if it does not occur in a subformula of the form $p \leftrightarrow q$ and it is embedded in an odd number of negations.

There are two general monotonicity properties for the case that all occurrences of u in $\varphi(u)$ have *uniform polarity*, i.e., are all positive or all negative.

Claim 2.1 (positive polarity) *If all occurrences of u in $\varphi(u)$ are positive, then*

$$(p \Rightarrow q) \rightarrow (\varphi(p) \Rightarrow \varphi(q))$$

is valid.

Claim 2.2 (negative polarity) *If all occurrences of u in $\varphi(u)$ are negative, then*

$$(p \Rightarrow q) \rightarrow (\varphi(q) \Rightarrow \varphi(p))$$

is valid.

And for completeness, we also list substitutivity as a property, restricted to the computations for which $p \Leftrightarrow q$.

Claim 2.3 (substitution) *For an arbitrary formula $\varphi(u)$,*

$$(p \Leftrightarrow q) \rightarrow (\varphi(p) \Leftrightarrow \varphi(q))$$

is valid.

2.3 CafeOBJ and Observational Transition System

In this section we present a brief overview of the CafeOBJ [FS95, FN97, DF98] and the basic concepts of Observational Transition System (OTS) as well as its description in CafeOBJ. The intention is to provide readers who are not familiar with CafeOBJ and parallel computational models with a perceptual understanding and useful references. The OTS model will be used to formally model, specify, and verify the distributed and parallel systems with CafeOBJ in the subsequent chapters, combined with our formal fault tree construction model.

2.3.1 Introduction of CafeOBJ

CafeOBJ is an executable, industrial strength, algebraic specification language which is a modern successor of OBJ [JWM+00, FGJM85], incorporating several new algebraic specification paradigms. It is perhaps the most famous algebraic language. Its definition is given in [DF98]. CafeOBJ is intended to be mainly used for system specification, the formal verification of specifications, rapid prototyping, programming, etc. Following is a brief overview of its underlying logic as well as some most important features.

Underlying Logic of CafeOBJ

CafeOBJ is a specification language based on three-way extensions to many-sorted equational logic: the underlying logic is order-sorted, not just many-sorted; it admits unidirectional transitions, as well as equations; it also accommodates hidden sorts, on top of ordinary, visible sorts. A subset of CafeOBJ is executable, where the operational semantics is given by a conditional order-sorted term rewriting system. These theoretical bases are indispensable to employ CafeOBJ properly. Fortunately, there is an ample literature on these subjects, and we are able to refer the readers to, e.g., [EM85, MG86] (for basics of algebraic specifications), [GM89, GD94a] (for order-sorted logic) , [GM97] (for hidden sorts) , [JR97] (for coinduction) , [Mes91] (for rewriting logic) , [GB92] (for institutions) , and [Klo87, DJ90] (for term rewriting systems) , as primers. The logical aspects of CafeOBJ are explained in detail in [DF96] and [DF98].

For a very brief introduction, we just highlight a couple of features of CafeOBJ. CafeOBJ is an offspring of the family of algebraic specification techniques. A *specification* is a text, usually of formal syntax. It denotes an algebraic system constructed out of *sorts* (or data types) and sorted (or typed) operators. The system is characterized by the *axioms* in the specification. An axiom was traditionally a plain equation ("essentially algebraic"), but is now construed much more broadly. For example, CafeOBJ accommodates conditional equations, directed transitions, and (limited) use of disequality.

The underlying logic of CafeOBJ is as follows.

Order-Sorted Logic [GM89]

A sort may be a subset of another sort. For example, natural numbers may be embedded into rational numbers. This embedding makes valid the assertion that 3 equals 6/2. It also realizes "operator inheritance", in the sense that an operator declared on rational numbers are automatically declared on natural numbers.

Moreover, the subsort relation offers a simple way to define partial operations and exception handling.

Rewriting Logic [Mes91]
In addition to equality, which is subject to the law of symmetry, you may use transition relations, which are directed in one way only. State transitions are naturally formalized by those relations. In particular, transition relations are useful to represent concurrency and/or indeterminacy.

Hidden Sorts [GM97]
There are two kinds of equivalence. One is a minimal equivalence, which identifies terms (elements) iff they are the same under the given equational theory. The other equivalence, employed for so-called hidden sorts, is behavioral: two terms are equivalent iff they behave identically under the given set of observations.

Important features of CafeOBJ

Now we present a brief overview of the main features of CafeOBJ, all of them reflected in the above logical semantics. These should be understood in their combination rather than as separated features. Combining some of these features (sometimes all of them!) result in new specification/programming paradigms that are often more powerful than the simple sum of the paradigms corresponding to the individual features.

Equational Specification and Programming
This is inherited from OBJ [JWM$^+$00, FGJM85] and constitutes the basis of the language, with the other features somehow built on top of it. As with OBJ, CafeOBJ is executable (by term rewriting), which provides an elegant, declarative way of functional programming, often referred as algebraic programming. As with OBJ, CafeOBJ permits the equational specification modulo to include several equational theories such as associativity, commutativity, identity, idempotence, and combinations among all these. This feature is reflected at the execution level by the term rewriting modulo of such equational theories.

Behavioural Specification
Behavioural specification [GD94b, GM97, DF00] provides a novel generalization of ordinary algebraic specification. Behavioural specification characterises how objects (and systems) behave, not how they are implemented. This new form of abstraction can be very powerful in the specification and verification of software systems since it naturally embeds other useful paradigms such as concurrency, object-orientation, constraints, nondeterminism, etc. (see [GM97] for details). Behavioural abstraction is achieved by using specifications with hidden sorts and a behavioural concept of satisfaction based on the idea of indistinguishability of states that are observationally the same, which also generalizes process algebra and transition systems (see [GM97]). CafeOBJ directly supports behavioural specification and its proof theory through special language constructs, such as

- Hidden sorts (for states of systems)
- Behavioral operations (for direct "actions" and "observations" on states of systems)

- Behavioral coherence declarations for (non-behavioral) operations (which may be either derived (indirect) "observations" or "constructors" on states of systems), and
- Behavioral axioms (stating behavioral satisfaction)

Behavioural specification is reflected at the execution level by the concept of behavioural rewriting [DF00, DF98] which refines ordinary rewriting with a condition ensuring the correctness of the use of behavioural equations in the proving strict equalities.

Rewriting Logic Specification

Rewriting logic specification in CafeOBJ is based on a simplified version of Meseguer's rewriting logic (abbreviated RWL) [Mes92] specification framework for concurrent systems which gives a non-trivial extension of traditional algebraic specification towards concurrency. RWL incorporates many different models of concurrency in a natural, simple, and elegant way, thus giving CafeOBJ a wide range of applications. From a methodological perspective, CafeOBJ develops the use of RWL transitions for specifying and verifying the properties of declarative encoding of algorithms (see [DF98]) as well as for specifying and verifying transition systems.

Module System

The basic building blocks of CafeOBJ are modules. The principles of the CafeOBJ module system are inherited from OBJ which builds on ideas first realized in the language Clear, most notably its institutions [BG80]. CafeOBJ module system features include:

- Several kinds of imports
- Sharing for multiple imports
- Parameterized programming allowing
  - Multiple parameters
  - Views for parameter instantiation
  - Integration of CafeOBJ specification with executable code in a lower level language
- Module expressions

However, the theory supporting the CafeOBJ module system represents an updating of the original Clear/OBJ concepts to the more sophisticated situation of multi-paradigm systems involving theory morphisms across institution embeddings [Dia98], and the concrete design of the language revise the OBJ view on importation modes and parameters [DF98].

Powerful Type System

CafeOBJ has a type system that allows subtypes based on order sorted algebra (abbreviated OSA) [GM89, GD94a] . This provides a mathematically rigorous form of runtime type checking and error handling, giving CafeOBJ a syntactic flexibility comparable to that of untyped languages, while preserving all the advantages of strong typing. The order sorted feature of CafeOBJ not only greatly increases

expressivity, but it might also provide a rigorous framework for multiple data representations and automatic coercions among them [GD94a]. CafeOBJ does not directly do partial operations but rather handles them by using error sorts and a sort membership predicate in the style of membership equational logic (abbreviated MEL) [Mes97] . The semantics of specifications with partial operations is given by MEL.

The above introduction to CafeOBJ features may be a little hard to understand because of some mathematic and technical terminology, interested readers can find further information in the references or by accessing http://www.ldl.jaist.ac.jp/cafeobj/. This CafeOBJ home page provides pointers to various aspects of the CafeOBJ research, including theoretical works, methodological works, the system, manuals, examples, and various applications.

2.3.2 Basic Computational Models: Transition Systems

UNITY [CM98] is a parallel computational model, and a specification and programming logic. It provides a proof system based on the logic that is an extension of Floyd-Hoare Logic [Flo67, Hoa69] to parallel programs, and is also influenced by temporal logic [MP92]. UNITY has a minimum notational machinery to represent the parallel computational model.

UNITY models are reformulated in the same manner as the definition of fair transition systems [MP92, MP95], which are called observational transition systems, or OTS's. We can use an OTS to model a parallel and distributed system. An OTS $\mathcal{S} = \langle \mathcal{V}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{V}$: A set of typed variables. Each variable has its own type. The variables (or their possible values) form the state space Σ of $\mathcal{S}$, and a state of $\mathcal{S}$ is a point, or an element of Σ.

- $\mathcal{I}$: The initial condition. This condition specifies the initial values of the variables. Since some variables may not be specified by $\mathcal{I}$, $\mathcal{S}$ may have more than one initial state.

- $\mathcal{T}$: A set of transition rules. Each transition rule $\tau \in \Sigma$ is a function $\tau : \Sigma \rightarrow \Sigma$ mapping each state $s \in \Sigma$ into a successor state $\tau(s) \in \Sigma$. Transition rules are generally defined together with conditions on which the transition rules are *effectively* executed, namely that their execution can change states of $\mathcal{S}$. If the condition of a transition rule is false in a state of $\mathcal{S}$, namely that the transition rule is *not effective* in the state, its execution does not change the state.

As defined above, an OTS is deterministic with respect to each transition rule. That is, given a state of $s \in \Sigma$ and a transition rule $\tau \in \Sigma$, exactly one successor state $\tau(s) \in \Sigma$ is determined. The reason is that, our purpose is not only to describe an OTS as a model in CafeOBJ, but also to verify that the system has some safety properties based on the CafeOBJ document with the help of CafeOBJ system. Hence, from this viewpoint, a deterministic transition system is more appropriate to CafeOBJ aided verification than a nondeterministic one, because CafeOBJ system does not support nondeterministic execution or term rewriting. Moreover, if nondetermination of a transition rule is needed,

we can easily achieve this by dividing the transition rule into multiple ones each of which is deterministic.

Similar to UNITY, an execution starts from one initial state and goes on forever; in each step of execution a transition rule is chosen nondeterministically and executed. Nondeterministic selection is constrained by the same *fairness rule* above. Given an OTS, a set of infinite sequences of states are obtained from execution, constrained by the fairness rule of OTS. Such an infinite sequence of states is called a computation of the OTS. More specifically, a computation of an OTS $\mathcal{S}$ is an infinite sequence $s_0, s_1, \cdots$ of states satisfying:

- Initiation: For each $v \in \mathcal{V}$, v satisfies $\mathcal{I}$ in s_0.
- Consecution: For each $i \in \{0, 1, \ldots\}$, $s_{i+1} = \tau(s_i)$ for some $\tau \in \mathcal{T}$.
- Fairness: For each $\tau \in \mathcal{T}$, there exist an infinite number of indexes $i \in \{0, 1, \ldots\}$ such that $s_{i+1} = \tau(s_i)$.

A state of an OTS is called reachable if it appears in a computation of the OTS.

The concept effectiveness is similar to *enabledness* used in description of transition systems in temporal logic such as TLA [Lam94] or in a precondition-effect style such as I/O automata [Lyn96].

2.3.3 Description of OTS in CafeOBJ

As introduced in Section 2.3.1, CafeOBJ is mainly based on two logical foundations: *initial* and *hidden* algebra [GM97]. Initial algebra is used to specify abstract data types such as integers; while hidden algebra is used to specify objects in object-orientation. Corresponding to it, there are two kinds of sorts (types in programming languages) in CafeOBJ: *visible* and *hidden* sorts. A visible sort represents an abstract data type, and a hidden sort represents the state space of an object. Two kinds of operations are used for hidden sorts: *action* and *observation*, corresponding to so-called methods in object-orientation. An action can change a state of an object. It takes a state of an object and zero or more data, and returns another (possibly the same) state of the object. An observation can be used to observe the value of a data component in an object. It takes a state of an object and zero or more data, and returns the value of a data component in the object; it does not change the state of the object.

Declarations of visible sorts are enclosed with '`[`' and '`]`', and those of hidden sorts with '`*[`' and '`]*`'. Declarations of observations and actions starts with '`bop`' or '`bops`', and those of other operations with '`op`' or '`ops`'. After bop or op (or bops or ops), an operator is written (or more than one operator is written), followed by '`:`' and a sequence of sorts (i.e. sorts of the operators' arguments), and ended with '`->`' and one sort (i.e. sort of the operators' results). Definitions of equations start with '`eq`', and those of conditional ones with '`ceq`'. After eq, two expressions, or terms connected by '`=`' are written, ended with a full stop '`.`'. After ceq, two terms connected by '`=`' are written, followed by if and a term denoting a condition, and ended with a full stop '`.`'.

Since objects can be regarded as transition systems, an OTS can be naturally described in CafeOBJ. The state space of an OTS is denoted by a hidden sort. For better understanding, we give a simple bank account system as a running example as follows.

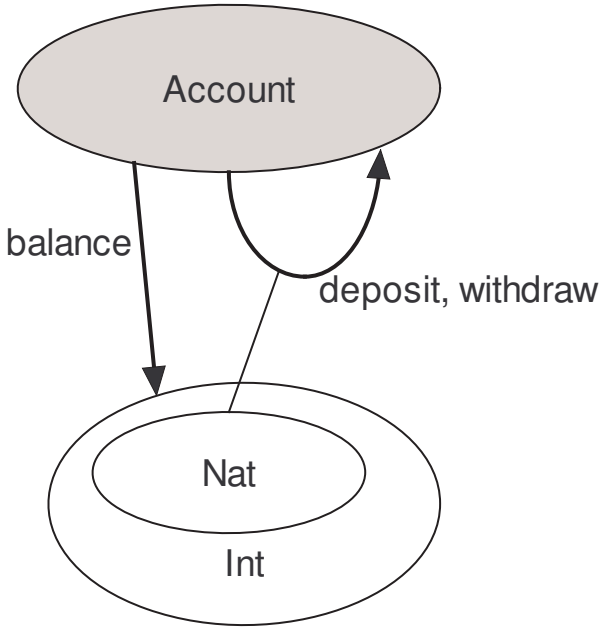


Figure 2.8: Bank account abstract machine

We specify a bank account object (abstract machine) in Figure 2.8. The (set of) states of the account abstract machine form a hidden sort Account. The account abstract machine has two actions: deposit and withdraw, and one observation balance. Hidden sorts are represented as gray ellipsoidal disks, actions and observations are represented by thick arrows.

The CafeOBJ specification of the bank account abstract machine is as follows:

```
mod* ACCOUNT {
  protecting(INT)  -- protecting means to import the designated module
                   -- INT (a built-in integer module in CafeOBJ), it
                   -- is often abbreviated as "pr".
  *[ Account ]*    -- hidden sort
  bop balance : Account -> Nat                      -- observation
  bops deposit withdraw : Account Nat -> Account  -- action
  var N : Nat  -- Nat is a built-in visible sort in CafeOBJ
  var A : Account

  eq balance(deposit(A,N)) = balance(A) + N .
  ceq balance(withdraw(A,N)) = balance(A) - N if N <= balance(A) .
  ceq balance(withdraw(A,N)) = balance(A)     if balance(A) < N .
}
```

In the above example, observation `balance` is defined as `bop balance : Account -> Nat`, where `Account` is the hidden sort denoting the state space of the account OTS, and `Nat` is the visible sort denoting natural numbers. Transitions rules `deposit` and `withdraw` are defined as actions to change the object state. Sometimes we need a operation to denote

any initial state, such as `op init : -> Sys`, where `Sys` is a hidden sort denoting the state space of an OTS (in this example we have no).

In conclusion, in description of an OTS in CafeOBJ, we first write the signature of the specification of the OTS, declaring sorts and operations, next write equations defining the initial values of the observations, and then write equations defining how a state of the OTS changes after each action is executed in that state [OF02].

Chapter 3

Formal Fault Tree Construction and Analysis

3.1 Incorrect Problem Analysis

As we discussed in the Introduction Chapter, traditional formal fault tree analyses often focused on providing formal semantics for fault tree constructors, such as different logic gates; and the common method is to develop the formal model and the fault trees as *separate* documents. That is to say, building up the fault trees is driven by intuition, while the events and sub-events of a gate are formalized afterward with respect to the formal model [RST00]. This approach is effective for quickly constructing a fault tree, but the informal construction creates problems later, when verifying its correctness.

For example, in [STR02], Schellhorn et al. noticed that sometimes the correctness condition was unprovable, in that the sub-events could not formally result in the top event through a logic gate. They found that such gates typically represent design decisions, in which all states described by the sub-events were considered already faulty regardless of side conditions. This fact has led traditional formalization attempts of fault trees either to require only the completeness condition [HRS94] (i.e. neglect the correctness condition in this case), or make the side conditions explicit in the fault tree by introducing an INHIBIT-gate [STR02]. But in that case, *finding* and confirming such side conditions may become another problem, especially if the gate is in the bottom layer of a large and complex informal fault tree. Furthermore, how can we ensure such side conditions are really *necessary*?

As an example, in order to better understand the problems caused by traditional formal FTA construction methods, we use a radio-based crossing control system as an running example, and its informal description are as follows.

3.1.1 The Radio-based Crossing Control System

The German railway organisation, Deutsche Bahn, uses a novel technology to control level crossings: the de-centralized radio-based level crossing control [bet96]. An overview of this system is given in Figure 3.1, and its brief informal description is as follows [RST00, STR02].

Shortly before the train arrives at the 'latest braking point' (latest point at which it

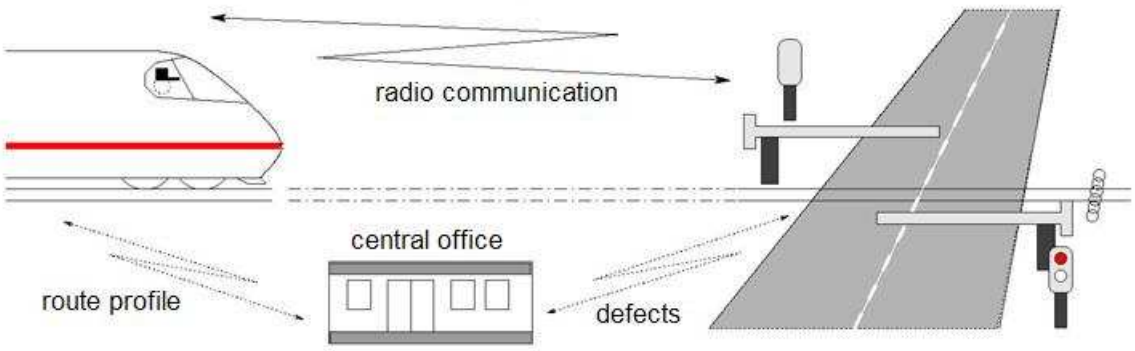


Figure 3.1: Radio-based crossing control system

is possible for the train to stop before the crossing), it sends a '*secure*' signal to the level crossing in order to check the status of the crossing. When the level crossing receives the command 'secure', it switches on the traffic lights, and then closes the barriers. After they have been closed, the level crossing is safe for a certain period of time and a '*release*' signal is sent to the train, which indicates that the train may pass the crossing. The '*stop*' signal on the train route, indicating an insecure crossing, is also initiated by computation and communication.

After the train has passed the crossing, it sends back a '*passed*' signal, which allows the crossing to open the barriers and switch back to its initial state. If no signal is received, the crossing waits for some minutes and then opens the barriers to protect cars against endless waiting (and is then unsafe).

The level crossing periodically performs self-diagnosis and automatically informs the central office about defects and problems. The central office is responsible for repair and for providing route descriptions for trains. These descriptions indicate the positions of level crossings and maximum speed on the route.

The main *difference* between this technology and traditional level crossing controls is that signals and sensors along the route are replaced by radio communication and software computations in the trains and at the level crossings. So, some system safety-critical issues are shifted from *hardware* to *software* while cheaper and more flexible solutions are offered. But this also brings out another important safety issue: we can use fault tolerance technology or spare parts to prevent hardware failures, but now software defects found in the fault tree should be taken more seriously. To prevent such software failures, we should first revise the system design, and then use formal methods to prove that those failures will be prevented in the revised system. This is exactly what we discussed in the Introduction section and will analyze in detail below.

3.1.2 Problem Analysis

Based on the introduction above, let's first present a fault tree model of the crossing control system, using the traditional FTA method (see Figure 3.2, which is taken from [STR02]).

To avoid a collision hazard, we can construct an informal fault tree by intuition as shown in Figure 3.2. The method for drawing this fault tree is as follows. First, we define

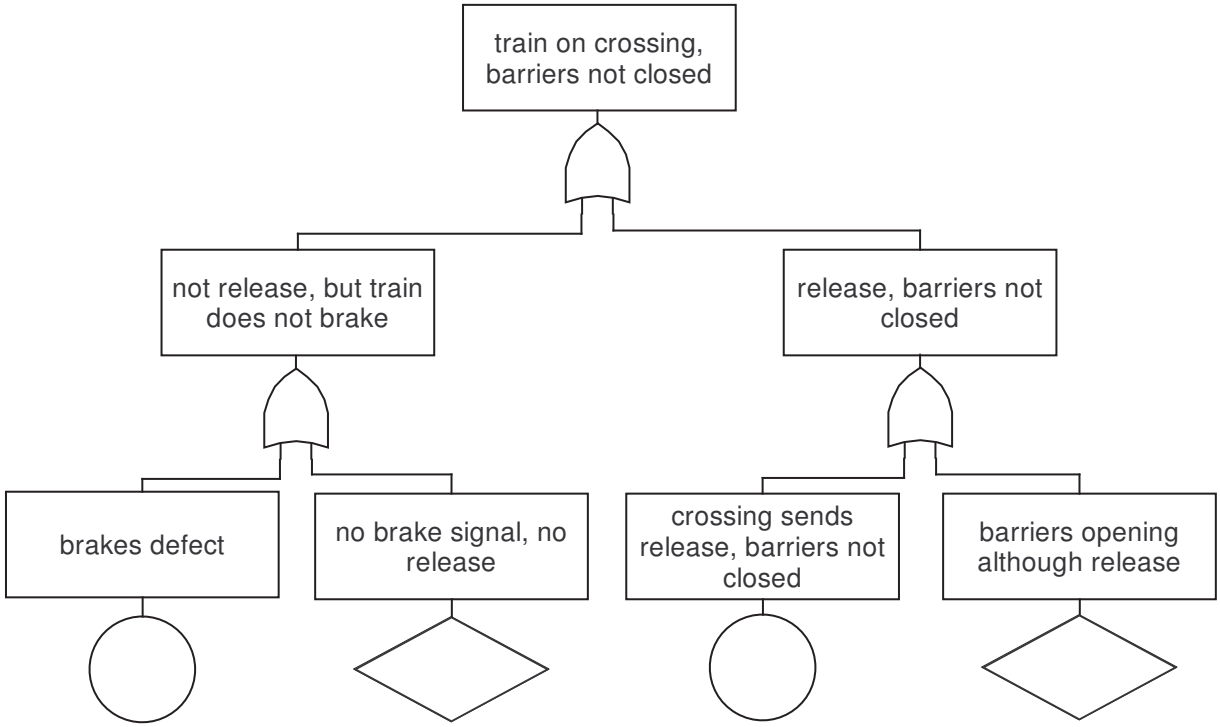


Figure 3.2: FTA for the hazard collision

the top event, i.e. collision as *train on crossing, barriers not closed*. This event occurs if either the train does not brake before the crossing despite it has no 'release' signal, or the crossing does send a 'release' signal regardless of that the barriers are not closed. The right sub-event is then further attributed to two possibilities. One is that when the crossing sends the 'release' signal, the barriers are open; the other is that the barriers are opening even the 'release' signal has been sent because of timeout or receiving a 'passed' signal (not further developed here and represented as an undeveloped event in Figure 3.2). The left sub-event has again two reason. Either the brakes are defective or they have not received a 'brake' signal from the train control, and so on.

As mentioned before, the traditional method for building a fault tree is efficient and the result is easy to understand and communicate. But at the same time, it causes some hidden problems for subsequent formal analysis. In this example, Schellhorn et al. found that the correctness of the topmost OR-gate was unprovable, where the formal specification of the top event is:

$$OnCrossing(tr) \wedge \neg Closed(ba) \tag{3.1}$$

While the right sub-event is defined as:

$$Release(tr) \wedge \neg Closed(ba) \tag{3.2}$$

where 'tr' and 'ba' denote 'train' and 'barriers' respectively.

Intuitively, we could not prove the correctness condition of this OR-gate, namely "(3.2) $\Rightarrow$ (3.1)", because (3.2) lacks the side-condition "$OnCrossing(tr)$" and is not

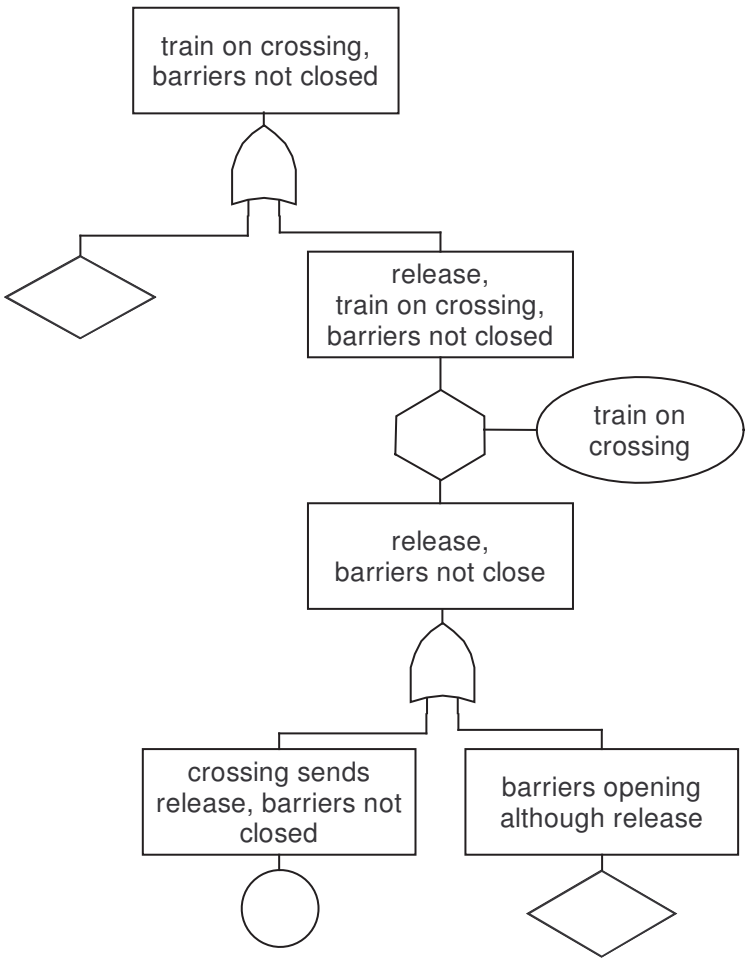


Figure 3.3: Revised fault tree for hazard collision

enough to imply (3.1). To correct this problem, Schellhorn proposed explicitly making this point in the fault tree by introducing an INHIBIT-gate between (3.1) and (3.2) with the side condition "$\chi := OnCrossing(tr)$" (the revised fault tree is shown in Figure 3.3)[STR02].

Note that we used the word '*intuitively*' in the last paragraph, because we think that, strictly speaking, in this example (3.2) itself is enough to imply (3.1); we will give our detailed proof in the next Section. Of course, Schellhorn's solution to this problem is instructive. But even so, other problems may arise: How can we discover all of the side-conditions, especially in a large and complex fault tree? Furthermore, how can we ensure that such side-conditions are necessary?

Another problem in Figure 3.2 is whether the basic event, "brakes defect," is enough to cause the top event, "collision" to occur. What if, in such conditions, the train receives the 'release' signal and the barriers are closed? We know in this case the collision will not occur, whether the brakes are defective or not. This ex-ample may be quibbling, because obviously "brakes defect" is an unacceptable faulty event and we could not let it occur; it could be explained as the sort of design decision we mentioned in the Introduction section. But as we know, in safety-critical system analysis, we can not neglect any minor side-conditions, even to a faulty event; to do so could result in major hazards or interfere

with our taking accurate and correct precautions.

After careful study, we find that the incorrectness problem stems from how to decompose a complex event consisting of a combination of several sub-formulas. As we know, FTA was originally developed to analyze hardware systems; in that case, it is quite easy to directly decompose a hardware component failure into some sub-component failures. But in software and integrated system analysis, this situation changes. This is illustrated in (3.1), in which the collision was defined as a conjunct of two sub-events; in this case, we can not simply decompose it by an AND-gate since each sub-event alone no longer describes a fault event. This is also our motivation for this study. Our formal fault tree construction model, illustrated below, can effectively solve these problems.

3.2 Formal Fault Tree Construction based on Temporal Logic

This section presents our formal fault tree construction model. It is based on a temporal logic variant of the regression procedure found in [vL91], which has also been widely applied for obstacles analysis in goal-oriented requirements engineering [vLL00, vLDL98].

3.2.1 Basic Concepts of Domain Rule

First, we define an important term in our model, namely domain rule. A domain rule is a rule about objects or operations in the environment which holds independently of the software-to-be. More specifically, in our model, it is an indicative statement or formula of domain knowledge with respect to an occurrence of event(s). For example, with respect to (3.1) in Section 3.1.2, i.e., $OnCrossing(tr) \wedge \neg Closed(ba)$, one of the domain rules as for the event $OnCrossing(tr)$ is that:

$$BypassSignal \vee BrakeFailure \vee Release(tr) \Rightarrow \Diamond OnCrossing(tr)$$

which states that either the driver/train bypasses a 'stop' signal (illegal driving or misoperation), or there is a hardware failure in the brake of the train (physical failure), or the train receives the 'release' signal from the crossing (system design), the train will *eventually* run on the crossing. After finding such a domain rule, then we can use it to formally deduce the corresponding sub-events of (3.1). In fact, the process of formal fault tree construction is guided by the domain rules to decompose the top event into sub events.

Before discussing how to decompose the fault events using transition rules, let's briefly review the monotonicity of temporal logic (see Section 2.2, page 21) as follows.

Let $\varphi(u)$ be a formula scheme with one or more occurrences of the sentence symbol u, if all occurrences of u in $\varphi(u)$ are positive [1], then $(p \Rightarrow q) \rightarrow (\varphi(p) \Rightarrow \varphi(q))$ is valid.

Here, if we regard the domain rule as $p \Rightarrow q$, and the fault event as $\varphi(q)$, then we can reduce the sub-event $\varphi(p)$ by replacing every occurrence of q in the fault event with p, and the correctness of the reduction/decompositon is guaranteed.

[1] An occurrence of u to be *positive* (*of positive polarity*) in φ if it does not occur in a subformula of the form $p \leftrightarrow q$ and it is embedded in an even (explicit or implicit) number of negations.

To illustrate this point, considering the above example, first we rewrite the domain rule into the following equivalent form:

$$\diamondminus\,(BypassSignal \vee BrakeFailure \vee Release(tr)) \Rightarrow OnCrossing(tr)$$

where the temporal operator $\diamondminus$ means some time in the past or once. Then we can reduce the sub-event as for the hazard collision as follows:

$$\diamondminus\,(BypassSignal \vee BrakeFailure \vee Release(tr)) \wedge \neg Closed(ba)$$

which can be further reduced into three sub-events connected by a OR-gate with the top event. There is a trivial trouble in the above formula, that is, the temporal operator $\diamondminus$ is introduced, and it may cause inconvenience for the understanding of the sub-events and subsequent reduction.

There are two solutions to this problem, one is to use the temporal entailment $p \Rightarrow \diamondminus\, p$ and reapply monotonicity so as to eliminate the temporal operator $\diamondminus$ as we once used in [XFH04a]. However, this solution is based on the assumption that p may hold in the current state, and the final result does not represent the time-delay between the sub-events and event, such as $Release(tr) \wedge \neg Closed(ba) \Rightarrow OnCrossing(tr) \wedge \neg Closed(ba)$, which may cause confusing even the logic deduction is correct. Therefore, in this thesis, we introduce another method based on the concept of constraint of transition rule as follows.

Rule 3.1 (Constraint of Selection of Domain Rule) *: Given a fault event R, a domain rule D selected to deduce R should obey that:* $\{R, D\} \nvDash false$.

More specifically, suppose $D \stackrel{def}{=} p \Rightarrow \diamondsuit\, q$, where q matches the sub-formula L of R, then p can not imply any negation of the other sub-formulas of R. In other words, for any other sub-formula of R, namely M, $p \wedge M \Rightarrow \diamondsuit\, M$.

The reason is obvious, if $\{R, D\} \vDash false$, then we will get some *false* fault sub-events, i.e., correct events. Once this happens, we should consider and find another domain rule for M instead of L. A special case is that there is no domain rule complying with the constraint, then the fault event R is a false fault event, and need not be further analyzed and considered in the fault tree.

To illustrate this point, let's return back the above example, since $(BypassSignal \vee BrakeFailure \vee Release(tr)) \wedge \neg Closed(ba) \Rightarrow \diamondsuit\, Closed(ba)$, then we can further deduce the sub-event into the following form:

$$\diamondminus\,((BypassSignal \vee BrakeFailure \vee Release(tr)) \wedge \neg Closed(ba))$$

Moreover, if we introduce temporal semantics to the logic gates (OR-gate in this example) [BA93], i.e., given a logic gate with some inputs (sub-events) and an output (top event), its correctness and completeness conditions are as follows:

Correctness Condition : $inputs \Rightarrow \diamondsuit\, output$

Completeness Condition : $output \Rightarrow \diamondminus\,(inputs)$

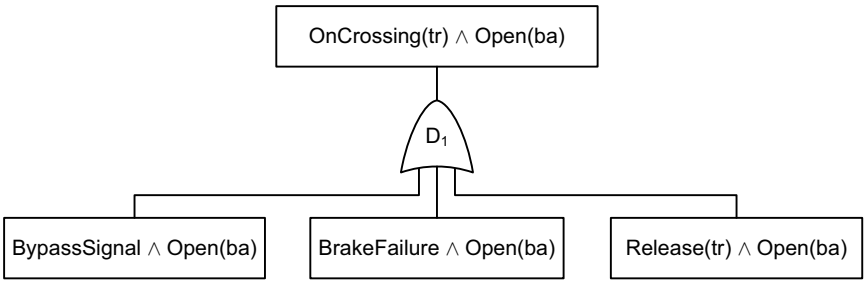


Figure 3.4: A formal fault tree for hazard collision

In this thesis, we call the above gates as Delay-gate (Short: D-gate) so as to distinguish it from the standard gates of fault tree analysis, which are usually defined by Boolean logic without the time-delay concept. And for simplification, we do not invent new gate symbols for the D-gates since the time-delay has already been denoted by the corresponding domain rules. Therefore, to distinguish a gate is a D-gate or a standard gate, we need only check whether there is a time-delay between the antecedence and consequence of the corresponding domain rule.

Then we can eliminate the temporal operator $\diamondsuit$ in the sub-event for better understanding and readability and get a fault tree connected by an OR-gate as shown in Figure 3.4, where the domain rule is recorded on the OR-gate and denoted by D_1.

Notice here we have answered the question posed in Section 3.1.2, namely, why we think that (3.2) itself is enough to logically imply (3.1), i.e., $Release(tr) \wedge \neg Closed(ba) \Rightarrow \diamondsuit (OnCrossing \wedge \neg Closed(ba))$. The point is that, by introducing monotonicity for event decomposition and temporal semantics to the logic gates, we can efficiently solve the incorrectness problem and avoid introducing unnecessary side-conditions.

3.2.2 Formal Fault Tree Construction Model

Based on the above discussion, we bring out our formal fault tree construction model as follows.

We take a catastrophic failure as the root node of the fault tree, namely R. Assume the necessary domain rule D for R or the sub-formulas in R take the form of $A \Rightarrow \diamondsuit C$, the regression procedure for the formal fault tree construction is as follows.

Initial step Define the formal specification of R.

Inductive step

Let $A \Rightarrow \diamondsuit C$ be the domain rule selected, with C matching the sub-formula L in R whose occurrences in R are all positive, and $\{D, R\} \nvdash false$;

Then $R := R(A/L)$, where $R(A/L)$ denotes substituting all the occurrences of L in R with A.

Iteration step

(1) Decompose the resulting R to some sub-events by an *appropriate* logic gate;

(2) Integrate and record the corresponding domain rule into the logic gate for further revising and rechecking of the correctness and completeness of the fault tree (Notice: the transition rule itself is not a sub-event, and the relationship between the domain rule and R should not be simply denoted as a kind of "AND-gate" as was used in [vLL00], our notation is shown in Figure 3.4);

(3) Then for each sub event, redo the inductive and iteration steps recursively until a basic event or the chosen abstraction level is reached.

To help engineers construct the fault tree more efficiently, we present two important general guidelines for selecting the "appropriate logic gate" in iteration step (1) as follows.

- First, the results of the decomposition should be meaningful *fault* event, and the logic gate selected should be as simple a standardized one as possible, for better understanding and readability of the fault tree figure. This decision is made by domain experts according to the logic structure of R as well as its actual meaning/semantics. Sometimes INHIBIT-gate plays an important role to represent the normal events as conditions rather than fault events as we will discuss later.

- Second, since one important goal of FTA is to find minimal cut sets, where each minimal cut set consists of the basic events of one conjunction in the disjunctive normal form (DNF) of the resulting formula, we recommend transforming R into DNF before the decomposition if possible. This can help us find more single faulty events and simplify the succeeding analysis. (we implement this automatic transformation in the system built-in modules of the fault tree specifications in next section).

To better understand this procedure, we can use the crossing control example from Section 3.1.1 as an illustration. In the crossing control system, the root node of collision is formally defined as:

$\mathbf{R} : OnCrossing(tr) \wedge Open(ba)$

Based on the domain rule introduced above,

$\mathbf{D}_1 : (BypassSignal \vee BrakeFailure \vee Release(tr)) \Rightarrow \Diamond\, OnCrossing(tr)$

We can get three sub-events connected by an OR-gate as follows.

$\mathbf{S}_{1a} : BypassSignal \wedge Open(ba)$

$\mathbf{S}_{1b} : BrakeFailure \wedge Open(ba)$

$\mathbf{S}_{1c} : Release(tr) \wedge Open(ba)$

Since $BypassSignal$ itself is fault event regardless of $Open(ba)$, then $\mathbf{S_{1a}}$ can be further decomposed into a basic event $BypassSignal$ connected by an INHIBIT-gate with condition $Open(ba)$. Notice here, we can not use an AND-gate to decompose S_{1a}, since the event $Open(ba)$ is not a fault event but a normal state of the barriers. The *difference* between INHIBIT-gate and AND-gate is that the condition in INHIBIT-gate is allowed to describe any set of states, it is not necessary to require it is a fault, otherwise we would use an AND-gate. Also, the conditions will not occur in the minimal cut sets, thus it

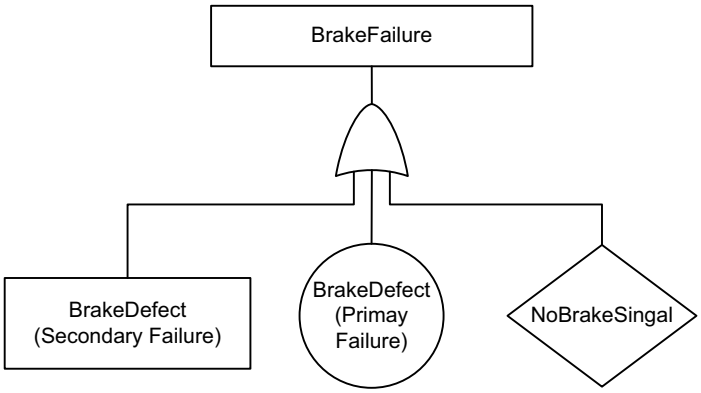


Figure 3.5: A fault tree of BrakeFailure

simplifies the problem and helps us focus on the simplified basic events.

The same can be done as to $\mathbf{S_{1b}}$, and with respect to the fault event $BrakeFailure$, since it is a component failure, based on the fault tree construction rule introduced in Section 2.1.2 (Rule 2.2, page 15), it can be directly resolved into three failure modes, i.e., primary, secondary and command modes ($NoBrakeSignal$), by adding an OR-gate below it as shown in Figure 3.5. But in this analysis, we limited ourselves at this abstraction level and regarded it as a basic fault event.

Then we focus on $\mathbf{S_{1c}}$, a domain rule to cause the train get a 'release' signal, i.e., $Release(tr)$, is that the level crossing has sent a 'release' signal and there is no radio communication failure between them (suppose the radio communication time can be eliminated or it is not an important factor should be considered in this case).

$\mathbf{D}_2 \ : \ Release(cr) \wedge \neg RadioFailure \Rightarrow \ \Diamond \, Release(tr)$

And from $\mathbf{D_2}$, we get the sub-event $\mathbf{S_2}$ as follows.

$\mathbf{S}_2 \ : \ Release(cr) \wedge \neg RadioFailure \wedge Open(ba)$

A subtle issue disclosed in $\mathbf{D}_2$ is that the radio communication failure will not cause the hazard collision in this case. Since the event $\neg RadioDefect$ itself is not a fault event, an INHIBIT-gate could also be introduced here in which $\neg RadioDefect$ is regarded as a condition rather than a fault event, and $\mathbf{S}_2$ is then resolved as:

$\mathbf{S}_{2a} \ : \ Release(cr) \wedge Open(ba)$

Focused on $Open(ba)$, we can derive another domain rule as follows.

$\mathbf{D}_3 \ : \ (\boxminus_{\geq d} Closed(ba) \vee Idling(sys) \vee Passed(cr)) \Rightarrow \ \Diamond \, Open(ba)$

which states that three possibilities for the barriers to open are either the crossing has been closed and waiting for the '*passed*' signal over a designed time d (and then opens the barriers to protect cars against endless waiting), or the system is in idling status (no train ask for pass), or the crossing has just received a '*passed*' signal from a train. Use this transition rule to regress $\mathbf{S_{2a}}$, we get three corresponding sub-events connected by an OR-gate below.

$\mathbf{S}_{3a} \ : \ Release(cr) \wedge \ \boxminus_{\geq d} Closed(ba)$

$\mathbf{S}_{3b}$: $Release(cr) \wedge Idling(sys)$

$\mathbf{S}_{3c}$: $Release(cr) \wedge Passed(cr)$

Notice here a subtle system safety design issues is exposed by $\mathbf{S}_{3c}$, which states that the two events, $Release(cr)$ and $Passed(cr)$ can not occur simultaneously, in other words, if the crossing sends a 'release' signal to a train and let it to pass the crossing, it can not receive a 'passed' signal from another train at this moment. More specifically, it implies that two trains may enter the crossing area at the same time.

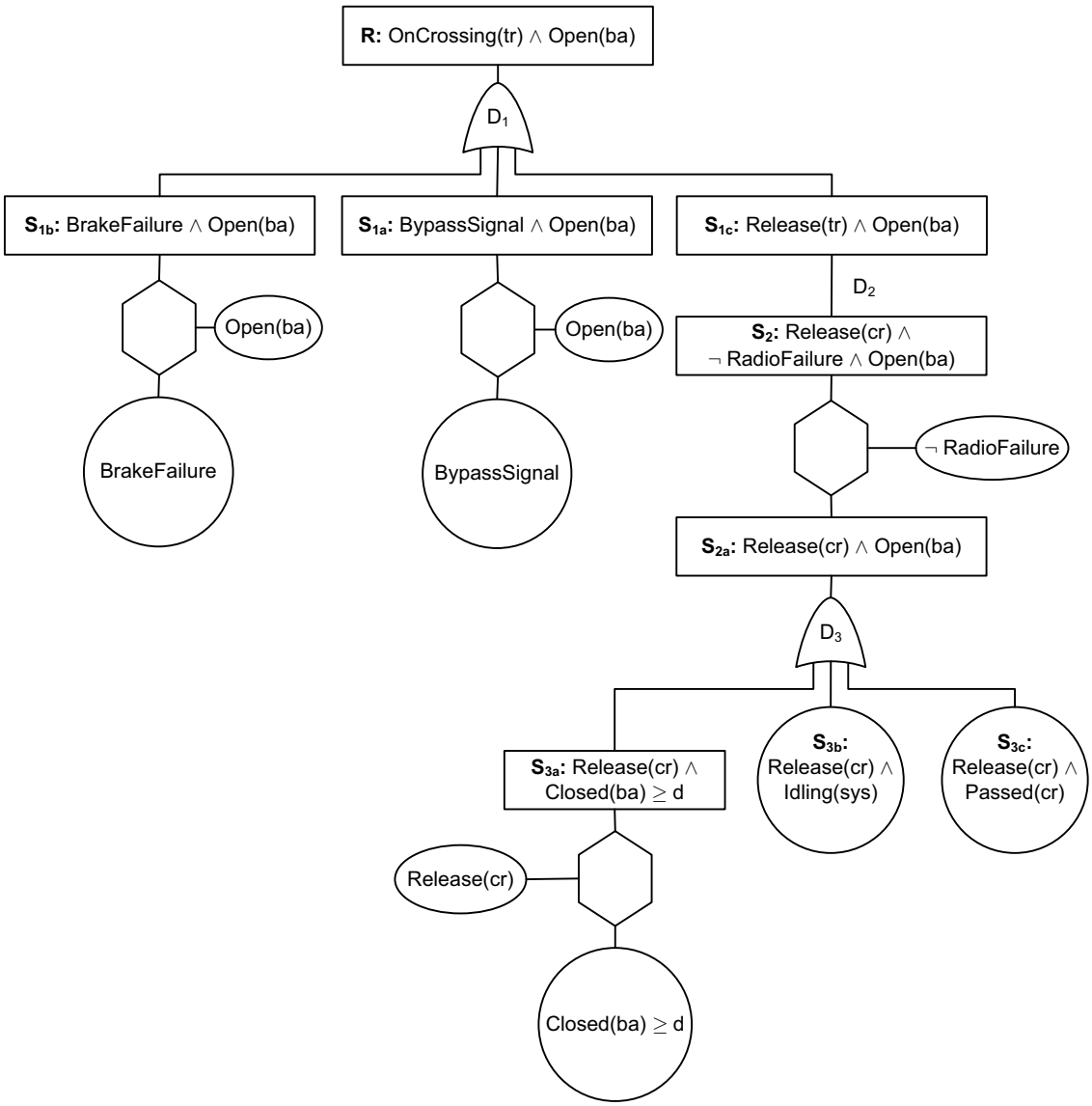
In $\mathbf{S_{3a}}$, the event $\boxminus_{\geq d} Closed(ba)$ can be regarded as basic fault event, and to solve it calls for human intervention instead of software or hardware, since it is a conflict between convenient design and the overriding system safety goal. Here, an INHIBIT-gate can also be introduced with the condition $Release(cr)$ to simplify the fault tree. With respect to the events $\mathbf{S_{3b}}$ and $\mathbf{S_{3c}}$, they belong to design failures and should be prevented and verified in the subsequent system design and formal specification. If we choose this abstraction level to stop our analysis, the whole formal fault tree is shown in Figure 3.6.

As discussed above, our formal construction model is a deductive method which enables us to build the fault tree more precisely and completely. It not only is useful for finding faulty events while avoiding the incorrectness problems inherent in traditional methods, but also guides designers and domain experts to elicit the domain rules incrementally.

Traditional methods often build informal fault trees by intuition, then try to separately establish the formal semantics for the fault tree constructs and check the correctness and completeness conditions. Our method guarantees the correctness of the fault trees by the construction process itself, making it more effective and precise. For instance, in our example the fault event $\mathbf{S_{1b}}$ reveals that: the brake failure can result in a collision only when the barriers are open; otherwise, the brake failure itself is not enough to cause the collision (Notice, even in our example, we further resolved this event into a basic event $BrakeFailure$ and a condition $Open(ba)$ connected by an INHIBIT-gate, but it does not mean that $BrakeFailure$ itself is enough to cause collision based on the semantics of INHIBIT-gate, that is, $BrakeFailure$ is only the necessary but not sufficient condition). As discussed in Section 3.1.2, this issue may be trivial in this case, but in other safety-critical systems, it is very important to define each event accurately and precisely.

3.3 From Safety Analysis to System Design

Traditional fault trees are used to analyze existing system design with respect to safety prosperities. Instead of first developing a detailed design, and then performing FTA, we propose that FTA and system design should proceed concurrently and complement each other. Actually, during fault tree construction, the derived domain rules can be used to guild the system design and subsequent formal system specification. And the incremental process of discovering domain rules itself provides an instructive approach to understand the system more comprehensively step by step as we discussed in the last section. In this section, inspired by Hansen's work [HR98], we focus on how to further derive safety assumption and commitment so as to achieve system safety requirement from the fault trees.



Illustrations :

$\mathbf{D}_1$: $(BypassSignal \vee BrakeFailure \vee Release(tr)) \Rightarrow \Diamond\, OnCrossing(tr)$

$\mathbf{D}_2$: $Release(cr) \wedge \neg RadioFailure \Rightarrow \Diamond\, Release(tr)$

$\mathbf{D}_3$: $(\boxminus_{\geq d} Closed(ba) \vee Idling(sys) \vee Passed(cr)) \Rightarrow \Diamond\, Open(ba)$

Figure 3.6: Formal fault tree of collision based on temporal logic

Definition 3.1 (Safety Requirement) *For each fault tree in which the root is interpreted as R, the system should be built to ensure that R never occurs, i.e., the* safety requirement *of the system is:*

$$\Box\, \neg R \tag{3.3}$$

In case there are n fault trees in which the roots are interpreted as $R_1, \ldots, R_n$, then the corresponding safety requirement deduced from these fault trees is

$$\Box\, \neg R_1 \wedge \ldots \wedge \Box\, \neg R_n$$

i.e., the system should ensure that no top event in any fault tree ever occurs. This corresponding to combining the fault trees by an OR-gate [HR98].

To achieve the safety requirement of the system, $\Box\, \neg R$, it is quite naturally to deduce the corresponding original safety commitment from the minimal cut sets of the fault tree, where the definition of the original safety commitment is as follows.

Definition 3.2 (Original Safety Commitment) *For each fault tree in which the minimal cut sets is interpreted as a disjunctive normal form, $MCS_1 \vee \ldots \vee MCS_n$, the system should be implemented and verified to guarantee that each $MCS_i (i = 1, \ldots, n)$ never occurs, i.e., the* original safety commitment *of the system is*

$$\Box\, \neg MCS_1 \wedge \ldots \wedge \Box\, \neg MCS_n \tag{3.4}$$

Therefore, the relationship between the original safety commitment and safety requirement is that:

$$\Box\, \neg MCS_1 \wedge \ldots \wedge \Box\, \neg MCS_n \Rightarrow \Box\, \neg R \tag{3.5}$$

which is supported by the fact, i.e., suppose each domain rule used for the fault tree construction is *sufficiently complete*, then we can derive the following formula:

$$R \Rightarrow \diamondsuit MCS_1 \vee \ldots \vee \diamondsuit MCS_n \tag{3.6}$$

A difference with Hansen's work [HR98] should be figured out is that, instead of deriving the global safety commitment gradually from local commitments of individual components , i.e., logic gates such as AND-, OR-, and PRIORITY AND-gates, we propose to deuce the *original* safety commitment from the minimal cut sets of the fault tree directly. And actually, the summarizing process of local commitments itself consists of the calculation of minimal cut sets as used in [HR98]. Therefore, we do not introduce the concept of local commitment in our approach since it would make the calculation of system safety commitment more complex.

Moreover, (3.5) can be further transformed into another form by using the concepts of minimal path sets. Suppose we can derive m minimal path sets (MPS) from the resulting formula of minimal cut sets, then we can get:

$$\Box\, MPS_1 \vee \ldots \vee \Box\, MPS_m \Rightarrow \Box\, \neg R \tag{3.7}$$

where each MPS_j $(j = 1, \ldots, m)$ consists of a conjunction of non-occurrences of primary fault events $b_{j_1}, \ldots, b_{j_l}$ (suppose there are l primary events in MPS_j) in a form of

$$\neg b_{j_1} \wedge \ldots, \wedge \neg b_{j_l}$$

Therefore, (3.7) provides a kind of checklist for system safety and reliability evaluation, that is, if we can ensure that all the non-occurrences of primary events of some minimal path sets, the the system is safe.

The reason why we called (3.4) as the *original* safety commitment of the system is that, in practice, some basic fault events of the minimal cut sets are *uncontrollable* or can not be implemented in the system (software) design, such as the failures related to hardware defects or human errors. To this end, these failures should be separated from the original safety commitment, and a concept of safety assumption should be introduced as follows.

Definition 3.3 (Safety Assumption) *Basic fault events that can not be prevented by system design and implementation should be regarded as uncontrollable failures, and the negation of these basic fault events should be regarded as* safety assumption *rather than safety commitment of the system.*

More specifically, three different failures can be added to the list of safety assumptions as follows.

- *Physical failures*, such as hardware component defects that are uncontrollable and unavoidable with respect to software design and implementation, e.g., the event $BrakeFailure$ in the fault tree of crossing control system.
- *Undeveloped events* need *human intervention* as for the solution, such as the event $\boxminus_{\geq d} Closed(ba)$.
- *Errors and misoperations of human being* that are usually prevented by corresponding safety regulation rather than system design. For example, with regard to the event $BypassSignal$, a corresponding safety regulation $\Box \neg BypassSignal$ should be made and added to the list of system safety assumption.

Therefore, after removing the uncontrollable failures from the original safety commitment, we can derive a list of safety assumption (SA) and get a refined safety commitment (SC) of the system, i.e., safety properties only related to system design and implementation. And the system safety requirement (SR) can be achieved and guaranteed by the following statement.

$$SR \stackrel{def}{=} (SA \Rightarrow SC) \tag{3.8}$$

The point is that, by deriving and distinguishing safety assumption and commitment from the fault tree, designers can get the following two important knowledge as for system design, implementation, and verification. On one hand, having an explicit assumption list and adding to this list during fault tree construction can help the designers understand the system safety more comprehensively, while such an assumption list is usually overlooked or difficult to discover if we start to design the system directly without FTA or other hazard analysis techniques. This is because from the point of view of system design, people usually focus on how to ensure the system will perform successfully, and thus

some physical failures or human errors are easily be overlooked or not considered into the system design and specification. Consequently, the corresponding safety regulations and fault-tolerant precautions will be ignored. This may cause some unexpected hazards even catastrophes, especially with respect to safety-critical systems. On the other, after deriving the refined safety commitment, the designers can have definite object in view as for more efficient system design and safety verification, this is because generally speaking, the basic events of a fault tree (in the refined safety commitment) are *simpler*, more *specific* and *manageable* compared with the top hazard event (root). Therefore, not only specific design issues are discovered, but also time and cost can be saved with respect to the formal verification of safety properties.

To illustrate this point, considering the crossing control system example as shown in Figure 3.6. The safety requirement (SR) of the fault tree is to ensure that no collision occurs, i.e.,

$$SR \stackrel{def}{=} \Box\,(OnCrossing(tr) \rightarrow Closed(ba))$$

which states that when the train is on crossing, the barriers must always be closed.

The minimal cut sets (MCSs) of the fault tree can be derived in a disjunctive normal form (DNF) as follows.

$$\begin{aligned} MCSs \stackrel{def}{=}\ & BrakeFailure \vee \\ & BypassSignal \vee \\ & \boxminus_{\geq d} Closed(ba) \vee \\ & (Release(cr) \wedge Idling(sys)) \vee \\ & (Release(cr) \wedge Passed(cr)) \end{aligned}$$

Therefore, the original safety commitment (OSC) can be defined as:

$$\begin{aligned} OSC \stackrel{def}{=}\ & \Box\,\neg BrakeFailure \wedge \\ & \Box\,\neg BypassSignal \wedge \\ & \Box\,\neg(\boxminus_{\geq d} Closed(ba)) \wedge \\ & \Box\,\neg(Release(cr) \wedge Idling(sys)) \wedge \\ & \Box\,\neg(Release(cr) \wedge Passed(cr)) \end{aligned}$$

Then, based on the properties of different failures, we can derive the safety assumption (SA) and refined safety commitment (SC) that should be further implemented and verified in the system design as follows.

$$\begin{aligned} SA \stackrel{def}{=}\ & \Box\,\neg BrakeFailure \wedge \\ & \Box\,\neg BypassSignal \wedge \\ & \Box\,\neg(\boxminus_{\geq d} Closed(ba)) \end{aligned}$$

$$\begin{aligned} SC \stackrel{def}{=}\ & \Box\,\neg(Release(cr) \wedge Idling(sys)) \wedge \\ & \Box\,\neg(Release(cr) \wedge Passed(cr)) \end{aligned}$$

And the system safety requirement can be achieved and guaranteed by the following statement.

$$SR \stackrel{def}{=} SA \Rightarrow SC$$

Based on the above discussion and analysis, two kinds of important knowledge have been discovered as for the system design and implementation as follows.

First, from the list of safety assumption, the system can only be safe in case there is no violation of the safety assumption, in other words, we know more clear and exactly about the limitations and *restrictions* as for the successful performing of the system. In addition, the corresponding precautions can be made in advance as follows.

- Fault tolerant technique or component should be introduced to prevent the brake failure, such as spare brakes.
- A safety regulation that drivers must obey the 'stop' signal should be established.
- In case the fault event $\boxminus_{\geq d} Closed(ba)$ occurs, the system should be stopped until it has been solved by human intervention.

Second, from the refined safety commitment, the system must be designed to ensure that when the system is idling, it can not response and send a 'release' signal to a train. And we must ensure and verify that there is no mutual exclusion problem in the system, that is, when the crossing sends a 'release' signal to one train, it receives a 'passed' signal from another train. This is the topic that we will discuss in the next chapter, i.e., formal system modeling, specifying, and verifying with CafeOBJ [FS95, FN97, DF98] and Maude [CELM96, CDE+02, CDE+03a, CDE+03b].

3.4 Summary

In this chapter, focuses on the incorrectness problem of traditional FTA, we have presented a formal fault tree construction model, in which the correctness of the fault tree is guaranteed by the construction process and monotonicity of temporal logic. In addition, we have also discussed how to further derive safety requirement, safety assumption, and safety commitment from the fault trees in order to assist the system design, implementation and verification.

As far as we know, we are the first to explicitly and clearly determine the harm of the incorrectness problem that can be caused by traditional formal FTA, and we present a corresponding detailed analysis and solution to this problem.

Compared with traditional FTA [VGRH81]and some of its formal models [HR98, STR02], the advantages of our formal fault tree construction model are as follows.

- It is a deductive method to build fault trees more precisely and effectively;
- The correctness of our formal fault tree is proved by the construction process itself, thus avoiding the problems that often arise with traditional methods. At the same time, it gives designers and domain experts the ability to discover domain rules and important design principles in an instructive way during the construction process;

- We integrate the domain rules into the fault tree, which makes the formal fault tree more reliable and is useful for further revising and rechecking of the fault trees.

It should be noted that in this chapter we mainly focused on the correctness of the fault tree, which is guaranteed by our formal fault tree construction processes. With respect to the *completeness* of the fault tree, we agree with that "*completeness is a notion relative to what is known about the domain* [vLL00]" and it requires the designers to work together with the domain experts. However, This problem can be solved to some extent if we can ensure that for each domain rule we have considered *all* the possibilities with respect to its consequence, this is another advantage of our method and why we introduce and record the domain rules explicitly in the fault trees as for revising and rechecking.

It should be also figured out that the temporal regression procedure based on the monotonicity of temporal logic was first found in [vL91], and was also widely used in obstacle analysis [vLDL98, vLL00] . The reason we introduce this technique into FTA and propose a revised formal fault fault tree construction model rather than just using obstacle analysis is that these two techniques have different focuses and applications. A comparison of FTA and obstacle analysis is shown in Table 3.1.

	FTA	**Obstacle Analysis**
Focus	*Hazard analysis*: discovery basic fault events that will result in a specific system hazard.	*Goal-oriented requirements analysis*: given a design goal, deduce possible obstacles by temporal logic reasoning.
Graphical Support	Rich and standard graphical notations	Simple AND and OR-patterns
Logic Foundations and Requirements	• An event is an occurrence of a specific system state (Boolean predicates) • Simple and understandable results • *Lightweight* temporal logic deduction	• Obstacles are interpreted as temporal formulas • Precise but complex results • *Strict* temporal logic deduction, require advanced logic background
Qualitative & Quantitative Analysis	• Minimal cut sets and minimal path sets • System reliability calculation and allocation	The methods (algorithms) used in FTA may also be applicable to OA, however, it is not the main concern of OA.

Table 3.1: Comparison between FTA and obstacle analysis

As shown in Table 3.1, obstacle analysis focuses on formal reasoning about obstacles to the satisfaction of goals elaborated in the requirements engineering process. It is based

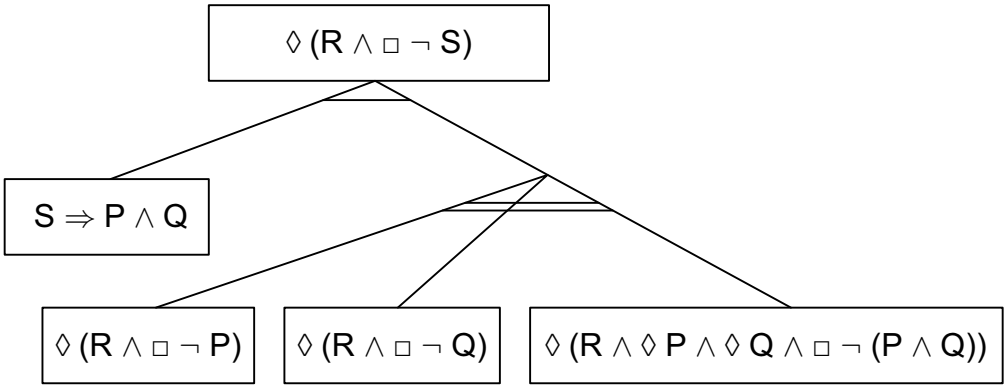


Figure 3.7: OR-refinement pattern for obstacles to the achieve goal

on a temporal logic formalization of goals and domain properties, and is integrated into an existing method for goal-oriented requirements elaboration with the aim of deriving more realistic, complete and robust requirements specifications [vLL00]. To better understand this technique as well as the difference with standard FTA, we use an example to illustrate the analyzing processes of obstacle analysis as follows.

Given an achieve goal, say $R \Rightarrow \Diamond S$, the regression procedure of obstacles to this goal is as follows:

1) Take the negation of the goal which yields:

   (NG) $\Diamond (R \wedge \Box \neg S)$

2) Suppose a domain theory (assertion) contains the following property:

   $$S \Rightarrow P \wedge Q$$

   And then a logically equivalent formulation is obtained by contraposition:

   (D) $\neg P \vee \neg Q \Rightarrow \neg S$

3) Regress (NG) through (D) by the monotonicity of temporal logic, and then derive the following potential obstacle:

   (O) $\Diamond (R \wedge \Box (\neg P \vee \neg Q))$

   This obstacle can be further decomposed into three sub-obstacles by an OR-refinement [vLL00] as follows, and the generated corresponding obstacle tree (pattern) is shown in Figure 3.7.

   (O1) $\Diamond (R \wedge \Box \neg P)$
   (O2) $\Diamond (R \wedge \Box \neg Q)$
   (O3) $\Diamond (R \wedge \Diamond P \wedge \Diamond Q \wedge \Box \neg (P \wedge Q))$

While in fault tree analysis, we need only focus on the hazard $\neg S$ because R is obviously not a fault event in this case (even we took $R \wedge \neg S$ as the top hazard event, we can

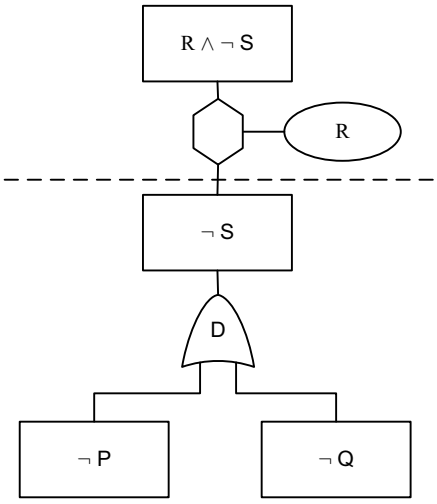


Figure 3.8: A fault tree of the achieve goal

use an INHIBIT-gate to simplify it as shown in Figure 3.8), and we do not care that the hazard $\neg S$ will always or eventually happen in the future, i.e., $\Box \neg S$ or $\Diamond \neg S$. Then based on the same domain rule (assertion) $\neg P \vee \neg Q \Rightarrow \neg S$, we can derive a corresponding fault tree as shown in Figure 3.8.

The main difference between Figure 3.8 and Figure 3.7 is that, in the convention of fault tree analysis, each node (fault event) is interpreted as *an occurrence of a specific system state*, which are usually denoted by simple Boolean predicates (formulas). Even in some current formal FTAs with temporal logic, such as [HR98, STR02], the temporal relationship between the event and sub-events are generally represented by the semantics of the gates, while not the event itself [2] (we will further discuss the temporal semantics of fault trees in Section 6.1). The advantage is that in the decomposition of fault events, we need not involve ourselves in some complex temporal logic deduction such as used in obstacle analysis, which starts with a temporal goal (formula), and then follows a strict temporal logic deduction and proving to derive the corresponding obstacles that are also interpreted as temporal formulas.

For example, as shown in Figure 3.7, the decomposition of obstacle **O** into the three sub-obstacles is a little bit difficult because the temporal HENCEFORTH operator $\Box$ is introduced, and it can not be distributed over disjunction $\vee$. And in some other cases when introducing more temporal operators such as Until Operator $\mathcal{U}$ and Waiting-for Operator $\mathcal{W}$ [MP92, vLL00], the deduction would be more complex. To solve this problem, it requires engineers with advanced temporal logic background to some extend (in [vLL00], it provides some refinement patterns which are verified by using STep [MtSG96] to assist the obstacle analysis). However, as we figured out above, such difficult issues are not the concern of fault tree analysis. And in our formal fault tree construction model, we introduce temporal logic only for the decomposition of fault events, and we introduce a method to intentionally remove the temporal operator from the resulting sub-events to the corresponding logic gates. This is the reason that we call the temporal logic deduction

[2] In this case, some temporal operators may be introduced to represent the events with time duration, such as discussed in [HR98]. However, these temporal operators are used mainly for more precise denotation rather than temporal logic deduction, i.e., they will not be involved in the subsequent deduction of sub-events.

in FTA is lightweight while in obstacle analysis it is strict (heavy).

Another interesting issue is that people may argue that the above example is a kind of liveness problem (because of the achieve goal $R \Rightarrow \Diamond S$), and it seems to be a bit unfair to use this example to compare FTA and OA since generally speaking, FTA is used mainly for safety related problems. This is a misunderstanding of definition of faults in FTA. We use a real-world example which is taken from [vLL00] to further illustrate this point.

Consider a meeting schedule system and the goal stating that intended people should participate to meetings they are aware of and which fit their constraints:

Goal	$Achieve[InformedParticipantsAttendance]$
FormalDef	$\forall m : Meeting, p : Participant$
	$Intended(p, m) \wedge Informed(p, m) \wedge Convenient(p, m)$
	$\Rightarrow \Diamond Participates(p, m)$

Then using obstacle analysis, we first take the negation of the goal and get a top obstacle (NG) as follows:

(NG) $\Diamond \exists m : Meeting, p : Participant$
$Intended(p, m) \wedge Informed(p, m) \wedge Convenient(p, m)$
$\wedge \Box \neg Participates(p, m)$

Suppose the domain theory contains the following property:

$\forall m : Meeting, p : Participant$
$Participates(p, m) \Rightarrow Holds(m) \wedge Convenient(p, m)$

This domain property states that a necessary condition for a person to participate in a meeting is that the meeting is being held and its data/locations is convenient to her. A logically equivalent formulation is obtained by contraposition:

(D) $\forall m : Meeting, p : Participant$
$\neg[Holds(m) \wedge Convenient(p, m)] \Rightarrow \neg Participates(p, m)$

Then we can formally derive the following sub-obstacle by applying the OR-refinement pattern of Figure 3.7:

(O1) $\Diamond \exists m : Meeting, p : Participant$
$Intended(p, m) \wedge Informed(p, m) \wedge Convenient(p, m)$
$\wedge \Box \neg Holds(m)$

(O3) $\Diamond \exists m : Meeting, p : Participant$
$Intended(p, m) \wedge Informed(p, m) \wedge Convenient(p, m)$
$\wedge \Diamond Holds(m) \wedge \Diamond Conveneient(p, m)$
$\wedge \Box \neg[Holds(m) \wedge Conveneient(p, m)]$

These sub-obstacles explains two situations, namely, one where some meeting never takes place and the other where a participant invited to a meeting whose data/location was first convenient to her is no longer convenient when the meeting takes places [vLL00]. They correspond to the first and third sub-nodes of the OR-refinement obstacle tree as

shown in Figure 3.7, the second sub-node, $\Diamond [R \wedge \Box \neg Q]$ is not considered because of the contradiction of $Convenient(p, m) \wedge \Box \neg Convenient(p, m)$ in the resulting formula.

The above example explains how OA works to solve the liveness problem. However, from the point of view of FTA, we need not to consider whether it is a liveness or safety problem, because we only concern with the *occurrence* of the undesired system *state*, $\neg Particiaptes(p, m)$, i.e., the state that the participant does not attend the meeting. And by applying the same domain rule D, we can directly conclude two basic fault events, $\neg Holds(m)$ and $\neg Convenient(p, m)$, in which complex temporal deduction are avoided.

In addition, as a widely used safety analysis technique which has been developed over 40 years, some other benefits of FTA can be briefly concluded as follows.

- FTA provides rich and standard graphical supports, not only limited to simple AND- and OR-patterns as used in obstacle analysis, which can help engineers understand and analyze the problem more clearly and efficiently, such as the INHIBIT-gate can be used to simplify the fault events as we discussed in Section 3.2.2. Some fault tree analysis programs (tools) have been developed [3], and Microsoft Visio2003 also provides a template for drawing fault trees [4].

- FTA has accumulated valuable experiences as for system analysis, such as the fault tree construction fundamentals introduced in Section 2.1.2.

- FTA can be used in not only qualitative analysis, but also quantitative analysis. The concept of minimal cut sets and algorithms for probabilistic analysis are very useful in system safety analysis. It should be figured out that these methods (algorithms) may also be applicable to obstacle analysis, however, this is not the main concern of obstacle analysis.

In short, our formal fault tree construction model can be regarded as an improvement and combination of these two techniques, which inherits the merits of standard FTA and draws on the idea of temporal regression procedure of obstacle analysis to achieve more efficient and reliable fault tree analysis.

[3] See http://www.enre.umd.edu/ftap.htm for the links of fault tree programs.
[4] The fault trees of this thesis are developed by Visio2003.

Chapter 4

Formal System Specification and Verification with FTA

As we discussed in the last chapter, engineers often seem to find formal deduction and proving tedious and difficult work, especially in complex system and fault tree analysis; thus an executable formal language to write the formal specifications of fault trees which also provides some automated logical deduction and proving mechanism is most desirable for them. At the same time, after using FTA to find the important system safety related problems, i.e., safety commitments, another inevitable and crucial problem is how to further revise the system design accordingly and formally verify that such safety properties have been preserved. In this chapter, we focus on these two important issues and demonstrate how CafeOBJ [FS95, FN97, DF98], a powerful algebraic specification language, can be used as an integrated tool with FTA to analyze, find and solve problems more efficiently and effectively. In addition, as a complement of theorem proving technique supported by OTS (observational transition system)/ CafeOBJ, we also demonstrate how to model-check OTSs with Maude [CELM96, CDE+02, CDE+03a, CDE+03b], a sibling language of CafeOBJ.

4.1 Formal Specification of FTA

In this Section, we discuss how to write the formal specification of the fault tree and automatically calculate minimal cut sets with Term Rewriting System (TRS) of CafeOBJ.

The reason why we choose CafeOBJ because it is an executable algebraic specification language which is a modern successor of OBJ [JWM+00, FGJM85], and incorporating several new algebraic specification paradigms. Its definition is given in [DF98]. CafeOBJ is intended to be primarily used for system specification, formal verification of specifications, rapid prototyping, programming, etc (those who are not familiar with CafeOBJ language and formal specification, can access http://www.ldl.jaist.ac.jp/cafeobj/, or read [FNT00] and [DF98] and Section 2.3 in this thesis for details).

Our formal fault tree specification consists of two parts. The first is the module (called TL here) that contains the definitions of logic operators, as well as useful axioms for term rewriting and system reasoning, such as axioms for reducing Disjunctive Normal Form (DNF) and absorption axioms for Minimal Cut Sets; the second part contains the speci-

fication of the fault trees. The former can be looked at as a built-in system module and it is transparent to the user after having been constructed once and for all; the latter is to help engineers write the formal specifications of the fault trees and reduce the minimal cut sets automatically.

As we introduced before, CafeOBJ is a wide spectrum specification language based on multiple logical foundations, and it provides an integrated, cohesive, and unified approach to programming/specification. By using CafeOBJ, we can define the logical operators as we need them; this property enables wide application to many application areas. For instance, in the collision control system example, we can define the TL module as follows (a comment starts with '--' or '-->' and terminates at the end of the line in CafeOBJ).

```
module TL {

  [ Formula ]

-- Primitive Boolean operators and logic gates of fault trees
  ops True False : -> Formula
  op !_   : Formula         -> Formula { strat: (0 1) prec: 53 }      -- Negation
  op _/\_ : Formula Formula -> Formula {assoc comm prec: 55 l-assoc } -- AND-gate
  op _\/_ : Formula Formula -> Formula {assoc comm prec: 59 l-assoc } -- OR-gate
  op _&_  : Formula Formula -> Formula {prec: 57}                     -- INHIBIT-gate

-- Temporal operator
  op ONCE_ : Formula -> Formula { strat: (0 1) prec: 53 } -- Once operator

  vars a b c : Formula

-- Basic axioms for logic deduction
  eq False /\ a = False .
  eq True /\ a = a .
  eq a /\ a = a .  -- also a absorption axiom for minimal cut sets
  eq a \/ a = a .  -- also a absorption axiom for minimal cut sets
  eq False \/ a = a .
  eq True \/ a = True .
  eq a \/ ! a = True .
  eq a /\ ! a = False .

-- axioms for Negative Normal Form
  eq ! True = False .
  eq ! False = True .
  eq ! ! a = a .
  eq ! (a \/ b) = ! a /\ ! b .
  eq ! (a /\ b) = ! a \/ ! b .

-- axioms for Disjunctive Normal Form (DNF)
  eq (a \/ b) /\ c = (a /\ c) \/ (b /\ c) .

-- absorption axioms for Minimal Cut Sets
  eq a \/ (a /\ b) = a .

-- absorb condition b of INHIBIT-gate in Minimal Cut Sets
  eq a & b = a .

-- absorb ONCE operator in the deduction of sub-events
```

```
  eq ONCE a = a .

}
```

As shown above, we can define the logic operators and axioms as we need, and the TL module is a extensible module with respect to different applications. Here, we only listed the definitions of AND-gate, OR-gate, and INHIBIT-gate, which are used in the fault tree of the crossing control system example.

There are three important axioms (equations) for the absorption of minimal cut sets, i.e.,

```
  eq a /\ a = a .
  eq a \/ a = a .
  eq a \/ (a /\ b) = a .
```

which are corresponding to the three absorption rules of Algorithm 2.1 we introduced in Section 2.1.3 (page 16), i.e.,

- If a cut set contains the same basic event more than once, then the redundant entries can be delete;
- If two cut sets are the same, then delete one;
- If one cut set is the subset of another, the latter can be removed since it is not a minimal cut set.

And since the resulting formula of all the minimal cut sets of a fault tree is a disjunctive normal form (DNF), in which each minimal cut set consists of the basic events of one conjunction, we also listed the axioms for the transformation of DNF as shown above. Notice here, we presented two other axioms to absorb the temporal operator $\diamondsuit$ (ONCE) and the condition of INHIBIT-gate at the end of TL module. We have demonstrated the soundness of the absorption of the ONCE operator in the fault tree construction model in the last chapter (given the temporal semantics to the corresponding logic gates), and the reason to absorb the conditions of INHIBIT-gates is that such conditions are usually normal state formulas and do not consist of the minimal cut sets as we discussed before.

After building the TL module according to the requirements of the practical application (necessary operators and axioms), we can write the formal specification of the fault tree as follows.

```
-- Formal Specification of fault tree and Calculation of Minimal Cut Sets

open TL
  [Object]
  ops tr cr ba sys : -> Object .
  ops OnCrossing Open Release Passed Idling          : Object -> Formula .
  ops BypassSignal BrakeFailure RadioFailure TimeOut :        -> Formula .

  op  Root : -> Formula .

-- formal specification of fault tree
```

```
  eq Root = OnCrossing(tr) /\ Open(ba) .

-- definition of domain rules
  eq [D1] : OnCrossing(tr) = ONCE (BypassSignal \/ BrakeFailure \/ Release(tr)) .
  eq [D2] : Release(tr) = Release(cr) /\ ! RadioFailure .
  eq [D3] : Open(ba) = TimeOut \/ Idling(sys) \/ Passed(cr) .

red Root .
```

As shown above, writing the formal specifications is quite *easy* and *straightforward* after the TL module has been built: first define the events and nodes of the fault tree, then use equations to record the domain rules used in event decomposition and fault tree construction (notice here to simplify the specification, we use `TimeOut` to represent the temporal formula $\boxminus_{\geq d} Closed(ba)$). The minimal cut sets can be automatically generated using a simple command '`red Root .`', and the result is shown as follows.

```
-- opening module TL.. done._*
-- reduce in %TL : Root
Release(cr) /\ ! RadioFailure /\ Passed(cr) \/
    BrakeFailure /\ Passed(cr) \/
    BypassSignal /\ Passed(cr) \/
    Idling(sys) /\ Release(cr) /\ ! RadioFailure \/
    TimeOut /\ Release(cr) /\ ! RadioFailure \/
    BrakeFailure /\ Idling(sys) \/
    BypassSignal /\ Idling(sys) \/
    BrakeFailure /\ TimeOut \/
    BypassSignal /\ TimeOut : Formula
(0.000 sec for parse, 13 rewrites(0.093 sec), 313 matches)
```

People may notice that the above result is different with Figure 3.6, that is, the minimal cut sets generated by the above fault tree specification are bigger than those in Figure 3.6. This is because in the above specification, we did not deal with the INHIBIT-gates, and all the conditional events were not eliminated but further decomposed and introduced into the minimal cut sets. For example, the event $BrakeFailure \wedge Open(ba)$ in Figure 3.6 are further decomposed into three sub-events (minimal cut sets) $BrakeFailure \wedge Passed(cr)$, $BrakeFailure \wedge Idling(sys)$, and $BrakeFailure \wedge TimeOut$, supported by the domain rule $\mathbf{D_3}$ for the decomposition of $Open(ba)$. Therefore, the result generated by the fault tree specification is more *complete* and *precise* but not inconsistent compared with Figure 3.6.

However, if people want to simplify the result and eliminate all the conditional event in the final minimal cut sets, it is also quite easy to achieve this goal by adding some equations (axioms) to explicitly record the INHIBIT gates and design decisions in the fault trees. As for this example, we need only introduce four equations as follows.

```
  eq BypassSignal /\ b = BypassSignal & b .
  eq BrakeFailure /\ b = BrakeFailure & b .
  eq TimeOut      /\ b = TimeOut      & b .

  eq a /\ ! RadioFailure = a & ! RadioFailure .
```

The first three equations can be called as design decisions, which state that either $BypassSignal$ or $BrakeFailure$ itself is a fault event regardless of any conditional event `b` (where `b` is a variable representing any event rather than a specific event name). Therefore, `b` can be represented as a conditional event connecting with an INHIBIT-gate that finally should be eliminated from the minimal cut sets, supported by the equation "`eq a & b`

= a .” in the TL module. The last equation states that !*RadioFailure* is normal event that should be regarded as a conditional event and need not be considered in the minimal cut sets.

The above four equations are all used to deal with the INHIBIT-gates, but their structures are different. This is depended on the property of conditional events. Here, a concept of primary conditional event should be introduced. A *primary conditional event* is a normal event that need not be further decomposed in the fault trees. Or more specifically, there is no domain rule used to decompose it. In the above example, the event *Open*(*ba*) is not a primary conditional event because it is further decomposed in the other branches of the fault tree, therefore, we can not just simply write it in either of the following two forms:

```
eq BrakeFailure /\ Open(ba) = BrakeFailure & Open(ba) .
eq a /\ Open(ba) = a & Open(ba) .
```

Because the former can not handle the other cases, such as `Brakefailure /\ Idling(sys)`, in case *Open*(*ba*) is rewritten (decomposed) before it; while the latter may delete some other branches of the fault tree if it is executed before the decomposition of *Open*(*ba*). This is because generally speaking, in the term rewriting system (TRS) of CafeOBJ, we can not ensure which equation will be used first for term rewriting.

Instead of involving in such troubles, we propose the following solutions to represent the INHIBIT-gate in the fault tree specification.

- If the conditional event connecting to an INHIBIT-gate is a primary conditional event, then it can be represented as:

  ```
  eq a /\ primary-conditional-event = a & primary-conditional-event .
  ```

  where `a` is a variable representing any fault event.

- Otherwise we should use a variable `b` to represent the conditional event, and specify the specific fault event of the INHIBIT-gate as follows:

  ```
  eq fault-event /\ b = fault-event & b .
  ```

After introducing the corresponding equations to handle the INHIBIT-gates, we can reduce the simplified minimal cut sets as follows.

```
-- reduce in %TL : Root
Release(cr) /\ Passed(cr) \/
    Idling(sys) /\ Release(cr) \/
    TimeOut \/
    BrakeFailure \/
    BypassSignal : Formula
(0.000 sec for parse, 28 rewrites(0.000 sec), 322 matches)
```

It should be figured out that in our fault tree specification, we did not follow the standard way, that is, just simply restating the logic structure and formulas of the fault tree, such as in the following form:

```
eq Root = S1a \/ S1b \/ S1c .
eq S1a  = BypassSignal /\ Open(ba) .
...
```

The above standard specification style strictly records the fault tree structure, and can be regarded as an *restatement* of the graphical fault tree. However, such kind of specification neglects a most important factor in our formal fault tree construction model, i.e., domain rules. It does not provide the ability for fault tree revising and rechecking based on the domain rules, even the minimal cut sets can also be automatically generated. And considering such kind of logic structure of the fault tree has already been interpreted well by the graphical tree, we recommend to record the domain rules rather than logic structure as for the formal specification of the fault trees – *it gives engineers and designers the ability to revise and recheck the fault tree more efficiently, and helps them focus on the most important factor of the fault tree, i.e., domain rules.*

It should also be noted that in this example, since the fault tree is not complex, the minimal cut sets can also be quickly calculated by hand using Algorithm 2.1. But with respect to complex system analysis, it is useful and effective. Moreover, as said above, such formal specification itself is an important formal documents for system revising and rechecking, which is the motivation of this study.

4.2 System Modeling, Specifying, and Verifying with OTS/CafeOBJ

As we discussed in the last chapter, after using FTA to find the important system safety related problems, another inevitable and crucial problem is how to further revise the system design accordingly and formally verify that such safety properties (safety commitment) have been preserved. In this section, we focus on this issue and demonstrate how CafeOBJ, can be used to formally model, specify, and verify a system as well as its important safety properties with OTS/CafeOBJ [1].

For better understanding, we first present a review and brief description of the observational transition system (OTS) in CafeOBJ, which has been introduced in detail in Section 2.3.

Observational transition systems, or OTSs are the definition of transition systems for writing transition systems in terms of equation [OF03]. We assume that there exists an universal state space called Υ. We also suppose that each data type used has been defined beforehand, including the equivalence between two values v_1, v_2 of the data type denoted by $v_1 = v_2$. An OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{O}$: A set of observers. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \rightarrow D$, where D is a data type and may be different for each observers. Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the equivalence between two states, denoted by $v_1 =_{\mathcal{S}} v_2$, with respect to $\mathcal{S}$ is defined as $v_1 =_{\mathcal{S}} v_2 \stackrel{def}{=} \forall o \in \mathcal{O}.o(v_1) = o(v_2)$.
- $\mathcal{I}$: A set of initial states such that $\mathcal{I} \subset \Upsilon$.
- $\mathcal{T}$: A set of conditional transition rules. Each $\tau \in \Sigma$ is a function $\tau : \Upsilon / =_{\mathcal{S}} \rightarrow \Upsilon / =_{\mathcal{S}}$ on equivalence classes of Υ with respect to $=_{\mathcal{S}}$. Let $\tau(v)$ be the representative

[1] In this section, we only list a part of important codes with respect to the crossing control system example. The complete CafeOBJ proof scores/codes are listed in Appendix A as for reference.

element of $\tau([v])$ for each $v \in \Upsilon$ and it is called the successor state of v with respect to τ. The condition c_τ for a transition rule $\tau \in \mathcal{T}$, which is a predicate on states, is called the effective condition. The effective condition is supposed to satisfy the following requirement: given a state $v \in \Upsilon$, if c_τ is false in v, namely τ is not effective in v, then $v =_{\mathcal{S}} \tau(v)$.

Multiple similar observers and transition rules may be indexed. Generally, observers and transition rules are denoted by $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, respectively, provided that $m, n \geq 0$ and we assume that there exist data type D_k such that $k \in D_k$ $(k = i_1, \ldots, i_m, j_1, \ldots, j_n)$.

CafeOBJ [FS95, FN97, DF98] is mainly based on two logical foundations: initial and hidden algebra [GM97]. Corresponding to the algebra, there are two kinds of sorts in CafeOBJ: visible and hidden. A visible sort represents an abstract data type, and a hidden sort represents the state space of an object. Two kinds of operations are used for hidden sorts: action and observation, corresponding to *transition rule* and *observer* of OTSs, respectively. An action can change the state of an object. While an observation can be used to observe the value of a data component in an object, it does not change the state of the object.

4.2.1 System Modeling

Based on the FTA results and the introduction of OTS in CafeOBJ, we modeled the radio-based crossing control system as an OTS. In this modeling, there are three discrete variables, t, l, and b, that show the states of the train, the level crossing, and the barriers, respectively.

The states of the train are *sect*0, *sect*1, *critical*, and *sect*2.

- *sect*0: the train is approaching, near enough to the 'latest braking point' to send the 'secure' signal to the level crossing.
- *sect*1: just after the train sends the 'secure' signal and until it gets permission (the 'release' signal) to enter its critical section from the level crossing (notice here for simplicity, we integrate the stop state into *sect*1, that is to say, the train will stop and stay in *sect*1 unless it gets a 'release' signal).
- *critical*: the train is passing over the level crossing.
- *sect*2: the train leaves the level crossing and sends a 'passed' signal to the level crossing.

The states of the level crossing are *state*0, *state*1, and *state*2.

- *state*0: the level crossing is waiting for a 'secure' signal from a train.
- *state*1: after getting the 'secure' signal and until it returns a 'release' signal to the train.
- *state*2: after the response to the train and until it gets the 'passed' signal from the train. After getting the 'passed' signal, the level crossing closes the barriers, and then enters idling state (*state*0) waiting for the next train.

The states of the barriers are only two: *open* and *close*.

According to the results of the fault tree analysis, we introduce a Boolean variable that is defined as follows:

$$\text{Boolean } idling = true$$

where '*idling*' is shared by all trains, initially set to *true*, denoting that the system is idling and the barriers are *open*.

There are three pairs of signals between the trains and the level crossing: 'secure', 'release', and 'passed'. Thus we can define six transition rules, based upon: 1) what each transition rule corresponds to in the scenario, 2) the condition under which each transition rule becomes effective, 3) the states of the train, the level crossing, the barriers, and/or the *idling* variable after each transition rule is executed if the transition rule is effective. The rules are as follows.

Initially, the states of the trains, the level crossing, the barriers, and *idling* are *sect*0, *state*0, *open*, and *true*, respectively.

- `tr-send-secure`

  1) The train sends a 'secure' signal to the level crossing.
  2) The state of the train is *sect*0.
  3) The state of the train becomes *sect*1.

- `le-get-secure`

  1) The level crossing gets a 'secure' signal from the train.
  2) The state of the train is *sect*1, and $idling = true$.
  3) – $idling = false$;
     – the state of the level crossing is *state*1;
     – the state of the barriers is *close*.

- `le-send-release`

  1) The level crossing sends the 'release' signal to the train.
  2) The state of the level crossing is *state*1.
  3) The state of the level crossing becomes *state*2.

- `tr-get-release`

  1) The train gets the 'release' signal from the level crossing and enters its critical section.
  2) The state of the level crossing is *state*2.
  3) The state of the train becomes *critical*.

- `tr-send-passed`

1) The train leaves the level crossing and sends the 'passed' signal to the level crossing.
2) The state of the train is *critical*.
3) The state of the train becomes *sect*2.

- `le-get-released`

1) The level crossing gets the 'passed' signal.
2) The state of the train is *sect*2.
3) – The state of the level crossing returns to *state*0;
   – The state of the train returns to *sect*0
   – $idling = true$

4.2.2 System Specification

In this section we describe how to specify the OTS for modeling the radio-based crossing control system in CafeOBJ.

The main part of the signature is shown below.

```
  *[System]*
  [Train]
  [TState LState BState]
-- any initial state
  op init : -> System
-- observations
  bop idling : System  -> Bool
  bop t : System Train -> TState
  bop l : System Train -> LState
  bop b : System       -> BState
-- actions (transition rules)
  bop tr-send-secure  : System Train -> System
  bop le-get-secure   : System Train -> System
  bop le-send-release : System Train -> System
  bop tr-get-release  : System Train -> System
  bop tr-send-passed  : System Train -> System
  bop le-get-passed   : System Train -> System
```

In the above signature, a comment starts with '`--`' and terminates at the end of the line. Declarations of visible sorts are enclosed with '`[`' and '`]`', and those of hidden sorts with '`*[`' and '`]*`'. Declarations of observations and actions starts with '`bop`', and those of other operations with '`op`'. Hidden sort `System` represents the state space of the OTS, and visible sorts `Train`, `TState`, `LState`, and `BState` represent the trains, the states of the trains, the states of the level crossing, and the states of the barriers, respectively. Operations with no arguments are called constants. Constant `init` denotes any initial state of the OTS.

We have basically seven sets of equations in the specification: one for any initial state, and the others for the six actions. The equations defining any initial state, along with the definitions of CafeOBJ variables used in this specification are as follows.

```
  var  S     : System
  vars T1 T2 : Train
  eq idling(init) = true .   -- variant init initially set true
  eq t(init, T1)  = sect0 .  -- any train initially set sect0
  eq l(init, T1)  = state0 . -- level crossing initially set state0
  eq b(init)      = open .   -- barriers initially set open
```

The equations which define the six transition rules are as follows.

```
-- (1) tr-send-secure
  ceq t(tr-send-secure(S, T1), T2) = (if (T1 = T2) then sect1 else t(S, T2) fi)
      if t(S, T1) = sect0 .
  eq  l(tr-send-secure(S, T1), T2) = (if (T1 = T2) then l(S, T1) else l(S, T2) fi) .
  eq  b(tr-send-secure(S, T1)) = b(S) .
  eq  idling(tr-send-secure(S, T1)) = idling(S) .
  ceq tr-send-secure(S, T1) = S if not (t(S, T1) = sect0) .

-- (2) le-get-secure
  ceq idling(le-get-secure(S, T1)) = false if t(S, T1) = sect1 and idling(S) .
  ceq b(le-get-secure(S, T1)) = close if t(S, T1) = sect1 and idling(S) .
  ceq l(le-get-secure(S, T1), T2) = (if (T1 = T2) then state1 else l(S, T2) fi)
      if t(S, T1) = sect1 and idling(S) .
  eq  t(le-get-secure(S, T1), T2) = (if (T1 = T2) then t(S, T1) else t(S, T2) fi) .
  ceq le-get-secure(S, T1) = S if not (t(S, T1) = sect1) or not idling(S) .

-- (3) le-send-release
  ceq l(le-send-release(S, T1), T2) = (if (T1 = T2) then state2 else l(S, T2) fi)
      if l(S, T1) = state1 .
  eq  b(le-send-release(S, T1)) = b(S) .
  eq  t(le-send-release(S, T1), T2) = (if (T1 = T2) then t(S, T1) else t(S, T2) fi) .
  eq  idling(le-send-release(S, T1)) = idling(S) .
  ceq le-send-release(S, T1) = S if not (l(S, T1) = state1) .

-- (4) tr-get-release
  ceq t(tr-get-release(S, T1), T2) = (if (T1 = T2) then critical else t(S, T2) fi)
      if (l(S, T1) = state2) .
  eq  l(tr-get-release(S, T1), T2) = (if (T1 = T2) then l(S, T1) else l(S, T2) fi) .
  eq  b(tr-get-release(S, T1)) = b(S) .
  eq  idling(tr-get-release(S, T1)) = idling(S) .
  ceq tr-get-release(S, T1) = S if not (l(S, T1) = state2) .

-- (5) tr-send-passed
  ceq t(tr-send-passed(S, T1), T2) = (if (T1 = T2) then sect2 else t(S, T2) fi)
      if t(S, T1) = critical .
  eq  l(tr-send-passed(S, T1), T2) = (if (T1 = T2) then l(S, T1) else l(S, T2) fi) .
  eq  b(tr-send-passed(S, T1)) = b(S) .
  eq  idling(tr-send-passed(S, T1)) = idling(S) .
  ceq tr-send-passed(S, T1) = S if not (t(S, T1) = critical) .

-- (6) le-get-passed
  ceq b(le-get-passed(S, T1)) = open if t(S, T1) = sect2 .
  ceq idling(le-get-passed(S, T1)) = true if t(S, T1) = sect2 .
  ceq l(le-get-passed(S, T1), T2) = (if (T1 = T2) then state0 else l(S, T2) fi)
      if t(S, T1) = sect2 .
  ceq t(le-get-passed(S, T1), T2) = (if (T1 = T2) then sect0 else t(S, T2) fi)
      if t(S, T1) = sect2 .
  ceq le-get-passed(S, T1) = S if not (t(S, T1) = sect2) .
```

4.2.3 Verifying Safety Properties

Reviewing the two basic fault events S_{3b} and S_{3c} found in the last chapter, we can derive two safety claims as follows.

Claim 100 : if the level crossing has sent a 'release' signal, then the state of the system should not be idling. If we focus on the state after the corresponding transition rule `le-send-release` is executed, i.e. `l(S, T1) = state2`, the invariant we want to prove is:

```
inv100:  (l(S, T1) = state2) implies not (idling(S))
```

Claim 200 : if the level crossing has sent a 'release' signal, then the system can not get a 'passed' signal at this moment. Here, if we focus on the states after the transition rules `le-send-release` and `le-get-passed` have been executed, i.e. `l(S, T1) = state1` and `l(S, T2) = state0`, we can derive the invariant as follows:

```
inv200:  (l(S, T1) = state2) implies not (l(S, T2) = state0)
```

Moreover, if we consider another consequence of the transition rule `le-get-release`, i.e. `idling(S) = true`, then inv200 is equal to inv100 in this sense.

The general proof structure of the invariants in CafeOBJ is as follows: An invariant is proved by INDUCTION on each transition rule applied or executed. If we want to prove that the system has an invariant P, the proof structure looks like this:

I) Base case

P(init)

II) Inductive cases

(1) P(s) implies P(tr-send-secure(s)) for any s.
(2) P(s) implies P(le-get-secure(s)) for any s.
(3) P(s) implies P(le-send-release(s)) for any s.
(4) P(s) implies P(tr-get-release(s)) for any s.
(5) P(s) implies P(tr-send-passed(s)) for any s.
(6) P(s) implies P(le-get-passed(s)) for any s.

In other words, first we prove the invariant P is true in any initial state, and then try to prove that it will be preserved by any transition rule. To this end, we should do case splitting based on the effective conditions of each transition rule. A part of the template module for case splitting is presented as follows.

```
-- This module is to specify the s and s' for induction based on the
-- effective conditions of each transition rule.

mod ICASE1 {
  pr(ISTEP) -- ISTEP is the module defining the invariants to prove
```

```
-- arbitrary trains
  op t10 : -> Train
-- assumptions for transition rule: tr-send-secure
  eq t(s, t10) = sect0 .
-- successor state
  eq s' = tr-send-secure(s, t10) .
}

mod nonICASE1 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect0) = false .
-- successor state
  eq s' = tr-send-secure(s, t10) .
}

mod ICASE2 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions for transition rule: le-get-secure
  eq t(s, t10) = sect1 .
  eq idling(s) = true .
-- successor state
  eq s' = le-get-secure(s, t10) .
}

mod nonICASE2-1 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect1) = false .
  eq idling(s) = true .
-- successor state
  eq s' = le-get-secure(s, t10) .
}

mod nonICASE2-2 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect1) = false .
  eq idling(s) = false .
-- successor state
  eq s' = le-get-secure(s, t10) .
}

mod nonICASE2-3 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect1) = true .
```

```
  eq idling(s) = false .
-- successor state
  eq s' = le-get-secure(s, t10) .
}

......
```

An important issue is to find lemmas for the invariant to prove in some inductive cases (transition rules). For example, when we try to prove inv100, we find that it does not hold in cases (3) and (6) above, thus we need to introduce two lemmas as follows:

```
-- lemma for inductive case (3)
eq inv310(S, T1) = (l(S, T1) = state1) implies not idling(S) .
-- lemma for inductive case (6)
eq inv320(S, T1, T2) = (t(S, T1) = sect2) and (l(S, T2) = state2) implies (T1 = T2) .
```

After introducing the lemmas inv310 and inv320, we need to prove the inductive cases (3) and (6) with respect to the proof of inv100 again as follows:

inv310 implies (P(s) implies P(le-send-release(s)) for any s.)

inv320 implies (P(s) implies P(le-get-passed(s)) for any s.)

And we have proved that these two revised inductive cases hold in the OTS.

Next, we have to prove inv310 and inv320 in the same way. Finally we will have found and proved all the invariants listed below.

```
  eq inv100(S, T1) = (l(S, T1) = state2) implies not idling(S) .
     -- need lemmas inv310 and inv320
  eq inv310(S, T1) = (l(S, T1) = state1) implies not idling(S) .
     -- need lemma inv330
  eq inv320(S, T1, T2) = (t(S, T1) = sect2) and (l(S, T2) = state2)
     implies (T1 = T2) .
     -- need lemmas inv340 and inv330
  eq inv330(S, T1, T2) = (t(S, T1) = sect2) implies not (l(S, T2) = state1) .
     -- need lemmas inv360 and inv420
  eq inv340(S, T1, T2) = (l(S, T1) = state2) and (t(S, T2) = critical)
     implies (T1 = T2) .
     -- need lemmas inv350 and inv360
  eq inv350(S, T1, T2) = (l(S, T1) = state2) and (l(S, T2) = state2)
     implies (T1 = T2) .
     -- need lemma inv370
  eq inv360(S, T1, T2) = (l(S, T1) = state1) implies not (t(S, T2) = critical) .
     -- need lemmas inv370 and inv380
  eq inv370(S, T1, T2) = (l(S, T1) = state1) implies not (l(S, T2) = state2) .
     -- need lemmas inv390 and inv100
  eq inv380(S, T1) = (t(S, T1) = critical) implies not idling(S) .
     -- need lemmas inv100 and inv400
  eq inv390(S, T1, T2) = (l(S, T1) = state1) and (l(S, T2) = state1)
     implies (T1 = T2) .
     -- need lemma inv310
  eq inv400(S, T1, T2) = (t(S, T1) = sect2) implies not (t(S, T2) = critical) .
     -- need lemmas inv320 and inv410
  eq inv410(S, T1, T2) = (t(S, T1) = critical) and (t(S, T2) = critical)
     implies (T1 = T2) .
```

```
    -- need lemma inv340
  eq inv420(S, T1) = (t(S, T1) = sect2) implies not idling(S) .
    -- need lemmas inv380 and inv430
  eq inv430(S, T1, T2) = (t(S, T1) = sect2) and (t(S, T2) = sect2)
    implies (T1 = T2) .
    -- need lemma inv400
```

As shown above, we have formally demonstrated that the safety properties (safety commitment) found in our formal fault tree model are preserved in the OTS; in other words, the basic fault events S_{3b} and S_{3c} will never occur in our refined system or OTS (we use the same invariant, inv100, to represent them as discussed above). Moreover, a typical and stronger safety property of this mutual exclusion algorithm has been proved, i.e., inv410, which states that *more than one train can never enter the same critical section simultaneously.*

4.3 Model-Checking OTSs with Maude

In the last section, we have been successfully applying the OTS/CafeOBJ method to modeling, specification, verification of the crossing control system based on the analyzing results of FTA. In the OTS/CafeOBJ method, systems are modeled as observational transition systems, or OTSs, which are the definition of transition systems for writing them in terms of equations, and OTSs are written in CafeOBJ. We then verify that the OTSs have properties (safety commitment found by FTA) by writing proof scores in CafeOBJ and executing them the CafeOBJ system (see Appendix A).

In the OTS/CafeOBJ method, theorem proving technique is mainly used to verify that OTSs have properties. There is also model-checking technique [CGP01] to verify that (usually finite) distributed systems have properties. Although theorem proving technique would not be totally replaced with model-checking technique, model-checking technique is generally easier to use and can be used as the complement of theorem proving technique. To this end, in the section, we introduce how to describe and write OTSs in Maude [CELM96, CDE+02, CDE+03a, CDE+03b] (a sibling language of CafeOBJ), and how to model-check them with the Maude LTL model-checker [EMS02]. The crossing control system example is used to demonstrate the method.

4.3.1 Description of OTSs in Maude

Maude [CELM96, CDE+02, CDE+03a, CDE+03b] is a specification and programming language based on rewriting logic [Mes92]. Maude is equipped with modern specification and programming language features such as fast (AC-) rewriting and meta programming. Maude also has model-checking facilities [EMS02]. The Maude 2.0 distribution includes the file `model-checker.maude` in which modules related to model-checking are declared.

Based on the definition of OTS introduced in Section 4.2, we describe how to write an OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ in Maude. $\mathcal{O}$ and $\mathcal{T}$ are denoted by sorts, say `OValue` and `TRule`, respectively. The state space Υ is denoted by a sort, say `Sys`. A snapshot of $\mathcal{S}$ is represented by a multiset, or a bag of observers and transition rules. `OValue` and `TRule` are then declared as subsorts of `Sys` as follows.

```
subsort OValue TRule < Sys .
```

The following operators are used as the constructors of bags:

```
op none : -> Sys .
op __   : Sys Sys -> Sys [assoc comm id: none] .
```

Generally, a snapshot of $\mathcal{S}$ is as the following form:

$$\textit{ovalue-1} \ldots \textit{ovalue-M}\ \ \textit{trule-1} \ldots \textit{trule-N}$$

where *ovalue-i*$(i = 1, \ldots, M)$ is a term denoting an observer, and *trule-i*$(i = 1, \ldots, N)$ is a term denoting a transition rule.

An observer $o_{i_1,\ldots,i_m} \in \mathcal{O}$ is denoted by an operator. We assume that the date types $D_k(k = i_1, \ldots, i_m)$ and D are defined and there exist sorts $V_k(k = i_1, \ldots, i_m)$ and V corresponding to the data types. The operator denoting $o_{i_1,\ldots,i_m}$ is declared as follows:

$$\texttt{op}\ (o\texttt{[\_,...,\_] =\_})\ :\ \ V_{i_1}\ \ldots V_{i_m}\ V\ \texttt{-> OValue .}$$

An transition rule $\tau_{j_1,\ldots,j_n} \in \mathcal{T}$ is denoted by an operator. We assume that the date types $D_k(k = j_1, \ldots, j_n)$ and D are defined and there exist sorts $V_k(k = j_1, \ldots, j_n)$ and V corresponding to the data types. The operator denoting $\tau_{j_1,\ldots,j_n}$ is declared as follows:

$$\texttt{op}\ r\ :\ \ V_{j_1}\ \ldots V_{j_n}\ \texttt{-> TRule .}$$

Transition rules are defined using Maude rules. We suppose that observers needed and affected by the execution of the transition rule $\tau_{j_1,\ldots,j_n}$ are $o^1_{i^1_1,\ldots,i^1_{m_1}}, \ldots, o^l_{i^l_1,\ldots,i^l_{m_l}}$, which are supposed to be denoted by operators $(o^1\texttt{[\_,...,\_] =\_}), \ldots, (o^l\texttt{[\_,...,\_] =\_})$. Then, the transition rule $\tau_{j_1,\ldots,j_n}$ denoted by r is generally defined as follows:

$$\begin{array}{l}
\texttt{crl [rule-}r\texttt{] :} \\
\quad r(X_{j_1}, \ldots, X_{j_n}) \\
\quad (o^1[X_{i^1_1}, \ldots, X_{i^1_{m_1}}] = X_1) \ldots (o^l[X_{i^1_l}, \ldots, X_{i^1_{m_l}}] = X_l) \\
\quad \texttt{=>} \\
\quad r(X_{j_1}, \ldots, X_{j_n}) \\
\quad (o^1[X_{i^1_1}, \ldots, X_{i^1_{m_1}}] = X'_1) \ldots (o^l[X_{i^1_l}, \ldots, X_{i^1_{m_l}}] = X'_l) \\
\quad \texttt{if}\ \ c\text{-}r(X_{j_1}, \ldots, X_{j_n}, X_{i^1_1}, \ldots, X_{i^1_{m_1}}, X_1, \ldots, X_{i^l_1}, \ldots, X_{i^l_{m_l}}, X_l)\ \ .
\end{array}$$

where c-r is the operator denoting $c_{\tau_{j_1,\ldots,j_n}}$. `rule-`$r$ is the label of the rule, which is optional. $X_k(k = j_1, \ldots, j_n, i^1_1, \ldots, i^1_{m_1}, 1, \ldots, i^l_1, \ldots, i^l_{m_l}, l)$ is a term or a variable for the intended sort. $X'_k(k = 1, \ldots, l)$ denotes the value returned by observer $o^k_{i^k_1,\ldots,i^k_{m_k}}$ in the successor state with respect to $\tau_{j_1,\ldots,j_n}$.

4.3.2 Model-Checking OTSs

We describe how to model-check that an OTS has properties with Maude. We suppose that the OTS is written in Maude as a module whose name is `SYSTEM`. We first define state predicates with which such properties are described. Such state predicates are declared in a module, say `SYSTEM-PREDS`, which looks like this

```
mod SYSTEM-PREDS is
  pr SYSTEM .
  inc SATISFACTION .
  subsort Sys < State .
  ...
endm
```

where the dots $\cdots$ indicate the part in which the syntax and semantics of state predicates are specified.

In the module `SATISFACTION` (included in the file `model-checker.maude`), the module `LTL` (included in the file `model-checker.maude`) where the propositional linear temporal logic (`LTL`) is described is imported, the sort `State` that denotes states of a system under consideration is declared and the following operator is declared:

```
op _|=_ : State Formula ~> Bool .
```

The sort `Formula` is declared in the module `LTL`, denoting propositional LTL formulas. The operator is used to define state predicates. That a state predicate denoted by a term *pred* holds in a state denoted by state is defined as follows:

`eq state |=` *pred* `= true .`

Generally, state is the following form:

ovalue-1 $\ldots$ *ovalue-M* S

where *ovalue-i* $(i = 1, \ldots, M)$ is a term for `OValue` and S is a variable for `Sys`.

We next define propositional LTL formulas denoting properties to be checked for the OTS and also initial states of the OTS. Such formulas and initial states are described in a module, say `SYSTEM-CHECK`, which looks like:

```
mod SYSTEM-CHECK is
    inc SYSTEM-PREDS    .
    inc MODEL-CHECKER   .
    inc LTL-SIMPLIFIER .
    ...
endm
```

where the dots $\cdots$ indicate the part in which operators denoting propositional LTL formulas to be checked for the OTS and initial states of the OTS, and the corresponding equations are declared.

In the module `MODEL-CHECKER` (included in the file `model-checker.maude`), the operator `modelCheck` is declared, which takes two arguments denoting an initial state and a propositional LTL formula, and returns the result of the model-checking. In the module `LTL-SIMPLIFIER` (included in the file `model-checker.maude`), operators and equations to simplify propositional LTL formulas are declared. It is optional to import `LTL-SIMPLIFIER`.

Propositional LTL formulas are constructed of state predicates declared in `SYSTEM-PREDS`, and Boolean connectives and temporal operators declared in `LTL`. Among temporal operators are Eventually ($\Diamond$) denoted by `<>_` , Henceforth ($\Box$) denoted by `[]_` and Leads-to

($\Rightarrow$) denoted by `_|->_` .

The term denoting an initial state is generally the following form:

$$\textit{ovalue-1} \ \ldots \ \textit{ovalue-}M \ \textit{trule-1} \ \ldots \ \textit{trule-}N$$

where *ovalue-i* ($i = 1, \ldots, M$) is a term for `OValue`, and *trule-i* ($i = 1, \ldots, N$) is a term for `TRule`.

Let *init* be a term denoting an initial state and prop be a term denoting a propositional LTL formula to be checked. We model-check that all the states reachable from the initial state satisfies the propositional LTL formula as follows:

$$\texttt{red modelCheck}(\textit{init}; \textit{prop}) \ .$$

4.3.3 Case Study: Crossing Control System

Based on the above analysis and the system formal modeling introduced in Section 4.2.1, we can model the crossing control system with OTS/Maude as follows.

- Observers
  - tr_i ($i \in Tid$) returns the state label of a train i, where Tid is the set of train IDs. Each tr_i initially returns label `sect0`.
  - le_i ($i \in Tid$) returns the state label of the level crossing with respect to a train i, where Tid is the set of train IDs. Each le_i initially returns label `state0`.
  - *ba* returns the state label of barriers, it initially returns label `open`.
  - *idling* returns the system Boolean value shared by all trains. It initially returns `true`.
- Transition rules
  - *tr-send-secure*$_i$ ($i \in Tid$) denotes that train i sends a 'secure' signal to the level crossing.
  - *le-get-secure*$_i$ ($i \in Tid$) denotes that level crossing gets a 'secure' signal from a train i.
  - *le-send-release*$_i$ ($i \in Tid$) denotes that level crossing sends a 'release' signal to a train i.
  - *tr-get-release*$_i$ ($i \in Tid$) denotes that train i gets a 'release' signal from the level crossing.
  - *tr-send-passed*$_i$ ($i \in Tid$) denotes that train i sends a 'passed' signal to the level crossing .
  - *le-get-passed*$_i$ ($i \in Tid$) denotes that level crossing gets a 'passed' signal from a train i.

The operators denoting the observers and transition rules are declared as follows:

```
*** Observable values
op tr[_] =_   : Train TState -> OValue .
op le[_] =_   : Train LState -> OValue .
op ba =_      : BState       -> OValue .
op idling =_  : Bool         -> OValue .

*** Transition rules
op tr-send-secure  : Train -> TRule .
op le-get-secure   : Train -> TRule .
op le-send-release : Train -> TRule .
op tr-get-release  : Train -> TRule .
op tr-send-passed  : Train -> TRule .
op le-get-passed   : Train -> TRule .
```

A comment starts with `***` and terminates at the end of the line. `Train`, `TState`, `LState`, `BState`, and `Bool` are the sorts denoting train IDs, state label of train, level crossing, barriers, and system idling state, respectively.

In the following, let `T`, `TS`, `LS`, `BS`, and `IS` be Maude variables for `Train`, `TState`, `LState`, `BState`, and `Bool`, respectively. Operator (transition rule) `tr-send-secure` is then defined with these equations:

```
crl [tr-send-secure] :
  tr-send-secure(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
  => tr-send-secure(T) (tr[T] = sect1) (le[T] = LS) (ba = BS) (idling = IS)
  if TS == sect0 .
```

And we can define the other operators in a similar way as follows.

```
crl [le-get-secure] :
  le-get-secure(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
  => le-get-secure(T) (tr[T] = TS) (le[T] = state1) (ba = close) (idling = false)
  if TS == sect1 and IS == true .

crl [le-send-release] :
  le-send-release(T) (tr[T] = TS) (le[T] = LS ) (ba = BS) (idling = IS)
  => le-send-release(T) (tr[T] = TS) (le[T] = state2) (ba = BS) (idling = IS)
  if LS == state1 .

crl [tr-get-release] :
  tr-get-release(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
  => tr-get-release(T) (tr[T] = critical) (le[T] = LS) (ba = BS) (idling = IS)
  if LS == state2 .

crl [tr-send-passed] :
  tr-send-passed(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
  => tr-send-passed(T) (tr[T] = sect2) (le[T] = LS) (ba = BS) (idling = IS)
  if TS == critical .

crl [le-get-passed] :
  le-get-passed(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
  => le-get-passed(T) (tr[T] = sect0) (le[T] = state0) (ba = open) (idling = true)
  if TS == sect2 .
```

Then let the OTS be finite by making the number of trains a fixed number, say three. We model-check that the finite OTS preserves the safety commitment found in FTA, i.e.,

$\Box\ \neg(Release(cr) \wedge Idling(sys))$.

Suppose `TC` is the module in which the OTS is written. The module `TC-PREDS` is declared as follows:

```
mod TC-PREDS is
  pr TC .
  inc SATISFACTION .
  subsort Sys < State .

  op cr-release    : Train -> Prop .
  op sys-idle      :       -> Prop .

  var T : Train .
  var S : Sys .

  eq (le[T] = state2)   S |= cr-release(T) = true .
  eq (idling = true)    S |= sys-idle = true .
endm
```

The module `TC-CHECK` is declared as follows:

```
mod TC-CHECK is
  inc TC-PREDS .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .

  ops t1 t2 t3 : -> Train .
  op init : -> Sys .
  op basic : -> Formula .

  eq init =
       tr-send-secure(t1)  tr-send-secure(t2)  tr-send-secure(t3)
       le-get-secure(t1)   le-get-secure(t2)   le-get-secure(t3)
       le-send-release(t1) le-send-release(t2) le-send-release(t3)
       tr-get-release(t1)  tr-get-release(t2)  tr-get-release(t3)
       tr-send-passed(t1)  tr-send-passed(t2)  tr-send-passed(t3)
       le-get-passed(t1)   le-get-passed(t2)   le-get-passed(t3)
       (tr[t1] = sect0) (tr[t2] = sect0) (tr[t3] = sect0)
       (le[t1] = state0) (le[t2] = state0) (le[t3] = state0)
       (ba = open)
       (idling = true) .

  eq basic = ([] ~(cr-release(t1) /\ sys-idle)) /\
             ([] ~(cr-release(t2) /\ sys-idle)) /\
             ([] ~(cr-release(t3) /\ sys-idle)) .
endm
```

The operator `basic` denotes the safety commitment (corresponding to INV100 in the proof score of OTS/CafeOBJ). We model-check that the finite OTS has the property as follows:

```
red modelCheck(init, basic) .
```

The result is true, which means that the safety commitment is preserved in this finite OTS. Moreover, we can also prove that the negation of the root of the fault tree ($\Box\ \neg(OnCrossing(tr) \wedge Open(ba))$) and the mutual exclusion problem (no more than one train can enter the crossing simultaneously) by introducing the following operators and equations:

```
op tr-oncrossing : Train -> Prop .
op ba-open       :       -> Prop .
eq (tr[T] = critical) S |= tr-oncrossing(T) = true .
eq (ba = open)        S |= ba-open = true .

ops root mutex : -> Formula .
eq root =  ([] ~(tr-oncrossing(t1) /\ ba-open)) /\
           ([] ~(tr-oncrossing(t2) /\ ba-open)) /\
           ([] ~(tr-oncrossing(t3) /\ ba-open)) .
eq mutex = ([] ~(tr-oncrossing(t1) /\ tr-oncrossing(t2))) /\
           ([] ~(tr-oncrossing(t2) /\ tr-oncrossing(t3))) /\
           ([] ~(tr-oncrossing(t1) /\ tr-oncrossing(t3))) .

red modelCheck(init, root) .
red modelCheck(init, mutex) .
```

The results are both true, which means that the finite OTS surely has the two properties (the complete codes and experimental results are listed in Appendix B).

Another benefit of using model-checking technique (`model-checker.maude`) is that it gives the ability to check some other properties in addition to safety properties (invariants) found in FTA. For instance, we can quickly check the *liveness* property (lockout free, i.e., once a train sends a 'secure' signal, it will eventually get the permission and pass on the crossing) of the above OTS by introducing the following operators and equations:

```
op wait : Train -> Prop .
eq (tr[T] = sect1)    S |= wait(T) = true .

op lockfree : -> Formula .
eq lockfree = (wait(t1) |-> tr-oncrossing(t1)) /\
              (wait(t2) |-> tr-oncrossing(t2)) /\
              (wait(t3) |-> tr-oncrossing(t3)) .
red modelCheck(init, lockfree) .
```

Unfortunately, the result is *not* true and we get a counterexample. Further analyzing the counterexample, we found the problem is caused by the competition of the shared system variable `idling`, or more specifically, a train may occupy the variable `idling` and pass the crossing repeatedly because in the OTS specification we simply reset its state to the initial state `sect0` after the crossing receives the 'passed' signal from the train. This special case may not happen in practice, because it is impossible or *unrealistic* to assume that a train can approach the crossing again immediately just after one pass. However, it explores a potential design defect that should be amended with regard to more reliable system design and specification. Some classical algorithms for mutual exclusion problem [Pet81, And90]can be introduced as for solution, such as using a queue or array to manage and schedule the requests of different trains instead of a simple Boolean variable. And an introduction of how to specify and verify some classical mutual exclusion algorithms with CafeOBJ can be found in [OF99].

4.4 Combination of FTA and OTS/CafeOBJ(Maude)

Based on the discussion of formal fault tree construction model (Chapter 3) and formal system specification and verification with OTS/CafeOBJ(Maude), we further present

detailed analysis about the motivation and advantages of the *combination* of these two techniques in this section.

First, let's review the role of traditional FTA and formal system specification and verification in requirements engineering, i.e., what can they do from each other's point of view.

- FTA
  - An system safety analysis technique, which starts from undesired hazards to analyze, and try to find sufficiently large part of possibilities that will result in those significant system failure modes.
  - Qualitative analysis of FTA can help us figure out minimal cut sets and classify the basic fault event into different categories, such as physical failures, human errors, or system design problems. Such kind of analysis is very important to assist us to understand the system more comprehensively and completely, and therefore take appropriate precautions.
  - Quantitative analysis of FTA can help us to evaluate the system reliability, given the failure rate of each basic event. And in reverse, it can also guide us to allocate reliability and system development resources to each sub-system or module at the system design stage [XFH03].
- Formal System Specification and Verification
  - Based on the primary requirements from domain experts and engineers, try to formalize and refine these requirements more formally and precisely. The generated formal system specification constitutes the foundation of subsequent software development and implementation.
  - Formal verification can help us formally prove that some important system properties are preserved in the formal specification.

Most importantly, these two techniques analyze the system and requirements from two *different standpoints*: FTA enumerates various ways for system failure from the concept of *failure space*, while formal system specification and verification depicts and proves various ways for system success from the concept of *success space*. Both of them complement each other efficiently and make the whole requirements analyses more comprehensive.

However, there are some *gaps* between these two methods. First of all, traditional FTA was developed from hardware system analysis, in which generally speaking, a system fault event can be directly attributed to some components failures. It is difficult to analyze complex system state that depends on event combinations rather than component state itself as we pointed out in the last chapter. Moreover, it lacks a formal construction model to ensure the correctness of the fault trees, therefore, the results of FTA are doubtful and can not be used for the formal specification and verification directly. To this end, we develop our formal fault tree construction model based on temporal logic, in which the correctness of the fault trees is guaranteed by the construction processes. Furthermore, we also discuss how to derive the safety assumption and commitment from the fault trees, which can be used to guild the subsequent formal system specification and verification

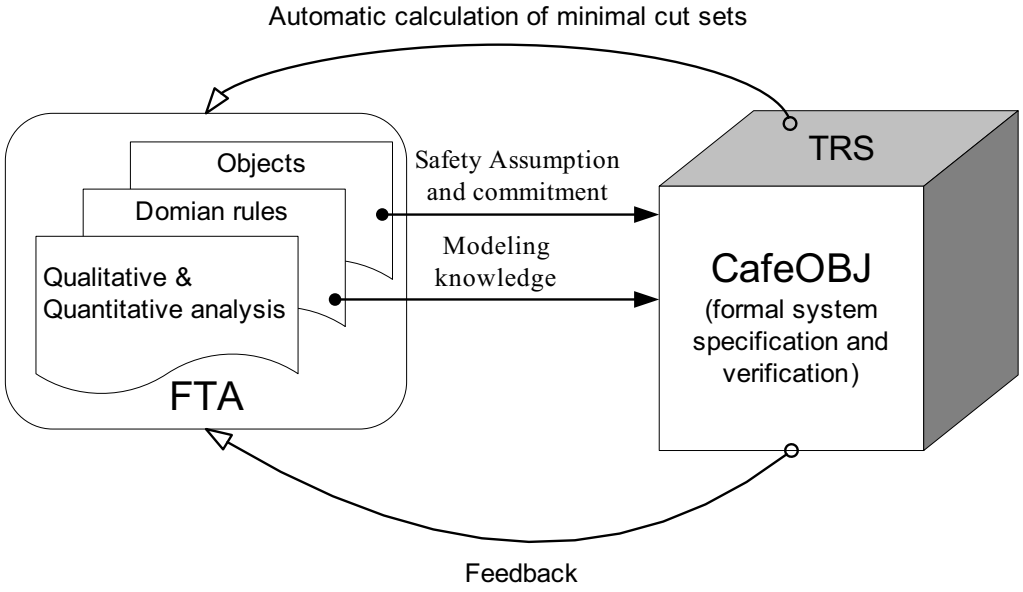


Figure 4.1: An overview of FTA and OTS/CafeOBJ

with OTS/CafeOBJ(Maude). An overview of the combination of our formal FTA model and OTS/CafeOBJ(Maude) is shown in Figure 4.1.

As shown in Figure 4.1, there are several benefits of the combination of our formal FTA and OTS/CafeOBJ(Maude) as for more efficient requirements analyses as follows.

- Firstly, from the standpoint of FTA, it can provide the following useful information to assist the subsequent system formal specification and verification with OTS/CafeOBJ(Maude).

  - The correctness of fault trees is guaranteed by the construction processes, thus a stand-alone verification of the fault tree itself is unneeded, which can help analyzers focus their attention on the problem per se.
  - The safety assumptions derived from FTA can help the analyzers understand the system more comprehensively, i.e., the system will only perform successfully under what kind of restrictions and assumptions. As we discussed in the last chapter, such limitation conditions are usually and easily overlooked if we carry out system design and specification directly without FTA or other hazard analysis techniques. In addition, it is also useful for taking proper precautions with respect to the corresponding anticipated failures in a timely fashion, such as making some regulations to prevent some potential human misoperations.
  - FTA also provides more precise and manageable *statements* (safety commitments) with respect to formal system verification with OTS/CafeOBJ(Maude). In complex system analysis, such kind of a divide-and-conquer method is especially a benefit in case the top event is too complex and difficult to be verified.
  - The domain rules discovered and defined during the fault tree construction can be reused to guide the subsequent formal system modeling and specifying. In other words, these domain rules can also be regarded as important system

design knowledge and enriched transition rules for formal system specification with OTS/CafeOBJ(Maude).

- Secondly, from the point of view of OTS/CafeOBJ(Maude), it can further formally specify and verify the system will work *successfully* based on the analyzing results of FTA.
  - On one hand, the formal verification can provide a *feedback* to FTA, that is, after the formal verification of the derived safety commitment, the corresponding basic fault events could be neglected (or marked with special symbols) in the qualitative and quantitative analysis of FTA. Thus, it can help us have a more definite object in view when allocating development and testing resources.
  - On the other, with the support of term rewriting system (TRS) of CafeOBJ, we can write the formal specification of the fault trees and reduce the minimal cut sets automatically, which can be regarded as another advantage of the combination of FTA and CafeOBJ.

It should be noted that in our experience, we recommend to do FTA before formal system specification and verification, this is based on the following facts in addition to the above discussion and analysis. First, it is generally easier to attain concurrence on what constitutes failure than it is to agree on what constitutes success. And another point is that from a practical standpoint there are generally more ways to success than there are failure, this is the advantage for the analyst to work in failure space as opposed to success space [VGRH81]. A good and typical example is the Minuteman missile analysis, only three fault trees were drawn and it was found that careful analysis of just these three fault trees covered all the significant failure modes [VGRH81]. And in our example, we have also demonstrated how FTA can assist the system design and requirements analyses as discussed in Section 3.3 — From safety analysis to system design.

4.5 Summary

In this chapter we have demonstrated how to formally model, specify, and verify the safety properties found in FTA with OTS/CafeOBJ. And as an alternative of theorem proving technique of CafeOBJ, we also discussed how to model checking OTSs with Maude, a sibling formal language of CafeOBJ.

This chapter can be regarded as the extension of Chapter 3, an overview of the combination of FTA and OTS/CafeOBJ is shown in Figure 4.1. The advantages of the combination can be concluded briefly as follows (the detailed discussion and analysis can be found in Section 4.4): On one hand, our formal fault tree analysis provides useful and instructive information and guidelines for the succeeding formal specification and verification of the system. On the other, in reverse, supported by OTS/CafeOBJ, we can formally verify and prove that some basic fault events of the fault tree will never occur in the refined system, which enables us to focus our attention on other key fault events or components, having a definite object in view when allocating development and testing resources.

We realize that for engineers, there may be a gap between FTA and the formal specification and verification process, if they are not familiar with formal methods. To solve this problem and make our method more practical, we currently are trying to develop some templates and guidelines which can help designers use formal methods more smoothly and efficiently. This is also the motivation that we study how to model-check OTSs with Muade.

In addition, since our formal specification and verification is mainly based OTS/CafeOBJ, but our formal fault tree construction model is based on temporal logic deduction, it causes some inconveniences in case experts and designers are not familiar with both techniques. And actually, the incorrectness problem in traditional FTA mainly originates from the difficulty of representing state transition and decomposing complex system state which consists of event combinations rather than component state itself. This brings about an important issue, that is, can we develop a formal fault tree model based on the framework of OTS instead of temporal logic?

There are two benefits of this idea. On one hand, it relieves the engineers of advanced temporal logic background, and helps them develop the fault tree in a comparatively easy way with basic concepts of OTS. On the other hand, more importantly, it makes the combination of FTA and OTS/CafeOBJ more *consistent*, since both of them are based on the common framework of OTS. Thus the communication between the engineers and designers as well as the transformation between these two processes is more efficient and smooth. This is exactly the motivation and goal of our another study discussed in the next chapter — Formal Fault Tree Analysis of Observational Transition Systems.

Chapter 5

Formal Fault Tree Analysis of Observational Transition Systems

This chapter presents our formal fault tree analysis model based on OTS. It focuses on analyzing state transition systems using fault trees, which seems to be a common problem of traditional FTA, especially on the representation of state transitions and decomposition of complex system states which consist of event combinations rather than component state itself as we figured out in Section 1.2.

The formal FTA model discussed in this chapter can be regarded as a simplification and revision of our formal fault tree construction model based on temporal logic discussed in the last chapter. There are two important motivations why we further develop our formal FTA based on OTS instead of temporal logic as follows. First of all, since our formal system specification and verification are based on OTS/CafeOBJ, proposing a formal FTA model based on the same underlying conceptual model — OTS, can make the combination of FTA and OTS/CafeOBJ more *consistent*, and thus makes the communication between the experts and analyzers as well as the transformation between these two processes more efficient and smooth. Second, as we know, one of the most important advantage of traditional FTA is that it can be easily drawn and reviewed by experts and then used by analyzers thanks to its simple Boolean logic and graphic support. To this end, generally speaking, a formal fault tree model based on classical Boolean logic and basic concepts of state transition is more desirable than the temporal logic one, which can also be regarded as an alternative for the experts who are not familiar with temporal logic.

Therefore, to achieve the above goals, we first present a refined formal semantics of event of fault trees, and then discuss transition rules and sub-event patterns for fault tree construction, together with corresponding augmented notation and semantics of logic gates. Finally, a formal FTA of OTS is presented based on the above augmented fault tree semantics.

5.1 Definitions of Events

In traditional fault tree analysis terminology, the nodes in a fault tree are called fault events, often meaning the occurrence of *one* specific undesired system or component state [VGRH81, Bah97] (see the definition of fault occurrence in Section 2.1.2). The events can be understood as the root node, intermediate nodes, or leaves in a fault tree

corresponding to the top event, intermediate events, and basic events respectively (see Figure 2.1, page 12), depending on the event is classified as "state-of-system" or "state-of-component" (see Rule 2.2 for fault tree construction, page 15).

But in the sense of OTS, a fault event often refers to a conjunction of *several normal* object states, such as in Section 3.1.1, the event *Collision* was defined as a conjunction of two normal object states: *train on crossing* and *barriers not closed* (see Equation (3.1), page 31). The point here is how to decompose such conjunct fault event. In numbers of formal approaches of FTA [HR98, LM99], events are just decomposed according to the structure of the formula describing the event. But in our example, to resolve the $Collision$ event, we can not just simply decompose it into two sub-events as $Crossing(tr)$ and $Open(ba)$, since neither of them is a fault independently and nobody would like to build a level crossing to ensure that a train never passes the crossing or the barriers are never open [1]. In [RST00], Reif et al. noticed this problem and proposed to "define events and sub-events of a gate separately, and to check their correct interrelation explicitly", however it still did not *explicitly* state how to derive the sub-events in a instructive formal way.

A related issue is that, transitions between states are not represented in traditional fault trees [Lev95], and thus it is difficult to resolve a system fault which consists of event combination (several normal object states) rather than depending on individual component failures. This problem also stems from that traditional FTA mainly focuses on fault occurrence rather than fault existence as for qualitative analysis of fault trees, and thus all the fault events (states) are considered as nonrepairable (unchangeable) [VGRH81]. This principle is useful for hardware system analysis, but it may cause troubles when analyzing transition systems which consist of several objects as we figured out above.

To avoid such misunderstanding and solve the decomposition problem, first we define the event of the fault trees in the sense of OTS as follows.

Suppose the system is modeled as an OTS, and the universal state space of the OTS is called Υ. And we assume that data types D_k $(k = i_1, \ldots, i_m)$ and D are described in initial algebra and there exist visible sorts V_k $(k = i_1, \ldots, i_m)$ and V corresponding to the data types.

Definition 5.1 (Event) *Each event in a fault tree is a specific system or object state, whose occurrence can be interpreted (observed) by an observation (predicate) o_i over a set of typed parameters in a form of*

$$o_i(X_{i_1}, \ldots, X_{i_m}) = c \tag{5.1}$$

where X_k $(k = i_1, \ldots, i_m)$ is a variable or constant with V_k, and c is a constant with V representing the value of the observation o_i (for simplification, in the following discussions, we use the short form o_i to denote $o_i(X_{i_1}, \ldots, X_{i_m})$).

For example, the event a train on crossing can be represented as

$$pos(T1) = crossing$$

[1] Here, one important principle for developing proper fault trees should be stressed, such as figured out in the fault tree handbook [VGRH81]: "all events/nodes that are linked together on a fault tree should be written as faults except, those statements that are added as simply remarks (e.g., CONDITIONING events)".

in the sense of OTS, where *pos* is an observation defined to observe the state of a train, $T1$ is a variable representing a train, and *crossing* is a constant representing the observation value, i.e., the state of the train.

In case there is no parameter in an event, such event usually represents a state of the system or an object which is *unique* (only one instance) and *independent* (its value is unrelated with any other variables) in the system. For the latter, we need not to introduce an extra variable to represent it since there is only one instance of the object. For example, in the crossing control system, there are different (more than one) trains but with only one (a pair of) barriers, therefore, to define the event that the barriers are open, we can formulate it as $bar = open$ instead of $bar(B1) = open$ or $bar(B1, T1) = open$ for simplification.

A special case is that the value of an observation belongs to the Boolean sort, i.e., only $true$ and $false$, we use two simple forms o_i and $\neg o_i$ to represent $o_i = true$ and $o_i = false$, respectively. For instance, the event that the brake of a train is broken can be defined in the following simple form:

$$err\text{-}brake(T1)$$

Definition 5.2 (Classification of Event) *A event can be classified as a normal or fault event depending on whether it is an undesired fault in the system, and the decision is usually made by domain experts. The normal events are usually denoted as CONDITIONING events, while the fault events are denoted as the nodes of the fault trees.*

For example, the above event $pos(T1) = crossing$ is a normal event since it is not an undesired failure in the crossing control system; while the event $err\text{-}brake(T1)$ is a fault event because it is obviously an undesired failure in the system.

However, as discussed in the beginning of this section, some fault events may consist of a conjunction of several normal events, in which neither of them is a fault event independently, but their simultaneous occurrences constitute an undesired fault event. To this end, we propose a concepts of *conjunct fault event* to distinguish the fault events as follows.

Definition 5.3 (Conjunct Fault Event) *A conjunct fault event is a conjunction of several normal events, e.g., $o_i = p \wedge o_j = q$ (p and q are constants corresponding to the observation values of o_i and o_j, respectively), in which neither $o_i = p$ nor $o_j = q$ is a fault event independently, but their conjunction constitutes an undesired failure in the system.*

The importance to distinguish the conjunct fault event is that, in OTS's, a system state can be observed by multiple observations and thus get different observational values from different viewpoints. Therefore, an undesired system fault may consist of a conjunction of several normal events, such as the above example, a train on crossing and the barriers are open are not fault events with respect to the train and the barriers, respectively. Without such distinction, it is difficult to resolve a conjunct fault event correctly using traditional FTA as we discussed above.

It should be figured out that there are two kinds of normal events proposed in traditional FTA, i.e., CONDITIONING and EXTERNAL (HOUSE) events [VGRH81]. The events that are not faults but specific conditions or restrictions are called CONDITIONING-events; while the events that are normally expected to occur are EXTERNAL-events, such as a phase change in a dynamic system [VGRH81]. Since the EXTERNAL-events are not often used and are not involved in our example, we focus our attention on the CONDITIONING-events that are usually applied to the INHIBIT-gates in this thesis, which will be discussed in Section 5.3 — Notation and Semantics of Logic Gates.

It is crucial that the safety engineer and the software engineer agree on the interpretation of the contents of events as formulas [HR98]. The above formulas (definitions) interpret the possible cases of the events in our formal fault tree model, or more specifically, FTA of observational transition systems.

5.2 Analysis of Transition Rules

In our formal fault tree analysis of OTSs, we define an important term, namely *transition rule*.

Definition 5.4 (Transition Rule) *Given an event $o_i = c$, a transition rule states* all *the immediate effective conditions that will change (rewrite) the value of the observation o_i to c in the successor system state.*

A standard transition rule can be represented by the following formula:

$$A_1 \vee A_2 \vee \ldots \vee A_n => o_i = c \tag{5.2}$$

where each A_h $(h = 1, 2, \ldots, n)$ is one possible effective condition which consists of a conjunction of events, and => denotes a one-step transition or rewrite relation in a sense of rewriting logic (it should be distinguished from the strongly implies $\Rightarrow$ of temporal logic and implies $\rightarrow$ of Boolean logic).

One important concept of defining transition rules in fault trees is that, we should consider *all* the possibilities (including physical environment faults such as hardware defects, human errors, and so on) for the event $o_i = c$. In other words, it holds independently of the software-to-be and is related to the completeness of FTA. For example, with respect to the event $pos(T1) = crossing$, a corresponding transition rule can be defined as follows:

$$\begin{aligned} & pos(T1) = sect1 \wedge signal\text{-}bypass(T1) \\ \vee \quad & pos(T1) = sect1 \wedge err\text{-}brake(T1) \\ \vee \quad & pos(T1) = sect1 \wedge level(T1) = state2 \wedge \neg err\text{-}comm(T1) \\ => \quad & pos(T1) = crossing \end{aligned} \tag{5.3}$$

where the constant $sect1$ is defined as the previous state of *crossing* of a train, i.e., after the train sends a 'secure' signal and until it gets a 'release' signal from the level crossing to enter the crossing segment; the event $signal\text{-}bypass(T1)$ denotes that the driver drives the train $T1$ illegally, i.e., bypassing the 'stop' signal; the event $level(T1) = stat2$ denotes

that the level crossing has sent the 'release' signal to the train $T1$, i.e., the observation value of the level crossing is $state2$; and the event $\neg err\text{-}comm(T1)$ denotes that there is no radio communication error between the level crossing and the train $T1$.

Another important issue is that each transition rule in FTA only concerns with one event, even the fault event to analyze may be a conjunct fault event which consists of several normal events. This is because generally speaking, it is very difficult to analyze and find a transition rule that will change several observation values at the same time as we discussed before. By using such kind of divide-and-conquer way, we can focus our attention on one event in each construction step, which is easier and more efficient to analyze the conjunct fault events and construct the fault trees.

Before discussing how to decompose the conjunct fault event with the transition rule (the decomposition of the single fault event is straightforward and can be regarded as a simple instantiation of the conjunct fault event, therefore, in the following discussions we focus our attention on the conjunct fault event), one important constraint of the transition rule should be introduced as follows.

5.2.1 Constraint of Transition Rule

Give a conjunct fault event that consists of two normal events [2], $o_i = p \wedge o_j = q^\alpha$ (Notice here we use q^α and q^β to denote two different values of the same observation o_j. This is because in practice, an observation may have more than two different values with the same sort defined beforehand, and only using q and $\neg q$ may cause misunderstanding and confusing in the following discussions.), and suppose a transition rule for the event $o_i = p$ is as follows:

$$A_1 \vee A_2 \vee \ldots \vee A_n => o_i = p$$

Then for each $A_h\,(h = 1, 2, \ldots, n)$, it can *not* change (cause) the value of o_j to any q^β, where $q^\alpha \neq q^\beta$.

The reason is obvious. If $A_h => o_i = q^\beta$, then it is impossible to derive the previous system state (predecessor) of the conjunct fault event. And if A_h violates the constraint, we should remove A_k from the transition rule. In case no A_h complies with the constraint, then we should consider another transition rule for the event $o_j = q^\alpha$ instead of $o_i = p$.

More specifically, the constraint can be classified into the following three sub-cases depending on whether A_k contains $o_j = q^\alpha$ or $o_j = q^\beta$.

1. If A_h contains $o_j = q^\alpha$, say $A_h \equiv C \wedge o_j = q^\alpha$, where C is a conjunction containing no observation of o_j, then the formula $C \wedge o_j = q^\alpha => o_j = q^\alpha$ must hold [3].

2. If A_h contains $o_j = q^\beta$, say $A_i \equiv C \wedge o_j = q^\beta$, the the transition rule $C \wedge o_j = q^\beta => o_j = q^\alpha$ must also hold.

[2] A conjunct fault event may consists of more than two events, but for better understanding, here we only discuss this simple case, and the complex cases can be derived straightforward in the same way.

[3] Some references defined it as a kind of 'idling transition' [MP92], but here we call it as a formula rather than a transition rule because strictly speaking, it does not change the value of o_j to q^α, and it just states that even with the effective condition C, the observation value of o_j will not be changed in the successor system state.

3. If A_h contains no observation of o_j, say $A_h \equiv C$, then the above two sub-cases may either or both hold. However, with respect to the second one, i.e., $C \wedge o_j = q^{\beta} => o_j = q^{\alpha}$, it is difficult for us to identify the event $o_j = q^{\beta}$ and confirm the above transition rule since there is no knowledge about $o_j = q^{\beta}$ in A_h at this moment. And actually, this transition rule can be covered afterwards when analyzing the event $o_j = q^{\alpha}$. To this end, we only require that the formula $C \wedge o_j = q^{\alpha} => o_j = q^{\alpha}$ holds with respect to this sub-case.

Based on the above analysis, we further present the *patterns* of sub-events to the conjunct fault event, $o_i = p \wedge o_j = q^{\alpha}$, as follows.

5.2.2 Patterns of Sub-events

1. If A_h contains no observation o_j, then the sub-event is: $A_h \wedge o_j = q^{\alpha}$.
2. If A_h contains event either $o_j = q^{\alpha}$ or $o_j = q^{\beta}$, then the sub-event is: A_h.

The proof of the above patterns is straightforward based on the discussion of constraint of transition rules as follows.

Pattern-1 : Since $A_h => o_i = p$ and $A_h \wedge o_j = q^{\alpha} => o_j = q^{\alpha}$, then $A_h \wedge o_j = q^{\alpha} => o_i = p \wedge o_j = q^{\alpha}$, and the sub-event is $A_h \wedge o_j = q^{\alpha}$.

Pattern-2 : Since $A_h => o_i = p$ and $A_h => o_j = q^{\alpha}$, then $A_h => o_i = p \wedge o_j = q^{\alpha}$, and the sub-event is A_i.

After defining the transition rule and the patterns of sub-events, we discuss how to decompose the fault events through a 'one-state regression' procedure, using the example of the conjunct fault event, $pos(T1) = crossing \wedge bar = open$, suppose *bar* is an observation defined to observe the state of the barriers with two values (constants): *open* and *close*.

A transition rule for $pos(T1) = crossing$ has already been found as follows.

$$\begin{aligned} \tau_1 : \quad & pos(T1) = sect1 \wedge signal\text{-}bypass(T1) \\ \vee \quad & pos(T1) = sect1 \wedge err\text{-}brake(T1) \\ \vee \quad & pos(T1) = sect1 \wedge level(T1) = state2 \wedge \neg err\text{-}comm(T1) \\ => \quad & pos(T1) = crossing \end{aligned} \tag{5.4}$$

Then based on pattern-1, we get three sub-events below.

$$\begin{aligned} & pos(T1) = sect1 \wedge signal\text{-}bypass(T1) \wedge bar = open \vee \\ & pos(T1) = sect1 \wedge err\text{-}brake(T1) \wedge bar = open \vee \\ & pos(T1) = sect1 \wedge level(T1) = state2 \wedge bar = open \end{aligned} \tag{5.5}$$

5.3 Notation and Semantics of Logic Gates

After discussing the fault events and transition rules for formal fault tree construction, there is still another problem need to solve, that is, *how to represent the relationship and semantics between the event, sub-events, and transition rules in a proper way.*

An important issue is that, in the standard FTA [VGRH81], it requires that causality never passes through an OR-gate, and the sub-events are just the *restatements* of the top event [VGRH81, Lev95] (Actually, we do not strongly agree with this point. To our understanding, it is better to use time-delay rather than causality to explain the difference, and we will discuss this issue in detail in Section 6.1.1). But in our formal fault tree model, if we decompose a conjunct fault event by a transition rule, and then get several sub-events connected by a OR-gate, the time-delay (cause-consequence) relationship does exist in the OR-gate. To distinguish this difference and explicitly represent the transition rules in the fault trees, we propose the following notations: if the sub-events are regressed though transition rules, then the the corresponding OR-gate must be labeled with the transition rule (or its symbol) for fault tree revising and the denotation of causality; otherwise a standard OR-gate is used.

Actually, this case also happens for other gates, especially when using INHIBIT-gate to decompose and simplify a fault event. To this end, we further propose two kinds of logic gates in our formal fault tree model, that is, a gate labeled with a transition rule is called a Transition gate (or short in T-gate), whose semantics can be interpreted in a form of $Sub\text{-}events => Event$ or $Sub\text{-}events \Leftrightarrow \bigcirc Event$ ($\Rightarrow$ for the correctness condition, and $\Leftarrow$ for the completeness condition of the gate) in the sense of temporal logic; while a gate with no transition rule is a standard gate of traditional FTA, whose semantics can be interpreted as $Event \Leftrightarrow Sub\text{-}events$.

To illustrate the above points, we present an incomplete fault tree based on the hazard collision $pos(T1) = crossing \wedge bar = open$, which was discussed with the analysis of transition rules in the last section (see Figure 5.1).

As shown in Figure 5.1, the first OR-gate is a Transition OR-gate (short in T-OR-gate), which states that three sub-events of this T-OR-gate are regressed by τ_1, and there is a causality passes through this T-OR-gate. Therefore, the semantics of this T-OR-gate is different to the standard OR-gate defined in [VGRH81]. And another OR-gate at the bottom of the fault tree is a standard OR-gate, which corresponds to the Ground Rule II for fault tree construction (see Section 2.1.2, page 15), i.e., if a fault event ($BrakeFailure(tr)$) is classified as "state-of-component," add an OR-gate below the event and look for primary, secondary and command modes ($no\text{-}brake\text{-}signal(T1)$). And in this standard OR-gate, there is no cause-consequence relationship between the event and sub-events.

A similar case is the INHIBIT-gate with the CONDITIONING-event $pos(T1) = sect1 \wedge bar = open$, this is also a standard INHIBIT-gate of FTA in which no transition rule is introduced. Notice here the INHIBIT-gate plays a role to simplify the fault event, because the sub-event $err\text{-}brake(T1)$ can be identified as a fault event regardless of the CONDITIONING-event $pos(T1) = sect1 \wedge bar = open$. By using such kind of simplification, we can focus our attention on the simplified fault event and decrease the

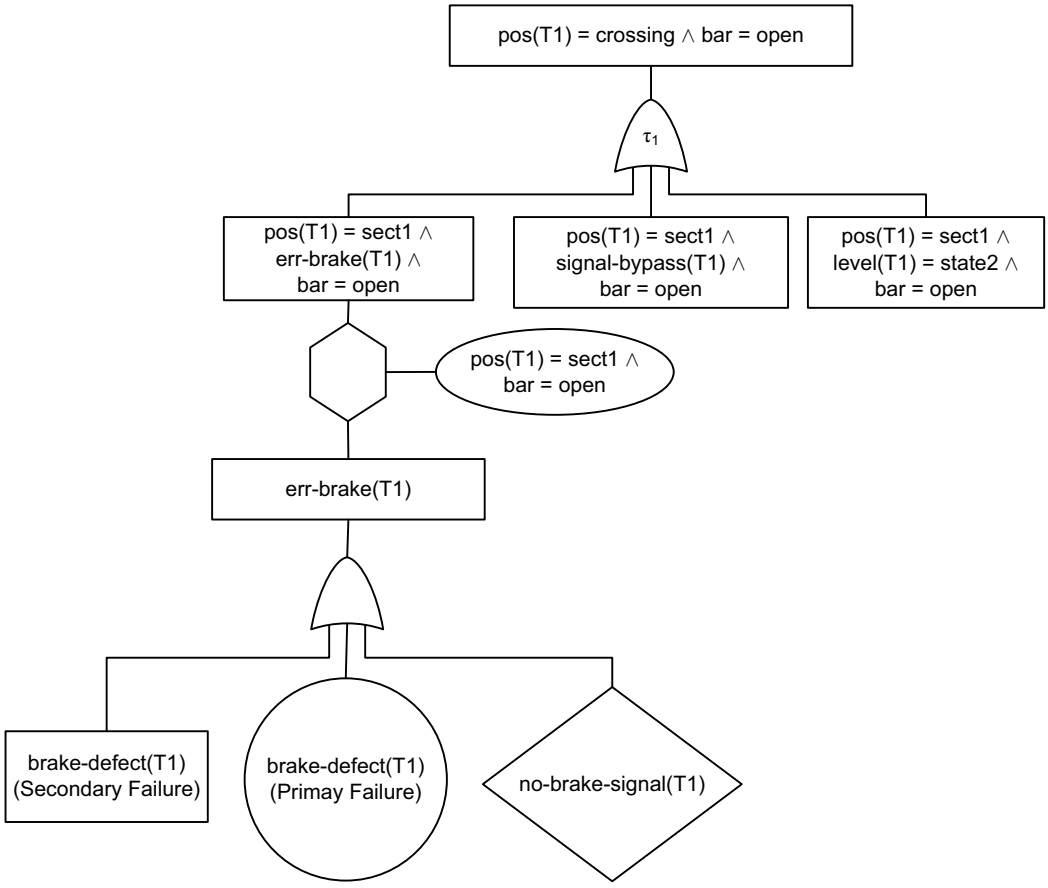


Figure 5.1: Logic gates with transition rules

complexity of the fault tree efficiently.

5.4 Formal Fault Tree Construction Model of OTS

Based on the above discussion and analysis, we present our formal fault tree construction model below.

We take a catastrophic failure as the root node of the fault tree, namely R. Assume R is a conjunct fault event consisting of two normal events, i.e., $o_i = p \wedge o_j = q^{\alpha}$, the regression procedure for the formal fault tree construction is as follows (in case R is not a conjunct fault event that consists of only one fault event $o_i = p$, the regression procedure is straightforward and can be regarded as a simple instantiation of the following procedure).

Initial step Define the formal specification of $R \equiv o_i = p \wedge o_j = q^{\alpha}$.

Regression step

Let $A_1 \vee \ldots \vee A_n \Rightarrow o_i = p$ be the transition rule selected, and A_h $(h = 1, \ldots, n)$ complies with the constraint, that is, $A_h \not=> o_j = q^{\beta}$, where q^{β} denotes any other observation value of o_j except q^{α}.

- For each A_h, if A_h does not contain $o_j = q^{\alpha}$ or any $o_j = q^{\beta}$, then $M_h := A_h \wedge o_j = q^{\alpha}$, else $M_h := A_h$, where M_h is a intermediate variable.
- $R := M_1 \vee M_2 \vee \ldots \vee M_n$

Iteration step

(1) Decompose the resulting R to some sub-events by an *appropriate* logic gate or edge;

(2) Integrate and record the corresponding transition rule into the logic gate or edge for further revision;

(3) Then for each sub event, redo the inductive and iteration steps recursively until a basic event or the chosen abstraction level is reached.

To better understand the above formal fault tree construction model and procedure, we can use the crossing control example from Section 3.1.1 as an illustration. In the crossing control system, the root node of collision is formally defined as:

$\mathbf{R}$: $pos(T1) = crossing \wedge bar = open$

Then we focus on the event "$pos(T1) = crossing$" to analyze which conditions will cause a collision to occur. We get the following transition rule:

$$
\begin{aligned}
\tau_{\mathbf{1}} : \quad & pos(T1) = sect1 \wedge signal\text{-}bypass(T1) \\
\vee \quad & pos(T1) = sect1 \wedge err\text{-}brake(T1) \\
\vee \quad & pos(T1) = sect1 \wedge level(T1) = state2 \wedge \neg err\text{-}comm(T1) \\
=> \quad & pos(T1) = crossing
\end{aligned}
$$

This transition rule complies with the constraint, i.e., neither of A_h will change the value of *bar*, and We thereby formally derive the following formula:

$$\begin{aligned}
&pos(T1) = sect1 \wedge signal\text{-}bypass(T1) \wedge bar = open \vee \\
&pos(T1) = sect1 \wedge err\text{-}brake(T1) \wedge bar = open \vee \\
&pos(T1) = sect1 \wedge level(T1) = state2 \wedge \neg err\text{-}comm(T1) \wedge bar = open
\end{aligned}$$

According to the structure of this formula, we thereby obtain three sub-events by adding an T-OR-gate with the root node **R**.

$\mathbf{S}_{1a}$: $pos(T1) = sect1 \wedge signal\text{-}bypass(T1) \wedge bar = open$

$\mathbf{S}_{1b}$: $pos(T1) = sect1 \wedge err\text{-}brake(S, T1) \wedge bar = open$

$\mathbf{S}_{1c}$: $pos(T1) = sect1 \wedge level(T1) = state2 \wedge \neg err\text{-}comm(T1) \wedge bar = open$

Here, we regard $signal\text{-}bypass(T1)$ and $err\text{-}brake(T1)$ as basic fault events, and since either $pos(T1) = sect1$ or $bar = open$ is a normal event, and their conjunction does not constitute a conjunct fault event, an INHIBIT-gate with the CONDITIONING-event $pos(T1) = sect1 \wedge bar = open$ can be introduced to further simplify $\mathbf{S}_{1a}$ and $\mathbf{S}_{1b}$ as shown in Figure 5.2. Two basic events $\mathbf{B}_1$ and $\mathbf{B}_2$ with a CONDITIONING-event $\mathbf{C}_1$ is defined as follows.

$\mathbf{B}_1$: $signal\text{-}bypass(T1)$

$\mathbf{B}_2$: $err\text{-}brake(T1)$

$\mathbf{C}_1$: $pos(T1) = sect1 \wedge bar = open$

It should be figured out that, the function range of an CONDITIONING-event is limited to the specific logic gate that it connects to. In other words, an event which is regarded as a CONDITIONING-event for a specific logic gate does *not* mean that, this event can also be identified as a global CONDITIONING-event in the fault trees.

Then we focus on $\mathbf{S}_{1c}$, it can also be simplified by an INHIBIT-gate, and we get a simplified conjunct fault event $\mathbf{S}_{1cs}$ and a CONDITIONING-event $\mathbf{C}_2$ as follows.

$\mathbf{S}_{1cs}$: $level(T1) = state2 \wedge bar = open$

$\mathbf{C}_2$: $pos(T1) = sect1 \wedge \neg err\text{-}comm(T1)$

We regard $\mathbf{C}_2$ as a CONDITIONING-event because $\mathbf{S}_{1cs}$ can be identified as a conjunct fault event (the level crossing has sent a 'release' permission to a train, but the barriers are still open), and both $pos(T1) = sect1$ (a train is waiting for the 'release' permission to pass the crossing) and $\neg err\text{-}comm(T1)$ (no radio communication error occurs) as well as their conjunction are normal events.

Focused on the event $level(T1) = state2$ of $\mathbf{S}_{1cs}$, one transition rule to cause the level crossing to send a 'release' signal to a train is that, the level crossing has accepted and confirmed a 'secure' signal from the train, i.e., $level(T1) = state1$. The corresponding transition rule τ_2 is defined as follows.

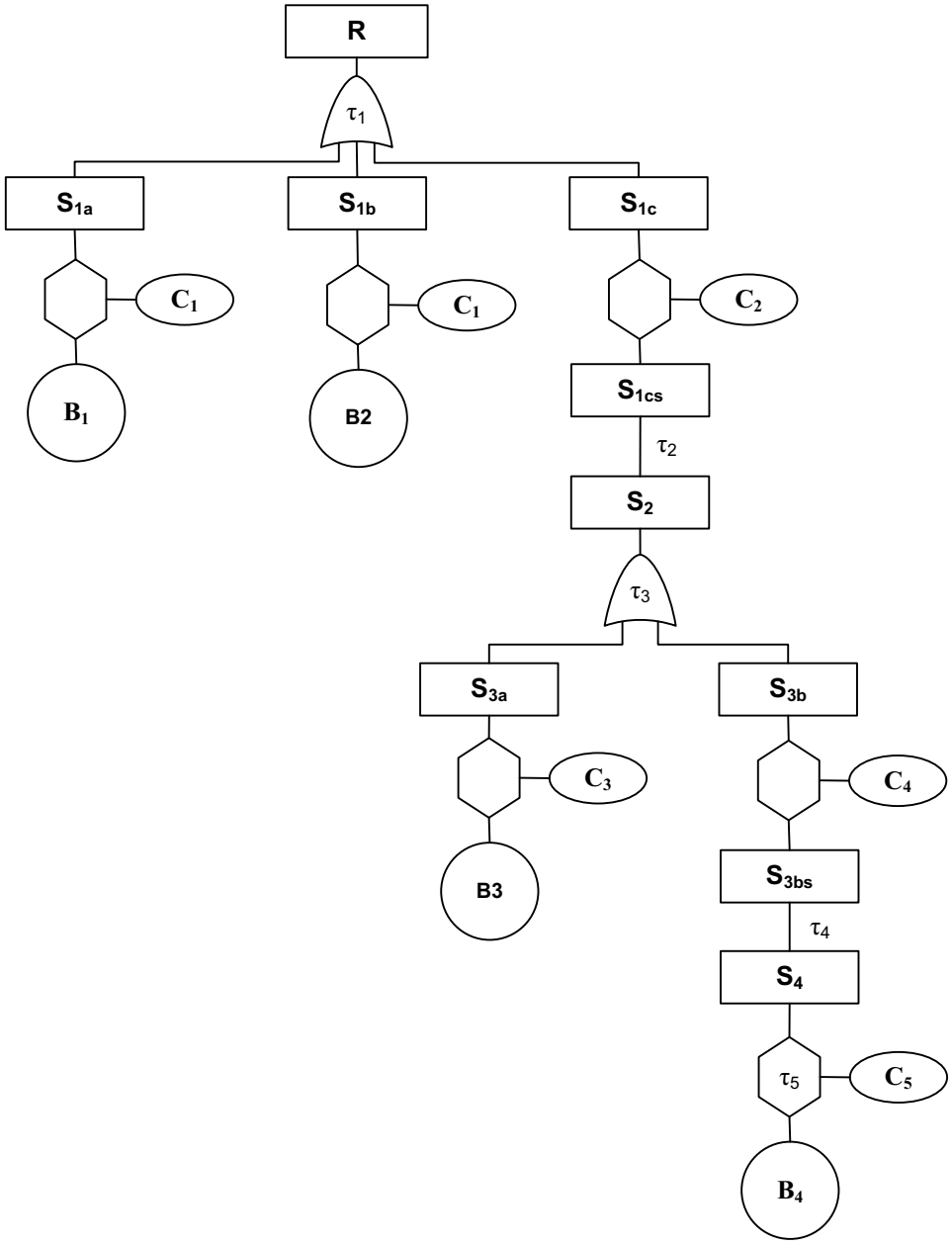


Figure 5.2: Formal fault tree of hazard collision based on OTS

τ_2 : $level(T1) = state1 => level(T1) = state2$

And from τ_2, we get a sub-event $\mathbf{S}_2$ as follows.

$\mathbf{S}_2$: $level(T1) = state1 \wedge bar = open$

Notice here we should use a edge rather than a logic gate to connect $\mathbf{S}_{1cs}$ and $\mathbf{S}_2$, and the transition rule τ_2 should also be labeled besides the edge for revising.

Further analysis will try to regress $level(T1) = state1$, and we would get a transition as follows, that is, if a train has sent a 'secure' signal to the level crossing (denoted by $pos(T1) = sect1$), and there is no radio communication error (denoted by $\neg err\text{-}comm(T1)$), and *the barriers are open* (denoted by $bar = open$), the crossing will accept this request (notice here $bar = open$ is regarded as shared variable to ensure that there is no train on the crossing at this moment, which is equal to the variable $Idling(sys)$ discussed in Section 3.2.2).

$$pos(T1) = sect1 \wedge \neg err\text{-}comm(T1) \wedge bar = open => level(T1) = state1$$

However, based on the system design knowledge, we know that once the crossing has accepted a 'secure' request, then it must close the barriers until it gets a 'passed' signal from the train. In other words, $A => q^{\beta}$ with respect to $\mathbf{S}_2$, which *violates* the constraint of the transition rule. Therefore, we should focus on $bar = open$ and derive another transition rule as follows.

τ_3 : $timeout(S, T2) \vee pos(S, T2) = sect2 \wedge \neg err\text{-}comm(S, T2) => bar(S) = open$

which states that two possibilities for the barriers to open are either a time-out event $timeout(T2)$ (the crossing has been closed and waiting for a '*passed*' signal over a designed time d, and then opens the barriers to protect cars against endless waiting) occurs, or another train $T2$ sends a '*passed*' signal (denoted by $pos(T2) = sect2$) and there is no radio communication error (denoted by $\neg err\text{-}comm(T2)$). Use this transition rule to regress $\mathbf{S}_2$, we get two corresponding sub-events connected by an T-OR-gate below.

$\mathbf{S}_{3a}$: $level(T1) = state1 \wedge timeout(T2)$

$\mathbf{S}_{3b}$: $level(T1) = state1 \wedge pos(T2) = sect2 \wedge \neg err\text{-}comm(T2)$

Notice here a subtle system safety design issues is exposed by $\mathbf{S}_{3b}$, which states that the two events, $level(T1) = state1$ and $pos(T2) = sect2$ can not occur simultaneously, in other words, if the crossing confirm and accepts a 'secure' request from one train, it can not receive a 'passed' signal from another train at this moment.

Keep doing analysis in the similar way, finally we derive all the transition rules and events as follows, and the entire fault tree is shown in Figure 5.2. And for better understanding, we list the explanation of the observations and events in Tabel 5.1.

$\mathbf{B}_3$: $timeout(T2)$

$\mathbf{C}_3$: $level(T1) = state1$

Observation	Values	Explanation
$pos(T1)$	$sect1$	A train has sent a 'secure' signal to the level crossing and is waiting for the 'release' signal before the crossing track segment.
	$crossing$	After getting the 'release' signal, a train is passing on the crossing
	$sect2$	A train has passed the crossing and sent a 'passed' signal to the level crossing.
$level(T1)$	$state1$	The level crossing has accepted a 'secure' request from a train.
	$state2$	The level crossing has sent a 'release' permission to a train.
bar	$open$	The barriers are open.
	$close$	The barriers are close.
$signal\text{-}bypass(T1)$	Boolean	The driver of a train bypasses a 'stop' signal and enters the level crossing illegally.
$err\text{-}brake(T1)$	Boolean	The brake of a train is broken.
$timeout(T1)$	Boolean	A time-out fault occurs.
$err\text{-}comm(T1)$	Boolean	A radio communication error occurs between the level crossing and a train.

Table 5.1: Illustration of observations and events

$\mathbf{S}_{3bs}$: $level(T1) = state1 \wedge pos(T2) = sect2$

$\mathbf{C}_4$: $\neg err\text{-}comm(T2)$

τ_4 : $pos(T2) = crossing => pos(T2) = sect2$

$\mathbf{S}_4$: $level(T1) = state1 \wedge pos(T2) = crossing$

τ_5 : $pos(T1) = sect1 \wedge \neg err\text{-}comm(T1) \wedge bar = open => level(T1) = state1$

$\mathbf{B}_4$: $bar = open \wedge pos(T2) = crossing$

$\mathbf{C}_5$: $pos(T1) = sect1 \wedge \neg err\text{-}comm(T1)$

It should be noted that as for $\mathbf{S}_4$, we focused on $level(T1) = state1$ instead of $pos(T2) = crossing$ because we have found the transition rule for $pos(T1) = crossing$ before (Notice here $T1$ and $T2$ are just variables denoting different trains, so the above two formulas/events can be understood as the same). An subtle issue is that finally we get $\mathbf{B}_4$, which is equal to the Root node, $\mathbf{R}$, i.e., $pos(T1) = crossing \wedge bar = open$. Here, an important system safety issue was disclosed, that is, the fault events linked on the branch from $\mathbf{R}$ to $\mathbf{B}_4$ are mutual dependent (shown in Figure 5.2). And in contrast, the corresponding safety properties or invariants (the negations of the fault events which are related to system design compared with those fault events linked on the other branches) are also *mutual dependent*.

Therefore, from Figure 5.2, we can get the following important information, while most of them are difficult to find by tradition FTA.

- Three basic events, *signal-bypass*($T1$), *err-brake*($T1$), and *timeout*($T1$) should be taken into account with respect to hazard collision, and their corresponding conditions (CONDITIONING-events connected with the corresponding INHIBIT-gates) are also useful for taking precautions.

- Some *mutual exclusion* problems have been discovered , such as the event $\mathbf{S}_4$, which states that when a train on the crossing, the crossing can not responds a 'secure' request from another train.

- An *mutual dependency* relation between the fault events has been discovered, which has also been demonstrated in our formal verification in the last Chapter.

- All the transition rules founded in FTA can be reused in subsequent system formal modeling and verifying, together with deep understanding of the system and primary design scheme followed by the stepwise and instructive fault tree construction processes.

- Last but not least, the correctness and completeness of the fault tree are guaranteed by the transition rules, which makes it possible to revise and recheck the fault tree in a easy and instructive way.

5.5 Summary

In this chapter focused on the incorrectness problem of traditional FTA, we have presented an approach to formal fault tree construction based on the OTS model. In our model, the semantics of fault events has been defined, and as for the conjunct fault events that seems difficult to analyze by traditional FTA, an one-state regression procedure has been proposed, by introducing the concept of transition rules. Therefore, the correctness of the fault tree is guaranteed by the transition rules and the fault tree construction process.

Compared with traditional FTA [VGRH81]and some of its formal models [HR98, STR02], the advantages of our formal fault tree construction model are as follows.

- It is a deductive method to build fault trees more precisely and effectively;

- The correctness of our formal fault tree is proved by the construction process itself, thus avoiding the problems that often arise with traditional methods. At the same time, it gives designers and domain experts the ability to discover transition rules and important design principles in an instructive way during the construction process;

- We integrate the transition rules into the fault tree, which makes the formal fault tree more complete and is useful for further revising and rechecking of the fault trees.

Compared with our formal fault tree construction model of temporal logic (or short in FTA/TL) proposed in Chapter 3, there are two important benefits of this formal fault tree analysis of observation transition systems (or short in FTA/OTS). First of all, since currently our research is mainly based on OTS/CafeOBJ, it makes the combination of

these two techniques and the transformation from system safety analysis to formal system specification and verification more consistent and efficient. In addition, it also provides an alternative for experts who are not familiar with temporal logic.

The combination of FTA and OTS/CafeOBJ is an important issue because it provides an approach/mechanism to transform the results of safety analysis into formal system specification (requirements analysis) more efficiently and effectively. In the next chapter, we will further discuss how to transform the results of FTA/OTS into formal system specification with OTS/CafeOBJ, and how to link fault tree analysis to program design and development by using the common signature and understanding model of OTS.

And as a complement and extension of the formal fault tree models proposed in this thesis, in the next chapter, we are also tying to present some objective analysis and comparison between our formal fault tree models and some other existing formal FTAs based on temporal logic.

Chapter 6

Analysis of Formal FTAs and Requirements Specifications

In this chapter, we are trying to present an analysis of different formal FTAs, and we focus our attention on the following two issues.

- Temporal semantics of fault trees. Currently, several temporal semantics have been proposed for formal fault tree analysis [BA93, HR98, STR02], and our formal fault tree construction models are also based on temporal logic [1], therefore, it is necessary to analyze the difference between these methods as well as the role of temporal logic in fault tree analysis.

- Global correctness and completeness of fault trees. Once temporal semantics has been introduced for fault tree constructors (logic gates and edges), the proof of global correctness and completeness of the fault trees becomes an non-trivial problem, and it generally is more complicated compared with the traditional fault trees which are based on simple Boolean logic. Therefore, we will try to prove the global correctness and completeness of our two proposed formal fault tree models.

- The combination between fault tree analysis and formal system specification and verification, i.e., how to link the safety analysis to requirements specification and verification in a consistent way. This issue is important because in practice, a common framework (model) should be provided for engineers and designers from multiple disciplines to work together more efficiently as for more effective and precise requirements analysis.

Therefore, in the work described here, we first present an objective analysis of different formal FTAs, especially on the role of temporal logic for fault trees. Then, we prove the global correctness and completeness of our two formal FTA models. Finally we focus on how to link fault tree analysis to program design and development, propose a transformation from the results of fault tree analysis to formal system specifications with OTS/CafeOBJ, by using the common signature and understanding model, OTS. This chapter can be regarded as an extension and complement of our formal fault tree analyses proposed in this thesis, which aims at an important issue of requirements analysis, that is,

[1] One is based on the monotonicity of temporal logic, the other is based on the OTS model which can also be interpreted by temporal logic.

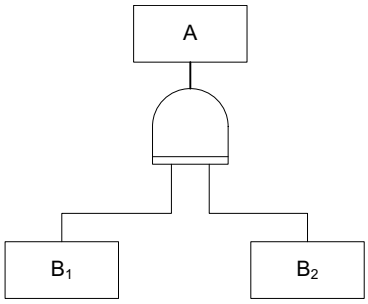


Figure 6.1: Fault tree with PRIORITY AND-gate

what is and how to provide a nice formalism for fault trees to support both more reliable safety analysis and more efficient requirement analysis? And we demonstrate that our FTA/OTS model can be a good solution for this problem.

6.1 Temporal Semantics of Fault Trees

In this thesis, we have proposed two formal fault tree models based on linear temporal logic (LTL) and classical propositional logic with the basic concepts of observational transition system (OTS), and the latter can also be interpreted by temporal logic[2]. Some other formal works of fault trees with temporal logic can be found in [HR98, STR02, BA93], and a discussion about the advantages as well as disadvantages of supporting fault tree analysis with or without temporal logic can be found in [BA93]. In this section, we will first introduce and further analyze the difference of different semantics of fault trees in detail, and then present our formal semantics of fault trees proposed in this thesis, which constitutes the basis of the proof of the global correctness and completeness of the fault trees discussed in the next section.

6.1.1 The Role of Temporal Logic in FTA

There are three main timing issues in fault trees: timing-order between sub-events, timing-relation between event and its sub-events, and synchronism and asynchronism of sub-events. In addition, a related issue is the causality of AND- and OR-gates, which seems to be a vague issue in traditional FTA [VGRH81, Lev95].

Timing-order between sub-events

This case usually happens when defining the formal semantics of PRIORITY AND-gate, which requires that the (output) event occurs when all of the (input) sub-events occur in a specific (usually left-to-right) order. For example, in [HR98], the formal semantics of a fault tree with PRIORITY AND-gate (see Figure 6.1) is defined as follows.

$$A \stackrel{def}{=} B_1 \wedge (\Diamond B_2)$$

The above temporal formula explicitly specifies that B_1 and B_2 occur in a left-to-right order by introducing the temporal Eventually operator $\Diamond$.

[2]In this case, we can use the temporal Next operator $\bigcirc$ to represent the transition between states.

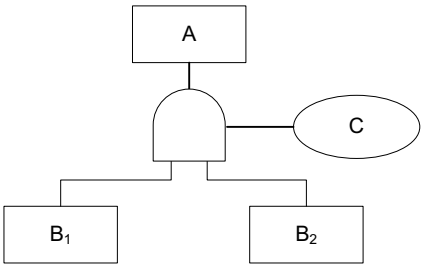


Figure 6.2: Fault tree with AND-gate and CONDITIONING-event

Another solution to represent such timing-order without temporal logic can be found in the Fault Tree Handbook [VGRH81], that is, by introducing a CONDITIONING event C connecting to a standard AND-gate, in which the semantics of C can be interpreted by a predicate as "B_1 occurs before B_2". The corresponding fault tree is pictured in Figure 6.2 , and the semantics of the tree can be interpreted in an equivalent form by propositional logic as follows.

$$A \stackrel{def}{=} B_1 \wedge B_2 \wedge C$$

Timing-relation between sub-events and event

Generally speaking, traditional FTA does not consider time delays between sub-events (causes) and events (consequences), this is mainly due to the fact, that only hardware systems are analyzed in the fault tree handbook [VGRH81]. In this case, time delay is not an important issue, and a system fault state can be simply partitioned into some component failures, such as a circuit system.

This problem becomes an important issue when we want to to make the fault tree specification more precise, and to distinguish the time-delay explicitly from the immediate cases. To this end, some researchers propose to define two different semantics and gates to represent and distinguish the immediate and time-delay relation between the sub-events and event [BA93, Gór94, STR02]. For example, in [BA93], G. Bruns et al. propose two temporal semantics for gates, and here we use a fault tree with AND-gate for illustration as follows.

Given a fault tree with AND-gate which is pictured in Figure 6.3, if there is no time-delay between the sub-events B_1 and B_2 and the top event A, the semantics of this fault tree can be defined as:

$$A \stackrel{def}{=} B_1 \wedge B_2$$

In case the time-delay should be considered and explicitly specified, i.e., B_1 and B_2 happen before A, then the semantics of it can be defined by the following temporal formulas:

$$\begin{aligned} \text{sufficient condition (local correctness)} &: B_1 \wedge B_2 \Rightarrow \Diamond A \\ \text{necessary condition (local completeness)} &: A \Rightarrow \Diamond\!\!\!-(B_1 \wedge B_2) \end{aligned}$$

And in [STR02], G. Schellhorn et al. further propose two different gates to distinguish this difference. One is the standard gates of traditional FTA, called decomposition gates (short: D-gates); the other is cause-consequence gates (short: C-gates) which require that the cause must occur before the consequence, and more subtle semantics are defined for

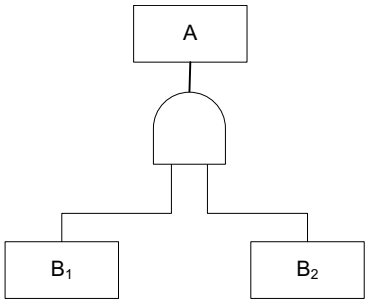


Figure 6.3: Fault tree with AND-gate

the C-gates based on Interval Temporal Logic (ITL).

However, such time-delay issue is disputed by some other researchers because they think that it allows an *optimistic* interpretation of fault trees in the sense that a system failure may be avoid if the operator intervenes fast enough, has enough luck, etc., while such speed, luck, and the like should not be design parameters in safety critical systems [HR98]. Therefore, Hansen et al., suggest to take the *pessimistic* semantics for fault trees, i.e., neglecting the time-delay between the sub-events and event even such a delay may exist.

To our experience, the time-delay should be explicitly represented when we use transition rules (domain rules with time-delay between the antecedence and consequence) to decompose a (conjunct) fault event. And we use transition gates (short: T-gates) and delay gates (short: D-gates) to distinguish the standard gates, especially the standard OR-gate that requires no causality and time-delay between the sub-events and output event [VGRH81]. This is an issue related to the difference of causality between standard AND- and OR-gates that we will discuss later.

Another issue is that if we introduce time-delay into the fault tree semantics, the minimal cut sets can not generally be obtained [BA93] (mostly because the temporal Eventually operator can not be distributed over conjunction/AND-gate), and the global correctness of the fault trees can not be proved considering the synchronism of the sub-events of AND-gate [STR02]. This is an issue we will discuss in Section **??**.

Synchronism and asynchronism of sub-events

This problem has been firstly noticed by Hansen et al. in [HR98], and in [STR02], G. Schellhorn et al. further proposed a kind of Asynchronous Cause-consequence And-gate (short: AC-AND-gate). The synchronism and asynchronism of sub-events are usually considered in AND-gate because sometimes we should distinguish whether the sub-events occur simultaneously or not to cause the output event; while in OR-gate, we need not consider such difference.

For example, with respect to the fault tree with AND-gate as shown in Figure 6.3, the standard semantics of it is defined as follows,

$$A \stackrel{def}{=} B_1 \wedge B_2$$

which states (supposes) that A occurs iff B_1 and B_2 occur simultaneously.

If we consider a more liberal interpretation of this AND-gate in which B_1 and B_2 need not occur simultaneously, we get the following semantics, which is defined as the AC-AND-gate in [STR02].

$$A \stackrel{def}{=} \diamondsuit B_1 \wedge \diamondsuit B_2$$

However, this interpretation has been finally rejected by G. Hansen et al. because they think that the above formula remembers both occurrences of B_1 and B_2, such that if B_2 becomes true 1 year after B_1, then A holds, and they think this is not the intended meaning of an AND-gate [HR98].

Furthermore, to our understanding, even B_1 and B_2 occurred asynchronously, one important fact is that, when A occurs, both B_1 and B_2 must (still) *hold* at this observational time. This is the intended meaning of an AND-gate. From this point of view, we think that it is not necessary to distinguish the difference and increase the complexity of the problem. Even in some special cases where the asynchronism must be specified, we can also use CONDITIONING-event instead of developing additional gates to solve this problem.

Causality of AND- and OR-gate

We focus on causality of AND- and OR-gate because it is an *vague* issue in traditional FTA. For example, there are several important statements in [VGRH81, Lev95]:

> It is important to understand that causality never passes through an OR-gate ([VGRH81, page IV-5]).
>
> One way to detect improperly drawn fault trees is to look for cases in which causality passes through an OR-gate. This is an indication of missing AND-gate and is a sign of improper logic in the conduct of the analysis ([VGRH81, page IV-6])
>
> The input events to an OR-gate do not cause the event above the gate, but are simply re-expressions of the output event. In contrast, the events attached to the AND-gate are the causes of the event above the gate [VGRH81] ([Lev95, pages 318–320]).

However, in these literatures, they did not explain the reason clearly, and there was no concrete example to demonstrate and support these points. This brings us some interesting questions:

- Why causality can not exist in the OR-gate but does exist in the AND-gate? Can we find some examples to illustrate the reason behind?
- In case causality passes through an OR-gate, what would happen? Is it really a serious problem and a sign of improper logic in the conduct of the analysis in every case?

To answer the above questions, we first analyze why causality *may not* exist in an OR-gate but *must* exist in an AND-gate, and then try to demonstrate that in some cases, it is also convenient to specify the input events to an OR-gate as the causes of the output

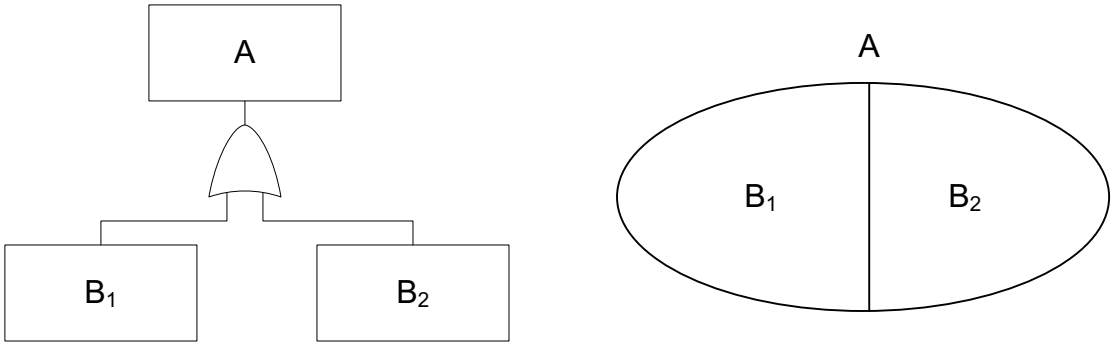


Figure 6.4: Causality of OR-gate

event, and it will not cause big troubles or serious problems as warned by [VGRH81].

Suppose a standard OR-gate consists of two input events B_1 and B_2 and an output event A, the semantics of this OR-gate can be interpreted as $A = B_1 \vee B_2$, which states that the fault space of A can be simply divided/paritioned as an *union* of two sub fault space of B_1 and B_2 (see Figure 6.4). In this case, it is safe to say that there is no causality between the input events and the output event, because the semantics of the OR-gate can be understood as a fault space partition (from higher abstract fault to lower concrete failures). A typical example of this case is using an OR-gate to divide a component fault into three sub fault spaces, i.e., primary fault, second fault, and command fault of the component [VGRH81, page V-3].

In contrast, could we interpret the output event A of an AND-gate as an *intersection* of the two input events B_1 and B_2? If so, then we can conclude that causality may also not exist in the AND-gate. To answer this question, let's consider the right graph of Figure 6.5, if the fault space of event A can be interpreted as an intersection of the fault spaces of B_1 and B_2, then the above hypothesis holds, and the semantics of the AND-gate can be defined as $A = B_1 \wedge B_2$ similarly.

However, if we consider this graph and the relationship between A and B_1 and B_2 more seriously, we will find a structural mistake, i.e., the output event A is a lower concrete failure rather than a higher abstract fault of the input events B_1 and B_2. This of course violates the top-down analyzing approach of standard FTA, and it is obviously unrealistic to attribute a lower level event (such as a component failures) to some higher level events (such as some system-level faults) with an AND-gate in the convention of FTA. Therefore, if we follow the top-down method of FTA, the output event of an AND-gate can never be defined as an intersection of the input events. In contrast, the input events must be defined as the causes of the output event, and the semantics of the AND-gate can be interpreted as $B_1 \wedge B_2 \leftrightarrow A$ [3].

After studying the difference of causality between AND- and OR-gate, we try to analyze the possible harm by introducing causality into the OR-gate as figured out in [VGRH81].

[3]Notice here we use $\rightarrow$ to denote the causality as well as the correctness of the AND-gate, while the $\leftarrow$ can be understood as the completeness of the AND-gate.

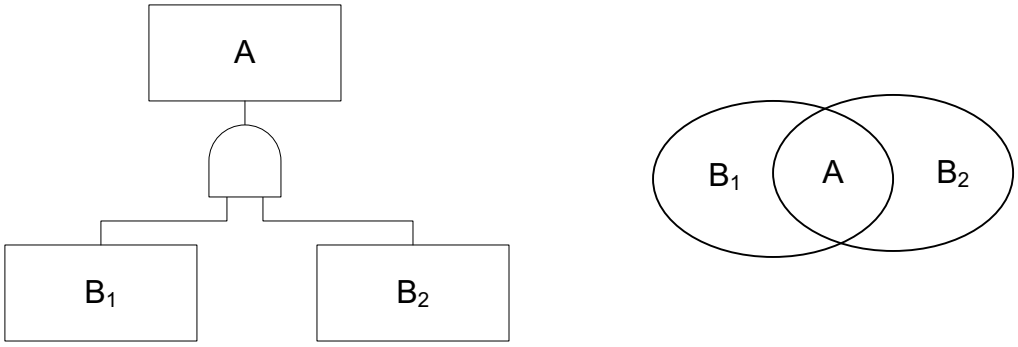


Figure 6.5: Causality of AND-gate

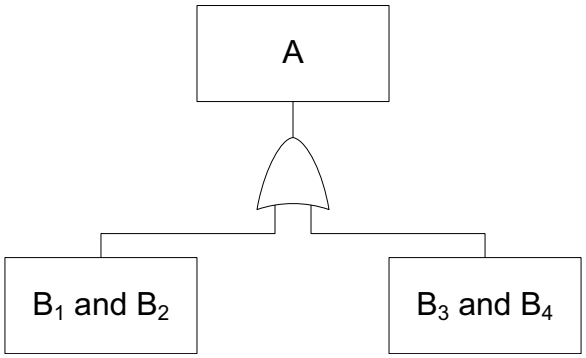


Figure 6.6: Fault tree missing AND-gate

Suppose a system consists of four components b_1, b_2, b_3, and b_4, and assume the component faults are named as B_1, B_2, B_2, and B_4, respectively. Suppose a system hazard A may be caused either if both B_1 and B_2 occur, or if both B_3 and B_4 occur. A corresponding fault tree then can be drawn as shown in Figure 6.6. The problem of Figure 6.6 is that each input event consists of two component faults, and both of them can be regarded as the causes with respect to the output event A, in other words, the causality passes through the OR-gate of the fault tree. From this point of view, it violates the definition of standard OR-gate. To solve this problem, we can introduce two more INTERMEDIATE events (M_1 and M_2) and two AND-gates, and a revised fault tree is shown in Figure 6.7.

There are two benefits of the revised fault tree (see Figure 6.7) compared with Figure 6.6. One is that by introducing one more abstraction level and two INTERMEDIATE events M_1 and M_2, it makes the analysis more complete, and at the same time, causality does not passes through the corresponding OR-gate because M_1 and M_2 can be interpreted as two sub fault spaces (restatements) of A. The other is by adding two AND-gates, we can guarantee that each node of the fault tree consists of a simple component fault rather than a conjunction of two component faults as shown in Figure 6.6, which makes the logic structure of the fault tree more clear and understandable.

The above example seems to support the argument posted in page 92, i.e., once causality exists in an OR-gate, it is an indication of a missing AND-gate and is a sign of the use of improper logic in the conduct of the analysis [VGRH81]. However, if we consider another special case using the same example above, i.e., suppose the system hazard A will

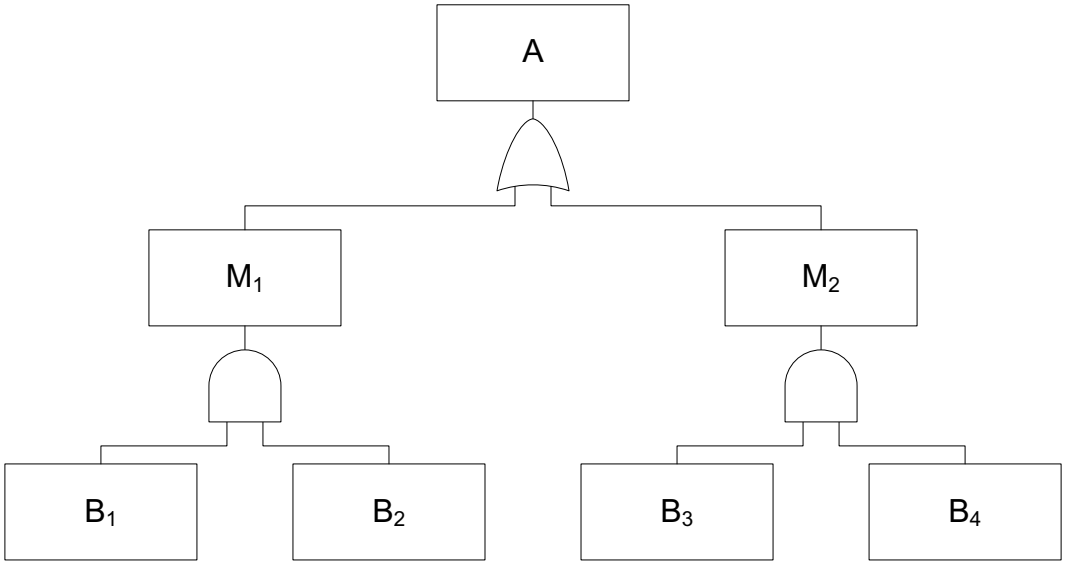


Figure 6.7: Fault tree adding AND-gate

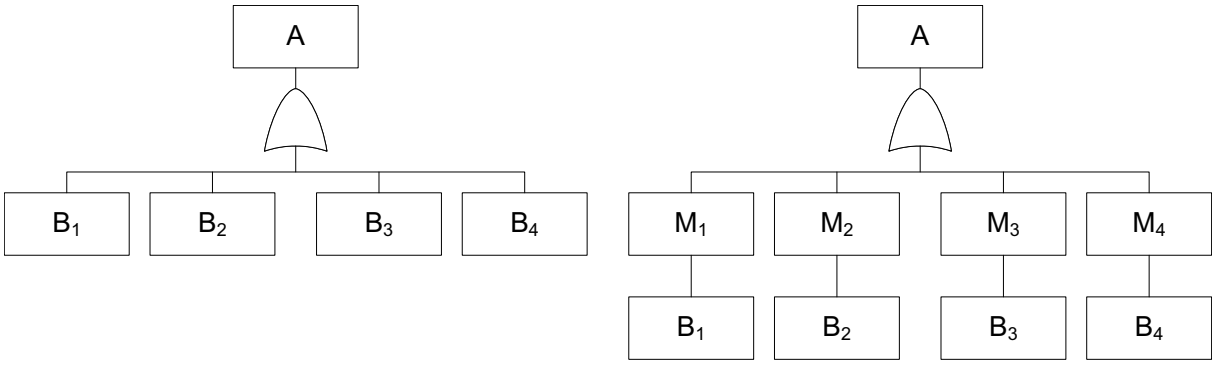


Figure 6.8: Special OR-gate

happen if one or more of the component faults $B_i (i = 1, \ldots, 4)$ occur. With respect to this specific case, since each single component fault is enough to cause the system hazard A, it is convenient and quite natural to use an OR-gate to depict their logic relationship, and the semantics of the generated fault tree can be interpreted as $B_1 \vee B_2 \vee B_3 \vee B_4 \leftrightarrow A$, in which causality does exist (see the lefthand fault tree of Figure 6.8).

People may argue the causality problem of this OR-gate, and to remedy it, a similar solution is to introduce four INTERMEDIATE events (see the righthand fault tree of Figure 6.8). However, with regard to this special case, we think it is unnecessary to stiffly follow the no causality rule, since the introduced INTERMEDIATE events are exactly the same meaning of the component faults (the only difference may be the abstraction levels), and such a solution seems only to make the analysis more complex by introducing redundant levels and events without any benefits.

Moreover, to our understanding, even with respect to the standard OR-gate, the input events (restatements) can also be interpreted as the causes of the output event from a more general viewpoint. Therefore it is unnecessary to insist on the importance of no causality in OR-gate as claimed by [VGRH81, Lev95], since sometimes it may cause confusing and inconvenience as shown in the above example.

	Correctness condition	Completeness condition
AND-gate	$a \wedge b \Rightarrow R$	$R \Rightarrow a \wedge b$
OR-gate	$(a \vee b) \Rightarrow R$	$R \Rightarrow (a \vee b)$
INHIBIT-gate	$(a \wedge C) \Rightarrow R$	$R \Rightarrow a$
D-AND-gate	$a \wedge b \Rightarrow \diamondsuit R$	$\diamondsuit R \Rightarrow a \wedge b$
D-OR-gate	$(a \vee b) \Rightarrow \diamondsuit R$	$\diamondsuit R \Rightarrow (a \vee b)$
D-INHIBIT-gate	$(a \wedge C) \Rightarrow \diamondsuit R$	$\diamondsuit R \Rightarrow a$
T-AND-gate	$a \wedge b \Rightarrow \bigcirc R$	$\bigcirc R \Rightarrow a \wedge b$
T-OR-gate	$(a \vee b) \Rightarrow \bigcirc R$	$\bigcirc R \Rightarrow (a \vee b)$
T-INHIBIT-gate	$(a \wedge C) \Rightarrow \bigcirc R$	$\bigcirc R \Rightarrow a$

Table 6.1: Semantics of gates

Based on the above discussion and analysis, we propose that all the input events of AND- and OR-gates are the causes of their output event, respectively. And instead of emphasizing the causality, we think it is more important to distinguish the time-delay issue of different gates. As discussed before, a gate with time-delay usually represents that the input events are decomposed by a domain or transition rule, which is called D- or T-gate in our formal fault tree models. In contrast, a gate with no time-delay is then a gate of standard FTA, such as the standard OR-gate, it is obvious that there is no time-delay between the output event and the input events if we just use partitioning (fault space splitting) to decompose the output event [4].

6.1.2 Formal Semantics of FTA/TL and FTA/OTS

Based on the above discussions, we present the formal semantics of our two formal fault tree models: fault tree analysis based on temporal logic (FTA/TL, Chapter 3) and observational transition system (FTA/OTS, Chapter 5).

In our formal fault tree models, we use the concepts of D-gate and T-gate to represent the time-delay between the input events and output event, while asynchronism is not considered, and we propose that causality passes through both AND- and OR-gates. Therefore, by using temporal logic, we present the formal semantics of three kinds of logic gates (standard gate, D-gate, and T-gate) discussed in this thesis in Table 6.1, where a, b denote the input events, R denotes the output event, and C denotes the CONDITIONING event.

In Table 6.1, the correctness condition of each gate (or called local correctness) guarantees that if the input events (causes) happen, the output event (consequence) must happen too. The completeness condition (or called local completeness) guarantees that all input *fault* events have been listed: the output event must not happen without the input fault events. Notice here we use the term of "input fault events" instead of "input events" in the completeness conditions because completeness is a concept related to safety analysis, and from the point of view of safety, we do not care about the input *normal*

[4]Notice here the input events can be understood as not only the restatements, but also the causes of the output event.

events (i.e., CONDITIONING-events). In other words, to ensure the output event never happen, we usually prevent that the input fault events will never happen (suppose the completeness condition is satisfied), while not to prevent the CONDITIONING-events. The difference is shown between the completeness conditions of AND- and INHIBIT-gates.

Another related issue is the definition of minimal cut set. In standard FTA, the minimal cut set is defined as a combination of primary fault events 'sufficient' for the top event [VGRH81]. This definition seems no problem if we limit the analysis to only AND- and OR-gates such as discussed in [VGRH81]. However, in case a fault tree consists of CONDITIONING events and some other gates, e.g., INHIBIT-gate, such a definition is not precise enough and may cause confusion and contradiction.

To this end, based on Table 6.1, we further propose two kinds of minimal cut sets of fault trees as follows.

Sufficient Minimal Cut Set : A sufficient minimal cut set SMCS is a smallest combination of primary events (including CONDITIONING events) which, if they occur, will cause the top event to occur.

Necessary Minimal Cut Set : A necessary minimal cut set NMCS is a smallest combination of only primary *fault* events which, if the top event occurs, then they must also occur.

The SMCSs can be calculated by the correctness conditions of gates, while the NMCSs can be calculated by the completeness conditions of gates, respectively. Moreover, from NMCSs, it is straightforward to derive the minimal path sets (the complements of the minimal cut sets), which is important for the prevention of the top event [VGRH81, pages VII-15 – VII-20] and is related to the global completeness of the fault tree that we will discuss in the next section.

6.2 Global Correctness and Completeness of Fault Trees

After analyzing the formal semantics of fault trees, in this section, we discuss and prove the global correctness and completeness of the fault trees.

Firstly, we present an informal description of the global correctness and completeness of the fault trees as follows.

global correctness of fault tree : If all (local) correctness conditions of gates can be proved, then the following holds: if all the primary events of some sufficient minimal cut sets occur, then the top event must also occur.

global completeness of fault tree : If all (local) completeness conditions of gates can be proved, then the following holds: if all the primary fault events of some minimal path sets do not occur(in other words, if for each necessary minimal cut set it is possible to guarantee the non-occurrence of at least one primary fault event), then the top event will never occur.

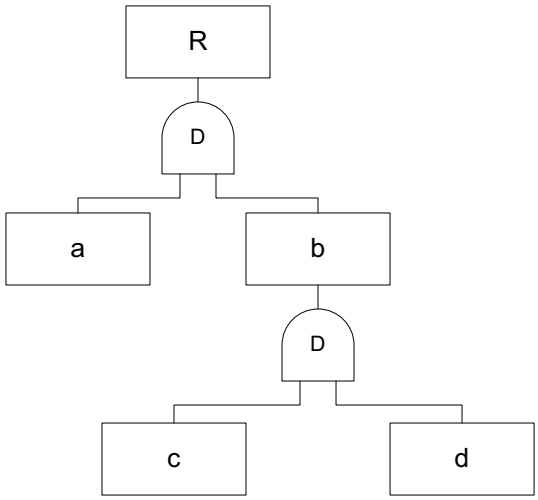


Figure 6.9: Correctness of D-AND-gates

With respect to standard FTA, the proof is straightforward since no temporal semantics are introduced. However, if we introduce temporal semantics for the gates (e.g., D-gates and T-gates), the proof becomes complex. In [BA93], G. Bruns et al. found that once time-delay was introduced for the gates (corresponding to the D-gates in Table 6.1), even the minimal cut sets (corresponding to SMCSs defined in the last section) could not generally be derived. This is because the temporal Eventually operator $\Diamond$ can not be distributed over conjunction, and so as to make the calculation of SMCSs over cascaded AND-gates very difficult.

With regard to this problem, we propose to overlook the temporal semantics of gates when calculating the SMCSs, and then to prove the global correctness based on the generated SMCSs afterwards. Similar solution can also be found in [STR02]. However, even the SMCSs can be calculated as usual by overlooking temporal semantics, the proof of the global correctness is still a hard work. A simple example can be used to illustrate this point as follows.

Suppose a fault tree consists of two D-AND-gates (see Figure 6.9), and assume the local correctness conditions are proved, i.e., $a \wedge b \Rightarrow \Diamond R$, $c \wedge d \Rightarrow \Diamond b$. Our goal is then to prove that $a \wedge c \wedge d \Rightarrow \Diamond R$ holds, where a, c, d constitute the minimal cut set of the fault tree. However, this goal (global correctness) can not be proved because we can not prove that $a \wedge \Diamond b \Rightarrow \Diamond R$ based on the local correctness conditions.

Similar attempt can also be found in [STR02], and they (G. Schellhorn et al.) stated that the global correctness could not hold neither for standard AND-gate nor for D-AND-gate, but only for asynchronous AND-gate. But in our formal fault tree models (FTA/TL and FTA/OTS), the asynchronous AND-gate is not supported. This brings us a problem: should we give up the attempt (the proof of global correctness)?

In contrast, the proof of the global correctness (considering D-AND-gate) is relatively easy. Using the same example, we can easily prove that $(\Box \neg a \vee \Box \neg c \vee \Box \neg d) \Rightarrow \Box \neg R$ holds, where $\neg a, \neg c$, and $\neg d$ are the three minimal path sets of the fault tree. But with respect to our FTA/OTS, since we use temporal Next operator $\bigcirc$ instead of Eventually operator $\Diamond$ to interpret the semantics of T-gates, the above formula of global correctness

does not hold too.

Therefore, focused on the above problems, by using structural induction over the size of the fault tree, we have finally proved the following theorems for the global correctness and completeness with respect to FTA/TL and FTA/OTS, respectively. The proofs have been verified formally using Maude LTL (Linear Temporal Logic) tautology checker [EMS02, CDE+03a].

- FTA/TL (fault tree consists of D-gates and standard gates)

  Global Correctness : If for all gates the correctness conditions hold, then for any SMCS S_i which consists of a conjunction of primary events in a form of $b_{i_1} \wedge \ldots \wedge b_{i_n}$, and suppose R is the top (Root) event, the following formulas hold:

$$\Diamond\,(b_{i_1} \wedge \ldots \wedge b_{i_n} \rightarrow \Diamond\, R) \tag{6.1}$$

$$\Box\,\Diamond\, b_{i_1} \wedge \ldots \Box\,\Diamond\, \wedge\, b_{i_n} \Rightarrow \Diamond\, R \tag{6.2}$$

  Global Completeness : If for all gates the completeness conditions hold, then for any minimal path set P_j which consists of a conjunction of the negations of primary fault events in a form of $\neg b_{j_1} \wedge \ldots \wedge \neg b_{j_n}$, the following formula holds:

$$\Box\,\neg b_{j_1} \wedge \ldots \wedge\, \Box\,\neg b_{j_n} \Rightarrow\, \Box\,\neg R \tag{6.3}$$

- FTA/OTS (fault tree consists of T-gates and standard gates)

  Global Correctness : If for all gates the correctness conditions hold, then for any SMCS S_i which consists of a conjunction of primary events in a form of $b_{i_1} \wedge \ldots \wedge b_{i_n}$, the following formulas hold:

$$\bigcirc_{[l-l_{b_{i_1}}]} b_{i_1} \wedge \ldots \wedge\, \bigcirc_{[l-l_{b_{i_n}}]} b_{i_n} \Rightarrow\, \bigcirc_{[l]} R \tag{6.4}$$

$$\Diamond\,(b_{i_1} \wedge \ldots \wedge b_{i_n} \rightarrow \Diamond\, R) \tag{6.5}$$

  where $l_{b_{i_k}}$ $(k = 1, \ldots, n)$ denotes the transition length of the primary event b_{i_k}, i.e., the number of T-gates and T-edges existing in the path from R to b_{i_k}, and l denotes the longest transition length among b_{i_k}, $\bigcirc_{[l]}$ is the abbreviation of l temporal Next operators.

  Global Completeness : If for all gates the completeness conditions hold, then for any minimal path set P_j which consists of a conjunction of the negations of primary fault events in a form of $\neg b_{j_1} \wedge \ldots \wedge \neg b_{j_n}$, the following formula holds:

$$\Box\,\neg b_{j_1} \wedge \ldots \wedge\, \Box\,\neg b_{j_n} \Rightarrow\, \Box\,\bigcirc_{[l]} \neg R \tag{6.6}$$

$$(\,\Box\,\neg b_{j_1} \wedge \ldots \wedge\, \Box\,\neg b_{j_n}) \wedge (\neg R \wedge \neg e_{x_1} \wedge \ldots \wedge \neg e_{x_y}) \Rightarrow\, \Box\,\neg R \tag{6.7}$$

where $e_{x_1}, \ldots, e_{x_y}$ denote all the intermediate events of the sub dual (complemented) tree with respect to the minimal path set P_j [5].

Notice:

- (6.1) and (6.5) should be regarded as a kind of weaker correctness, since it just states that if all the primary events of a sufficient minimal cut set occur, then it is *possible* for the root event to occur.
- (6.2) only works for traditional fault tree analysis based on the assumption that, a fault event once occurs, it will always hold and there is no other events will repair (change) the fault (see Section 2.1.2: Fault Occurrence vs. Fault Existence) [VGRH81, page V-1 – V-2]. However, this assumption does not hold in our FTA/TL and FTA/OTS in case conjunct fault events are introduced.
- (6.7) is a refinement of (6.6), which states that if the non-occurrence of all the primary fault events of one minimal path set can be always guaranteed, and the non-occurrence of the root event and all the intermediate events of the sub dual tree of the minimal path set can be guaranteed at the current state, then the system will always be safe. The second condition can be regarded as a kind of initialization (initial case) in the sense of theorem proving by induction. In contrast, without the second condition, (6.6) can only ensure that after l transitions, the system will always be safe even we have guaranteed that all the primary fault events will not happen from now on, which seems to be not strong enough for system safety analysis.

6.3 From FTA to Formal System Specification with OTS/CafeOBJ

In this section, we will discuss how to transform fault tree (specification) into formal system specification with OTS/CafeOBJ. Our intention is to provide a nice formalism of fault trees, and so as to combine the safety analysis and formal system specification with OTS/CafeOBJ (requirements analysis) more consistently and smoothly.

Since we have proposed a formal fault tree model based on OTS, which uses the same underlying conceptual model of formal system specification with OTS/CafeOBJ, the transformation and interaction between these two techniques and processes becomes possible. On one hand, the results of FTA can be used not only limited to safety analysis, but more importantly, it also contributes and constitutes the core part of the formal specification of the system. On the other hand, the complemented and refined formal system specification can assist us to further revise the fault trees and get more reliable safety analysis.

To achieve this goal, a common signature for both fault trees and formal specification with OTS/CafeOBJ should be provided, and a transformation from the transition rules

[5] The dual of the original fault tree can be obtained directly from the original tree by complementing all the events and substituting OR-gates for AND-gates and vice versa [VGRH81], and thus with respect to each minimal path set, we can divide the generated dual tree by OR-gates and derive a corresponding sub dual tree.

of fault trees to axioms (actions) of OTS/CafeOBJ should be studied. In the following discussions, we will use the fault tree developed in the last chapter (see Figure 5.2, page 83) as an example to illustrate the above issues.

6.3.1 Common Signature for Fault Trees and OTS/CafeOBJ

An OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ described in CafeOBJ usually consists of two parts, one is the signature part including the definition of sorts and operators, the observations $o \in \mathcal{O}$ and transitions $\tau \in \mathcal{T}$ of the OTS are also denoted (declared) by CafeOBJ observation and action operation (operator)s, respectively; the other is the axioms part which consists of a set of equations describing how the values of the observations are changed by actions, i.e., the definition of actions.

In our FTA/OTS model, there are two important constructors (integrants): events and transition rules for event decomposition. A event in FTA/OTS is defined as an observation *value* over a set of typed variable, therefore, it is quite natural to transform the events into CafeOBJ observation operations as follows.

Suppose OTS $\mathcal{S}$ is described in CafeOBJ, and the universal state space Υ is denoted by a hidden sort, say `Sys`, by declaring `*[Sys]*`. And we assume that data types D_k $(k = i_1, \ldots, i_m)$ and D are described in initial algebra and there exist visible sorts V_k $(k = i_1, \ldots, i_m)$ and V corresponding to the data types. An event of fault tree can be observed by an observation o_i as follows.

$$o_i(S, X_{i_1}, \ldots, X_{i_m}) = c \tag{6.8}$$

where S is a variable belonging to the hidden sort `Sys` [6], X_k $(k = i_1, \ldots, i_m)$ is a variable with V_k, and c is a constant with V representing the value of the observation o_i.

Correspondingly, the CafeOBJ specification denoting the observation o_i and the event can be declared straightforward as follows:

```
op c :  -> V
bop oi :  Sys Vi1 ... Vim -> V
```

It should be noted that with respect to a set of events discovered in the fault trees which are observed (based on) by the same observation o_i, we need not to define a set of corresponding observations in the CafeOBJ signature. Instead of that, we need only to declare that all of the values of the observation o_i as a set of constants with the same data type V. For example, suppose there are n events observed by the same o_i as follows.

$$\begin{aligned} o_i(S, X_{i_1}, \ldots, X_{i_m}) &= c_1 \\ &\vdots \\ o_i(S, X_{i_1}, \ldots, X_{i_m}) &= c_n \end{aligned}$$

[6] Notice here we add the variable S with `Sys` in o_i in the sense of OTS/CafeOBJ, which is omitted in the definition of event in the fault trees (see (5.1), page 74).

where $c_1, \ldots, c_n$ are constants with V representing different observation values of o_i.

Then, in the CafeOBJ specification, we need only to declare as follows.

```
ops c_1 ··· c_n :  -> V
bop o_i :  Sys V_{i_1} ... V_{i_m} -> V
```

For example, with respect to the events based on the observation $pos(S, T1)$ in Table 5.1 (page 85): $pos(S, T1) = sect1$, $pos(S, T1) = crossing$, and $pos(S, T1) = sect2$, the corresponding CafeOBJ specification can be declared as follows.

```
ops sect1 crossing sect2 : -> TState -- states of a train
bop pos : SYS Train -> TState
```

A transition rule τ_i found in FTA is denoted by a CafeOBJ action operation. We assume that data types D_k $(k = i_1, \ldots, i_m)$ and D are described in initial algebra and there exist visible sorts V_k $(k = i_1, \ldots, i_m)$ and V corresponding to the data types. The CafeOBJ action operation denoting τ_i is declared as follows.

```
bop a_i :  Sys V_{i_1} ... V_{i_m} -> Sys
```

where V_k $(k = i_1, \ldots, i_m)$ represents the data type of the variable X_k $(k = i_1, \ldots, i_m)$ declared in τ_i.

For example, a transition rule τ_5 found in the fault tree in the last chapter, i.e.,

$$pos(S, T1) = sect1 \wedge \neg err\text{-}comm(S, T1) \wedge bar(S) = open \Rightarrow level(S, T1) = state1$$

can be declared in the CafeOBJ specification as follows.

```
bop a5 : Sys Train -> Sys  -- the level crossing accepts a 'secure'
                           -- request from a train
```

Therefore, the fault trees developed by our FTA/OTS provide all the necessary and concrete information to write the formal specification (signature part) with OTS/CafeOBJ. In other words, since they share a *common* signature framework (the fault tree signature can be regarded as a *core subset* of the signature in OTS/CafeOBJ), the transformation between them is straightforward as we demonstrated above. For instance, with respect to the fault tree developed in the last chapter (see Figure 5.2, page 83; and Table 5.1, page 85), we can easily and quickly construct the signature of the OTS with CafeOBJ as shown in Figure 6.10.

6.3.2 From Transition Rules to Axioms

The second part of the formal system specification with OTS/CafeOBJ is the axioms, which defines the variables and (conditional) equations describing how the observation values are changed by the execution of actions. In this section, we discuss how to transform the transition rules of fault trees into the axioms of CafeOBJ specification.

```
-- sorts and constants declaration
  [Train]  -- sort denoting different trains

  [TState] -- sort denoting train state
  ops sect1 crossing sect2 : -> TState

  [LState] -- sort denoting level crossing state
  ops state1 state2        : -> LState

  [BState] -- sort denoting barriers state
  ops open close           : -> BState

  *[Sys]*  -- hidden sort denoting universal system state

-- declaration of observations
  bop pos   : Sys Train -> TState
  bop level : Sys Train -> LState
  bop bar   : Sys       -> BState
  bop signal-bypass : Sys Train -> Bool
  bop err-brake     : Sys Train -> Bool
  bop timeout       : Sys Train -> Bool
  bop err-comm      : Sys Train -> Bool

-- declaration of actions
  bop a1 : Sys Train -> Sys -- cause a train on crossing
  bop a2 : Sys Train -> Sys -- the level crossing sends a 'release' signal to a train
  bop a3 : Sys Train -> Sys -- cause the barriers to open
  bop a4 : Sys Train -> Sys -- a train sends a 'passed' signal to the level crossing
  bop a5 : Sys Train -> Sys -- the level crossing accepts a 'secure' signal from a train
```

Figure 6.10: Signature of OTS/CafeOBJ from FTA/OTS

Suppose an observation o_i and an action a_j are declared in CafeOBJ as follows:

```
bop o_i :  Sys V_i1 ... V_im -> V
bop a_j :  Sys V_j1 ... V_jm -> Sys
```

The conditional equation defining that the value of o_i may be changed by a_j can be described in CafeOBJ generally as follows:

```
ceq o_i (a_j (S, X_j1, ..., X_jm), X_i1, ..., X_im)
    = e-a_j(S, X_j1, ..., X_jm, X_i1, ..., X_im) if c-a_j(S, X_j1, ..., X_jm) .
```

where `e-a`$_j$`(S, X`$_{j_1}$`, ..., X`$_{j_m}$`, X`$_{i_1}$`, ..., X`$_{i_m}$) means a term (consisting of `S, X`$_{j_1}$`, ..., X`$_{j_m}$`, X`$_{i_1}$`, ..., X`$_{i_m}$) corresponding to the value of `o`$_i$ in the successor state, and `c-a`$_j$`(S, X`$_{j_1}$`, ..., X`$_{j_m}$) means a term (consisting of `S, X`$_{j_1}$`, ..., X`$_{j_m}$) corresponding the effective condition of `a`$_j$.

And a transition rule in fault trees is usually defined in the following form.

$$A => o_i = c$$

where A is a disjunctive normal form consisting of a set of effective conditions for the event $o_i = c$. Therefore, it is quite naturally to transform a transition rule of the fault trees into the axioms of CafeOBJ, just regarding A as `c-a`$_j$`(S, X`$_{j_1}$`, ..., X`$_{j_m}$), and the value c as `e-a`$_j$`(S, X`$_{j_1}$`, ..., X`$_{j_m}$`, X`$_{i_1}$`, ..., X`$_{i_m}$), respectively.

For example, with respect to the transition rule (declared as `a5` in the signature of CafeOBJ in the last section)

$$pos(S, T1) = sect1 \wedge \neg err\text{-}comm(S, T1) \wedge bar(S) = open => level(S, T1) = state1$$

The corresponding axioms can be described in CafeOBJ as follows.

```
var  S     : Sys
vars T1 T2 : Train

ceq level(a5(S, T1), T2) = (if (T1 = T2) then state1 else level(S, T2) fi)
    if (pos(S, T1) = sect1 and not err-comm(S, T1) and bar(S) = open) .
```

It should be noted that in the above condition equation, we discussed the possibility whether `T1 = T2`, this is because in the OTS of the crossing control system, there may be more than one train in the system.

If the value of o_i is not affected by executing a_j in any state (regardless of the truth value of `c-a`$_j$`(S, X`$_{j_1}$`, ..., X`$_{j_m}$) or A), which is the case when using the transition rule to decompose a conjunct fault event (i.e., the constraint of transition rules defined in the last chapter), the following equation is declared:

```
eq o_i (a_j (S, X_j1, ..., X_jm), X_i1, ..., X_im) = o_i(S, X_i1, ..., X_im) .
```

```
-- variables
  var  S     : Sys
  vars T1 T2 : Train

-- transition rule a1: train passes on crossing
  ceq pos(a1(S, T1), T2) = (if (T1 = T2) then crossing else pos(S, T2) fi)
      if ((pos(S, T1) = sect1 and signal-bypass(S, T1) = true)
           or (pos(S, T1 = sect1) and err-brake(T1))
           or (pos(S, T1) = sect1 and level(S, T1) = state2) and not err-comm(S, T1)) .
  eq  bar(a1(S, T1)) = bar(S) .

-- transition rule a2: level crossing send 'release' signal
  ceq level(a2(S, T1), T2) = (if (T1 = T2) then state2 else level(S, T2) fi)
      if (level(S, T1) = state1) .
  eq  bar(a1(S, T1)) = bar(S) .

-- transition rule a3: barriers open
  ceq bar(a3(S, T1)) = open
      if (timeout(S, T1) or (pos(S, T1) = sect2 and not err-comm(S, T1))) .
  ceq level(a3(S, T1), T2) = (if (T1 = T2) then state0 else level(S, T2) fi) .
      if (timeout(S, T1) or (pos(S, T1) = sect2 and not err-comm(S, T1))) .

-- transition rule a4: train passed crossing
  ceq pos(a4(S, T1), T2) = (if (T1 = T2) then sect2 else pos(S, T2) fi)
      if (pos(S, T1) = crossing) .
  eq  level(a4(S, T1), T2) = level(S, T2) .

-- transition rule a5: level crossing accepts 'secure' request
  ceq level(a5(S, T1), T2) = (if (T1 = T2) then state1 else level(S, T2) fi)
      if (pos(S, T1) = sect1 and not err-comm(S, T1) and bar(S) = open) .
  eq  pos(a5(S, T1), T2) = pos(S, T2) .
```

Figure 6.11: Axioms of OTS/CafeOBJ from FTA/OTS

For example, with respect to the conjunct fault event $level(S, T1) = state1 \wedge pos(S, T2) = crossing$, we know that the transition rule τ_5 (or `a5` in CafeOBJ signature) will not change the value of $pos(S, T2)$ based on the constraint of the transition rule, so the following equation can be declared in the axioms of CafeOBJ specification.

```
eq pos(a5(S, T1), T2) = pos(S, T2) .
```

And in a similar way, we can develop all the axioms with respect to the transition rules found in Figure 5.2 (page 83) as shown in Fiugre 6.11.

It should be noted that with respect to the transition rule τ_3 (or `a3` in Figure 6.11), after receiving a 'passed' signal or an occurrence of a timeout event from (or of) a train, the level crossing should be set to an initial state `state0` with respect to the train. However, the state `state0` has not been discovered in Figure 5.2 and Table 5.1, therefore, we should complement `state0` as a constant into the signature of OTS/CafeOBJ (Figure 6.10) with sort `LState`.

6.3.3 Complement Formal Specification for Formal Verification

After the transformation from the events and transition rules of fault trees into the signature and axioms of CafeOBJ specification, we derive the *core* part of the formal system specification (short: core specification) with OTS/CafeOBJ. The transformation itself is straightforward thanks to the common signature and concepts shared by both FTA/OTS and OTS/CafeOBJ. Therefore, we demonstrate and provide an approach from safety analysis to requirements analysis more efficiently and smoothly.

However, the generated core specification may not be complete as for the subsequent important process — formal verification with OTS/CafeOBJ (called proof scores in the sense of OTS/CafeOBJ). Based on our experiences, there are three portions should be complemented as follows.

Firstly, each action defined in the CafeOBJ specification should cover all the observations, i.e., the value of each observation will or will not be affected by the execution of the action. In the last section, we only focus on the transition rule and the corresponding event (conjunct fault event or non-conjunct fault event) to decompose, and the generated axioms (equations of the action) may not cover all the observations defined in the OTS. Therefore, we should complement the axioms for each action. For example, with respect to the action `a5` discussed in the last section, we only analyzed its effect to the observations `level(Sys, Train)` and `pos(Sys, Train)` based on the conjunct fault event $level(S, T1) = state1 \wedge pos(S, T2) = crossing$. To make the specification complete, we should further analyze whether this transition rule will affect the value of the other observations, such as `bar(Sys)`, `err-brake(Sys, Train)`, and so on. The complemented axioms for this transition rule (action `a5`) are declared as follows.

```
ceq level(a5(S, T1), T2) = (if (T1 = T2) then state1 else level(S, T2) fi)
    if (pos(S, T1) = sect1 and not err-comm(S, T1) and bar(S) = open) .
eq  pos(a5(S, T1), T2) = pos(S, T2) .

eq  bar(a5(S, T1)) = close . -- a5 will close the barriers

eq signal-bypass(a5(S, T1), T2) = signal-bypass(S, T2) .
eq err-brake(a5(S, T1), T2) = err-brake(S, T2) .
eq err-comm(a5(S, T1), T2) = err-comm(S, T2) .
eq timeout(a5(S, T1), T2) = timeout(S, T2) .
```

Actually, the last four equations declared above can be added to each action if we classify their corresponding events are *uncontrollable* fault events in the fault trees, that is, the values of these observations will not be affected by any action in the OTS.

Similarly, we can develop the complementary axioms for all the transition rules listed in Figure 6.11 as shown in Figure 6.12.

Secondly, we should specify the initial state for the system and each observation. A constant `init` can be declared to denote any initial state as follows:

```
op init : -> Sys
```

```
-- complementary axioms for a1
  eq level(a1(S, T1), T2) = level(S, T2) .

  eq signal-bypass(a1(S, T1), T2) = signal-bypass(S, T2) .
  eq err-brake(a1(S, T1), T2) = err-brake(S, T2) .
  eq err-comm(a1(S, T1), T2) = err-comm(S, T2) .
  eq timeout(a1(S, T1), T2) = timeout(S, T2) .

-- complementary axioms for a2
  eq pos(a2(S, T1), T2) = pos(S, T2) .

  eq signal-bypass(a2(S, T1), T2) = signal-bypass(S, T2) .
  eq err-brake(a2(S, T1), T2) = err-brake(S, T2) .
  eq err-comm(a2(S, T1), T2) = err-comm(S, T2) .
  eq timeout(a2(S, T1), T2) = timeout(S, T2) .

-- complementary axioms for a3
  ceq pos(a3(S, T1), T2) = (if (T1 = T2) then sect1 else pos(S, T2) fi)
      if (timeout(S, T1) or (pos(S, T1) = sect2 and not err-comm(S, T1))) .
      -- set the state of T1 to its initial state sect1

  eq signal-bypass(a3(S, T1), T2) = signal-bypass(S, T2) .
  eq err-brake(a3(S, T1), T2) = err-brake(S, T2) .
  eq err-comm(a3(S, T1), T2) = err-comm(S, T2) .
  eq timeout(a3(S, T1), T2) = timeout(S, T2) .

-- complementary axioms for a4
  eq bar(a4(S, T1)) = bar(S) .

  eq signal-bypass(a4(S, T1), T2) = signal-bypass(S, T2) .
  eq err-brake(a4(S, T1), T2) = err-brake(S, T2) .
  eq err-comm(a4(S, T1), T2) = err-comm(S, T2) .
  eq timeout(a4(S, T1), T2) = timeout(S, T2) .

-- complementary axioms for a5
  eq  bar(a5(S, T1)) = close .

  eq signal-bypass(a5(S, T1), T2) = signal-bypass(S, T2) .
  eq err-brake(a5(S, T1), T2) = err-brake(S, T2) .
  eq err-comm(a5(S, T1), T2) = err-comm(S, T2) .
  eq timeout(a5(S, T1), T2) = timeout(S, T2) .
```

Figure 6.12: Complementary axioms of transition rules

```
op init : -> Sys

eq bar(init) = open .
eq pos(init, T1) = sect1 .
eq level(init, T1) = state0 .

eq signal-bypass(init, T1) = false .
eq err-brake(init, T1) = false .
eq err-comm(init, T1) = false .
eq timeout(init, T1) = false .
```

Figure 6.13: Initial values of OTS

And the initial value of each observation should also be defined in CafeOBJ in the following form:

eq $\mathtt{o}_i$(init, $\mathtt{X}_{i_1}$, ..., X_{i_m}) = c_0 .

where c_0 means a constant with V (the sort denoting the value of $\mathtt{o}_i$). For example, we can set the initial value of `bar(S)` as `open` (i.e., suppose the initial state of the barriers are open) as follows:

```
eq bar(init) = open .
```

With respect to the uncontrollable fault events found in the fault trees, as we discussed in Chapter 3, we should make the negations of these fault events as safety assumptions for safety analysis and verification. Therefore, in the CafeOBJ specification, we need only to specify that the initial values of the corresponding observations are false (and we have already discussed that the values of these observations will not be affect by any action). Therefore, we can get all the initial states of the observations defined in Figure 6.10 as shown in Fiugure 6.13.

Last, we should check whether the OTS can be *reset*. For example, after the level crossing has got a 'passed' signal from one train, is there an action resetting the level crossing to its initial value (state)? Since the fault trees focus on the discovering of (basic) fault events, such reset issues (transition rules) may be overlooked or not considered in the fault trees. We should check and complement the specification by adding equations or actions if this problem happens. In addition, some other issues may related to whether there exist deadlocks or livelocks in the OTS, but these problems are unrelated with the fault trees, so we do not discuss them here. With respect to our example, since the states the level crossing, train, and barriers can be reset by transition rule `a3`, we can combine the results of Figure 6.10, Figure 6.11, Figure 6.12, and Figure 6.13, and then get a complete formal specification of OTS with CafeOBJ for the fault tree depicted in Figure 5.2.

6.3.4 Combination of FTA/OTS and OTS/CafeOBJ

In the above sections, we discussed and demonstrated how to transform the analyzing results of the fault trees (or called fault tree specification) into the formal system specification with OTS/CafeOBJ. In this section, we make a short summary about the combination of FTA/OTS and OTS/CafeOBJ, and discuss how these two techniques can

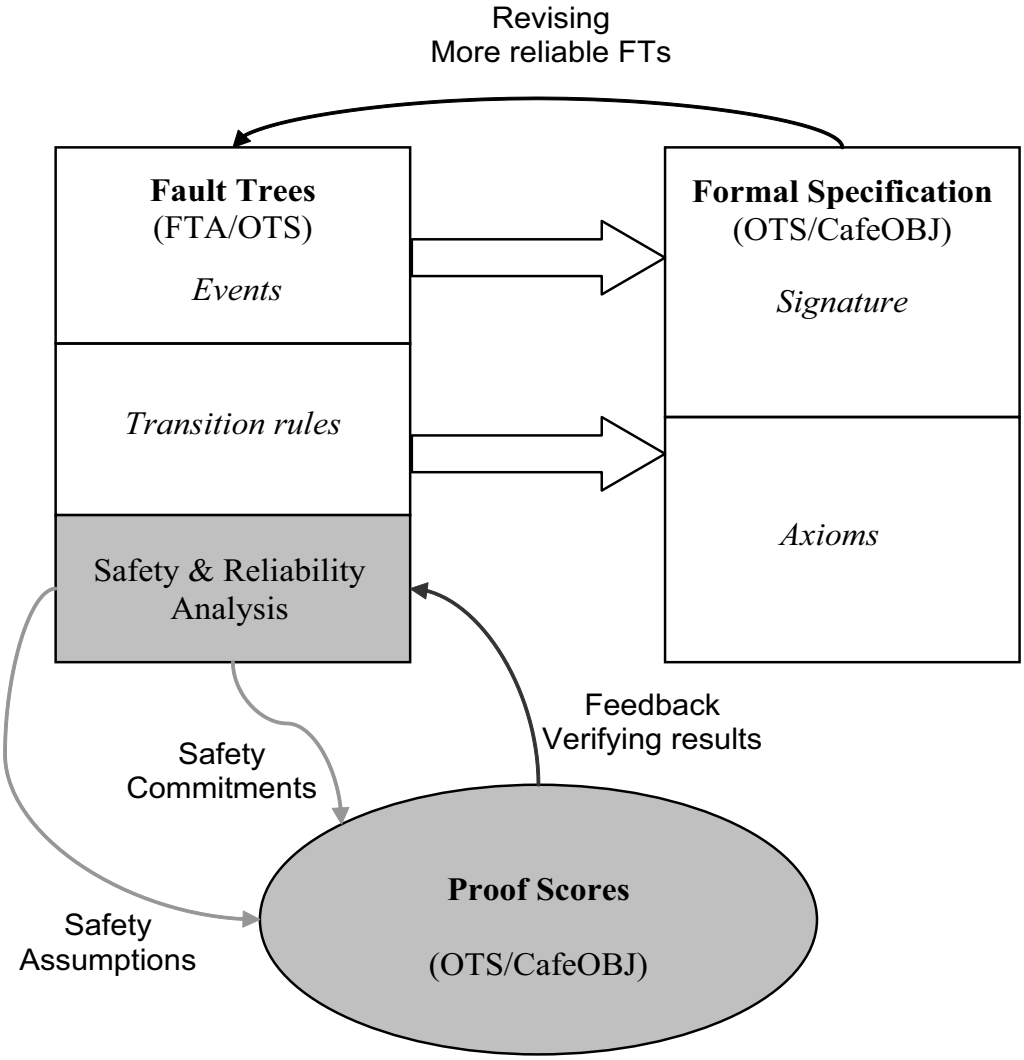


Figure 6.14: Combination of FTA/OTS and OTS/CafeOBJ

work together and complement each other as for more efficient and effective safety and requirements analyses. An overview of the combination of these two techniques is shown in Figure 6.14.

As shown in Figure 6.14, there are several benefits of the combination between the FTA/OTS and OTS/CafeOBJ as follows.

- First of all, since the fault trees are constructed and specified based on the OTS model and signature, it is quite natural and straightforward to transform the fault tree specification into the formal system specification with OTS/CafeOBJ. In other words, we link fault tree analysis to program development in a consistent and smooth way, by requiring that both use the same system model, i.e., OTS model. The significance of this connection and transformation can be further interpreted as follows:
  - By using a common model and signature, it is possible to use the results of the fault tree analysis directly, when specifying and designing the software.

  - It is known that a common framework is important whenever engineers from multiple disciplines need to work together [HR98], therefore a common interpretation to terms used in safety analysis and in requirements formulation is really important and should be provided.
  - It is also possible to formally prove that a system (program) is safe, i.e., that it does not cause the system to violates its safety goals (the negation of the top hazard events of the fault trees), if we can guarantee that all the basic fault events will not happen (or make as safety assumptions with respect to the uncontrollable failures).

- In reverse, after developing formal system specifications (requirements specifications) from the fault trees, we can use the augmented formal requirements specifications to further revise the fault trees, i.e., to check the correctness and consistency of the fault trees, so as to achieve more reliable fault trees. Such kind of feedback and interaction is possible also because of the common model, and thus more precise and reliable safety analysis can be achieved.

- In addition, the results of the fault tree analysis are also useful for the formal verification (proof scores) with OTS/CafeOBJ.
  - On one hand, by using fault tree analysis, we can discover as much as possible uncontrollable basic fault events (failures such as hardware defects, human errors, and physical environment failures), the negations of these failures can be made as a list of safety assumptions with respect to the formal verification of system safety requirements.
  - On the other hand, by using fault tree analysis, we can decompose a complex and abstract system safety goal into some smaller and more manageable subgoals, i.e., concrete safety commitments which usually consist of the negations of the basic conjunct fault events. Such kind of divide-and-conquer method can efficiently decrease the difficulty and complexity of formal verification as for some complex systems and problems.

6.4 Summary

In this chapter, we discussed some temporal logic semantics of fault trees, and presented an analysis of the role of temporal logic in FTA. Based on the analysis, we proposed a simple logic semantics for our FTA/OTS based on classical propositional logic and basic concepts of observational transition systems. Then we focused on how to link fault tree analysis to program development (system requirements specifications), demonstrated how to transform the fault tree specifications into formal system specifications with OTS/CafeOBJ. An analysis of the combination of the FTA/OTS and OTS/CafeOBJ was concluded in Section 6.3.4.

To our understanding, to ensure the correctness of the fault trees, providing an understandable formal construction model as well as simple logic semantics is more efficient and better than the approach that first build the fault trees by intuition (in an informal way), and then verify its correctness afterwards by some complex logic semantics defined

for the fault trees. This is the main motivation why we further refine our formal fault tree model from temporal logic to the OTS model. Another motivation is inspired by Hansan's work [HR98], that is, trying to link the fault tree analysis to program development by providing a common system model for both of them.

Hansan's work contributes an approach from safety analysis (with FTA) to Software requirements based on duration calculus and interval temporal logic (ITL) for the interpretation of both fault trees and software requirements [HR98]. However, it does not discuss how to ensure the correctness of the fault trees and how to derive the concrete formal system specifications from the fault trees.

Another related work about the formal semantics and specifications of fault trees can be found in [CSD00] (D. Coppoit, K. J. Sullivan and J. B. Dugan, 2000), which presents an approach to write the formal specification of dynamic fault trees with Z specification language. However, as we known, Z itself is not an executable formal language, which limits its usage if we want to prove some safety properties based on the formal system (or fault tree) specifications. In addition, Coppoit et al. do not consider the decomposition problem with respect to the conjunct fault events because their attention is focused on the formal semantics of dynamic fault trees.

Some other formal works on fault trees mainly focus on the formal semantics (usually with temporal logic) of fault trees, such as [STR02, BA93], while as far as we know, none of them focus on how to connect the fault tree analysis with program development. And even with the temporal semantics of fault trees, some issues are still disputed as we discussed in Section 6.1.

In a nutshell, thanks to the advantages and properties of OTS/CafeOBJ, our work presents an approach from safety analysis to program development (requirements specifications and verifications), which makes engineers and designers from multiple disciplines work together more efficiently and effectively, and makes it possible to achieve more reliable and efficient safety and requirements analysis.

Chapter 7

Conclusions

7.1 Contributions

This dissertation has developed techniques to achieve more reliable formal fault tree analysis as well as more efficient requirements analysis. Three important issues have been addressed, that is, the correctness of the fault trees, the combination of fault tree analysis and formal methods with CafeOBJ and Maude, and the transformation from the results of fault tree analysis into formal system specifications with OTS/CafeOBJ by using the common framework of OTS. The technical contributions of this dissertation can be concluded as follows.

1. It identifies decomposition of fault events as a core issue that guarantees the correctness of the fault trees, i.e., the sub events must formally result in their top event through the given logic gate. And for solution, it proposes two formal fault tree construction models based on temporal logic and observational transition systems, respectively. The advantages of the formal fault tree construction models proposed in this dissertation are as follows:

    - It (both of the two models) is a heuristic and deductive method to build fault trees more precisely and effectively;
    - The correctness of the generated fault trees is proved by the construction process itself, rather than requiring a stand-alone verification, and thus avoiding the problems that often arise with traditional methods. At the same time, it gives designers and domain experts the ability to discover domain (transition) rules and previously unnoticed design deficiencies during the construction process;
    - It integrates (records) the domain (transition) rules into the fault trees, which makes the analysis more complete and is useful for revising and rechecking of the fault trees;
    - The formal construction model based on classical propositional logic and basic concepts of OTS provides an alternative for engineers who are not familiar with temporal logic, and thus it complements the model of temporal logic one for different applications.

2. It presents a study on the combination of fault tree analysis and formal methods with CafeOBJ and Maude.

   - It proposes an approach to derive concrete requirements (safety assumptions and commitments) from FTA so as to guild and assist the subsequent system design and verification;
   - It demonstrates how to write fault tree specification and realize automatic calculation of minimal cut sets of fault trees with term rewriting system (TRS) of CafeOBJ;
   - It demonstrates how to formally model, specify, and verify OTSs with CafeOBJ based on the analyzing results of FTA. And as a complement of theorem proving technique provided by CafeOBJ, it also demonstrates how to model-check OTSs with Maude (a sibling language of CafeOBJ).

3. Most important, it further proposes a novel formal fault tree analysis based on OTS. The point is that, by using a common framework of OTS, it is possible to use the results of fault tree analysis directly, when specifying and verifying the system with OTS/CafeOBJ. The transformation from fault tree specifications into formal system specifications with OTS/CafeOBJ is demonstrated. Therefore, it presents an approach to link safety analysis and requirements analysis in a consistent and smooth way, and it provides a common framework for engineers and designers from multiple disciplines to work together more efficiently and effectively.

7.2 Future Directions

Due to the complexity of acquiring reliable and precise safety analysis and requirements specifications, it is not a trivial task to develop a common framework that can support and link both of these two important processes in program (system) development. In this dissertation, we mainly focus on developing techniques to achieve both *reliability* and *efficiency* for requirements engineering, by combining fault tree analysis and formal methods with CafeOBJ. And in our future work, we will further improve our research by focusing on the following aspects:

- Although we have demonstrated that the transformation from the results of fault trees into the formal CafeOBJ specifications is straightforward, by using the common framework of OTS, it still may be a hard work for engineers who are not familiar with formal languages. To this end, we are trying to develop some templates which can help the engineers derive formal system specifications more easily and efficiently. And a longer-term goal is to develop a semi-automatic environment for the interaction of FTA and OTS/CafeOBJ.

- Fault tree analysis can also be used in verifying formal specifications (or programs) itself in addition to system safety and reliability analysis, some typical cases are using software fault trees to verify Ada programs [MJC+99, Lev91] and SOFL formal specifications [Liu00]. Inspired by these works, we are trying to carry out some studies on verifying CafeOBJ specifications using FTA.

- As we know, a hard task of formal verification with theorem proving technique of CafeOBJ is finding lemmas by hand. Generally speaking, it is a time-consuming and intellectual hard work, especially in complex system analysis. To attack this problem, an automatic theorem proving tool of CafeOBJ is in developing by some other members of our group (this is also one of the motivations why we study how to model-check OTSs with Maude as for finite state systems in this dissertation). In our experiences, fault tree analysis can also be used to assist in finding lemmas and their relations at a lower abstraction level, given a complete and consistent formal specification of the system. This study is related with the above one, i.e., verifying CafeOBJ specifications using fault tree analysis.

- We realize that our method has not yet been applied in full scale industrial practice, although it has been demonstrated by one case in this dissertation. Our next step is trying to find more (big scale) case studies to further improve our work.

- In addition, since this dissertation covers only the main research of our work, some of other studies are not included, such as knowledge management, software reliability allocation using fault tree analysis, and data mining (see Publications at the end of the dissertation). How to improve these studies and find some new interdisciplinary topics is also one of our future works.

Appendix A

Proof Scores of CafeOBJ

This appendix includes the complete CafeOBJ codes (proof score) with respect to the crossing control system discussed in Section 4.2. The whole proof score consists of four important CafeOBJ files (with the suffix ".mod") as follows.

TrainCrossing.mod : The specification of the railway crossing control system.

TCInvariants.mod : Definition of the invariants (safety property and corresponding lemmas) to prove.

TCIcases.mod : Case splitting for induction based on the effective conditions of each transition rule defined in the specification of the crossing control system (TrainCrossing.mod).

TCproof100.mod : Proof of INV100, i.e., the safety property we want to prove. We do not list the other 13 proof files of the corresponding lemmas, since they can be easily derived and written in a similar way.

A.1 TrainCrossing.mod

```
-- A Radio-based crossing control system is informally described as follows.

-- Shortly before the train arrives at the 'latest braking point' (latest point at
-- which it is possible for the train to stop before the crossing), it sends a
-- 'secure' signal to the level crossing in order to check the status of the level
-- crossing.

-- When the level crossing receives the command 'secure', it instructs the barriers
-- to close. After the barriers have been closed, the level crossing is safe for
-- a certain period of time and a 'release' signal is sent to the train, which
-- indicates that the train may pass the crossing; Otherwise the train should stop
-- until it gets the 'release' signal.

-- When the train has passed the crossing, it sends a 'passed' signal to the
-- crossing, which allows the crossing to open the barriers and switch back to
-- its initial state.

-- Based on the above informal description, a formal Modeling and specifying of
```

```
-- this crossing control system can be made as follows.

-- The states of the train are sect0, sect1, critical, and sect2.
-- sect0: the train is approaching near enough to the 'latest braking point'
--   and sends 'secure' signal to the level crossing;
-- sect1: just after the train sent the 'secure' signal and until it gets the
--   permission ('release' signal) to enter its critical section from the level
--   crossing;
-- critical: the train is passing over the level crossing;
-- sect2: the train leaves the level crossing and sends 'passed' signal to
-- the level crossing.

-- The states of the level crossing are state0, state1 and state2.
-- state0: the level crossing is waiting for a 'secure' signal from a train;
-- state1: after getting the 'secure' signal and until the level crossing responds
--   a 'release' signal to the train;
-- state2: after the response to the train and until the level crossing gets
--   the 'passed' signal from the train.
-- after getting the 'passed' signal, the level crossing closes the barriers,
--   and then enters idling state waiting for the next train.

-- There are two status of the barriers: open and close, initially set to open.
-- Notice here we look the barriers as an integrated part of the leveling crossing,
-- in other words, we ignore the the communication as well as its time delay between
-- them since it is trivial in this analysis.

-- A Boolean variable is defined to solve the mutual exclusion problem as follows:
-- Boolean idling = true;
-- idling is shared by all trains (processes), initially set to true, which means
-- that the level crossing is in state0 and the barriers are opened.

-- Six transition rules are used below.
--
-- * tr-send-secure: The train sends 'secure' signal to the level crossing.
--   - effective conditions: the state of the train is sect0
--   - results: the state of the train gets sect1

-- * le-get-secure: The level crossing gets 'secure' signal from the train.
--   - effective conditions: the state of the train is sect1.
--   - results: if idling = true then
--                idling = false;
--                barriers = close;
--                the state of the level crossing gets state1
--              else
--                any values are not changed
--              fi

-- * le-send-release: the level crossing sends the 'release' signal to the train.
--   - effective conditions: the state of the level crossing is state1
--   - results: the state of the level crossing gets state2;

-- * tr-get-release: the train gets the 'release' signal from the level crossing
--   and enters its critical section.
--   - effective conditions: the state of the level crossing is state2
--   - results: the state of the train gets critical;

-- * tr-send-passed: The train leaves the level crossing and sends the 'passed'
```

```
--   signal to the level crossing.
--   - effective conditions: the state of the train is critical.
--   - results: the state of the train gets sect2.

-- * le-get-passed: The level crossing gets the 'passed' signal
--   - effective conditions: the state of the train is sect2.
--   - results: (1) barriers = open;
--              (2) the state of the level crossing gets state0;
--              (3) idling = true


-- ************************************
-- Formal Specification begins

mod! TRAIN {
  [Train]
  op _=_ : Train Train -> Bool {comm}
  var T : Train
  eq (T = T) = true .
}

mod! TSTATE{
  [TState] -- sort of Train state
  ops sect0 sect1 critical sect2 : -> TState
  op _=_ : TState TState -> Bool {comm}
  var TS : TState
  eq (TS = TS) = true .
  eq (sect0 = sect1) = false .
  eq (sect0 = sect2) = false .
  eq (sect0 = critical) = false .
  eq (critical = sect1) = false .
  eq (critical = sect2) = false .
  eq (sect1 = sect2) = false .
}

mod! LSTATE{
  [LState] -- sort of Level Crossing state
  ops state0 state1 state2 : -> LState
  op _=_ : LState LState -> Bool {comm}
  var LS : LState
  eq (LS = LS) = true .
  eq (state0 = state1) = false .
  eq (state0 = state2) = false .
  eq (state1 = state2) = false .
}

mod! BSTATE{
  [BState] -- sort of Barriers state
  ops open close : -> BState
  op _=_ : BState BState -> Bool {comm}
  var BS : BState
  eq (BS = BS) = true .
  eq (open = close) = false .
}
--
-- The formal model is described in module TC.
--
```

```
-- - Hidden sort ‘System’ denotes the state space.
-- - Constant ‘init’ denotes any initial state.
-- - Operator ‘idling’ denotes variable idling

-- module TrainCrossing (TC) System
mod* TC {
  pr(TRAIN + TSTATE + LSTATE + BSTATE)
  *[System]*
-- any initial state
  op init : -> System

-- observations
  bop idling : System -> Bool
  bop t : System Train -> TState
  bop l : System Train -> LState
  -- bop l : System -> LState
  bop b : System -> BState

-- actions (transition rules)
  bop tr-send-secure  : System Train -> System
  bop le-get-secure   : System Train -> System
  bop le-send-release : System Train -> System
  bop tr-get-release  : System Train -> System
  bop tr-send-passed  : System Train -> System
  bop le-get-passed   : System Train -> System

-- CafeOBJ variables
  var S : System
  vars T1 T2 : Train

-- for any initial state
  eq idling(init) = true .
  eq t(init, T1) = sect0 . -- any train initially set sect0
  eq l(init, T1) = state0 . -- level crossing initially set state0
  -- eq l(init) = state0 . -- level crossing initially set state0
  eq b(init) = open . -- barriers initially set open

-- * traisition rules
-- * (1) tr-send-secure
  ceq t(tr-send-secure(S, T1), T2) = (if (T1 = T2) then sect1 else t(S, T2) fi)
      if t(S, T1) = sect0 .
  eq l(tr-send-secure(S, T1), T2) = (if (T1 = T2) then l(S, T1) else l(S, T2) fi) .
  eq b(tr-send-secure(S, T1)) = b(S) .
  eq idling(tr-send-secure(S, T1)) = idling(S) .
  ceq tr-send-secure(S, T1) = S if not (t(S, T1) = sect0) .

-- * (2) le-get-secure
  ceq idling(le-get-secure(S, T1)) = false if t(S, T1) = sect1 and idling(S) .
  ceq b(le-get-secure(S, T1)) = close if t(S, T1) = sect1 and idling(S) .
  ceq l(le-get-secure(S, T1), T2) = (if (T1 = T2) then state1 else l(S, T2) fi)
      if t(S, T1) = sect1 and idling(S) .
  eq t(le-get-secure(S, T1), T2) = (if (T1 = T2) then t(S, T1) else t(S, T2) fi) .
  ceq le-get-secure(S, T1) = S if not (t(S, T1) = sect1) or not idling(S) .

-- * (3) le-send-release
  ceq l(le-send-release(S, T1), T2) = (if (T1 = T2) then state2 else l(S, T2) fi)
      if l(S, T1) = state1 .
```

```
  eq b(le-send-release(S, T1)) = b(S) .
  eq t(le-send-release(S, T1), T2) = (if (T1 = T2) then t(S, T1) else t(S, T2) fi) .
  eq idling(le-send-release(S, T1)) = idling(S) .
  ceq le-send-release(S, T1) = S if not (l(S, T1) = state1) .

-- * (4) tr-get-release
  ceq t(tr-get-release(S, T1), T2) = (if (T1 = T2) then critical else t(S, T2) fi)
      if (l(S, T1) = state2) .
  eq l(tr-get-release(S, T1), T2) = (if (T1 = T2) then l(S, T1) else l(S, T2) fi) .
  eq b(tr-get-release(S, T1)) = b(S) .
  eq idling(tr-get-release(S, T1)) = idling(S) .
  ceq tr-get-release(S, T1) = S if not (l(S, T1) = state2) .

-- * (5) tr-send-passed
  ceq t(tr-send-passed(S, T1), T2) = (if (T1 = T2) then sect2 else t(S, T2) fi)
      if t(S, T1) = critical .
  eq l(tr-send-passed(S, T1), T2) = (if (T1 = T2) then l(S, T1) else l(S, T2) fi) .
  eq b(tr-send-passed(S, T1)) = b(S) .
  eq idling(tr-send-passed(S, T1)) = idling(S) .
  ceq tr-send-passed(S, T1) = S if not (t(S, T1) = critical) .

-- * (6) le-get-passed
  ceq b(le-get-passed(S, T1)) = open if t(S, T1) = sect2 .
  ceq idling(le-get-passed(S, T1)) = true if t(S, T1) = sect2 .
  ceq l(le-get-passed(S, T1), T2) = (if (T1 = T2) then state0 else l(S, T2) fi)
     if t(S, T1) = sect2 .
  ceq t(le-get-passed(S, T1), T2) = (if (T1 = T2) then sect0 else t(S, T2) fi)
      if t(S, T1) = sect2 .
  ceq le-get-passed(S, T1) = S if not (t(S, T1) = sect2) .

-- Formal Specification ends
}
```

A.2 TCInvariants.mod

```
-- Based on the analysis result of our fault tree, the safety properties
-- that we want to prove are as follows.
-- modified by XJW040325

-- Claim 100: if the level crossing has sent a 'release' signal, then the
-- state of the system should not be idling.
-- Here, we need to focus on the result of the transition rule le-send-release,
-- i.e. l(S, T1) = state2
-- invariant (l(S, T1) = state2) implies not (idling(S))

-- Claim 200: if the level crossing has sent a 'release' signal, then the system
-- could not get a 'passed' signal at this moment.
-- Here, let's focus on the results of the transition rules le-send-release and
-- le-get-passed, i.e. l(S, T1) = state1 and l(S, T2) = state0, respectively. We can
-- derive our invariant as follows.
-- invariant (l(S, T1) = state2) implies not (l(S, T2) = state0)

-- * Moreover, if we consider another consequence of the transition rule
-- le-get-release, i.e., idling(S) = true, then inv200 is equal to inv100 in this
```

```
-- sense.

mod INV {
  pr(TC)
-- arbirtrary trains
  ops p q : -> Train

  op inv100 : System Train       -> Bool  -- need inv310, inv320

  op inv310 : System Train       -> Bool  -- need inv330
  op inv320 : System Train Train -> Bool  -- need inv340, inv330
  op inv330 : System Train Train -> Bool  -- need inv360, inv420
  op inv340 : System Train Train -> Bool  -- need inv350, inv360
  op inv350 : System Train Train -> Bool  -- need inv370
  op inv360 : System Train Train -> Bool  -- need inv370, inv380
  op inv370 : System Train Train -> Bool  -- need inv390, inv100
  op inv380 : System Train       -> Bool  -- need inv100, inv400
  op inv390 : System Train Train -> Bool  -- need inv310
  op inv400 : System Train Train -> Bool  -- need inv320, inv410
  op inv410 : System Train Train -> Bool  -- need inv340
  op inv420 : System Train       -> Bool  -- need inv380, inv430
  op inv430 : System Train Train -> Bool  -- need inv400

-- CafeOBJ variables
  var  S     : System
  vars T1 T2 : Train

-- define invariants to prove
  eq inv100(S, T1) = (l(S, T1) = state2) implies not idling(S) .

  eq inv310(S, T1) = (l(S, T1) = state1) implies not idling(S) .
  eq inv320(S, T1, T2) = (t(S, T1) = sect2) and (l(S, T2) = state2)
     implies (T1 = T2) .
  eq inv330(S, T1, T2) = (t(S, T1) = sect2) implies not (l(S, T2) = state1) .
  eq inv340(S, T1, T2) = (l(S, T1) = state2) and (t(S, T2) = critical)
     implies (T1 = T2) .
  eq inv350(S, T1, T2) = (l(S, T1) = state2) and (l(S, T2) = state2)
     implies (T1 = T2) .
  eq inv360(S, T1, T2) = (l(S, T1) = state1) implies not (t(S, T2) = critical) .
  eq inv370(S, T1, T2) = (l(S, T1) = state1) implies not (l(S, T2) = state2) .
   eq inv380(S, T1) = (t(S, T1) = critical) implies not idling(S) .
  eq inv390(S, T1, T2) = (l(S, T1) = state1) and (l(S, T2) = state1)
     implies (T1 = T2) .
  eq inv400(S, T1, T2) = (t(S, T1) = sect2) implies not (t(S, T2) = critical) .
  eq inv410(S, T1, T2) = (t(S, T1) = critical) and (t(S, T2) = critical)
     implies (T1 = T2) .
  eq inv420(S, T1) = (t(S, T1) = sect2) implies not idling(S) .
  eq inv430(S, T1, T2) = (t(S, T1) = sect2) and (t(S, T2) = sect2)
     implies (T1 = T2) .

}

-- Invariants are proved by INDUCTION on the numbers of transition rules
-- applied or executed.
--
-- If we prove that the system has invariant P by induction on the number of
-- transition rules applied, the proof structure looks like this:
```

```
--
--   I) Base case
--     P(init)
--   II) Inductive cases
--     1. P(s) implies P(tr-send-secure(s)) for any s.
--     2. P(s) implies P(le-get-secure(s)) for any s.
--     3. P(s) implies P(le-send-release(s)) for any s.
--     4. P(s) implies P(tr-get-release(s)) for any s.
--     5. P(s) implies P(tr-send-passed(s)) for any s.
--     6. P(s) implies P(le-get-passed(s)) for any s.
--
-- Predicates to prove in each inductive step are defined as CafeOBJ terms in
-- module ISTEP.
--
mod ISTEP {
  pr(INV)
-- arbitrary System states
  ops s s' : -> System
-- declare predicates to prove in inductive step
  op  istep100                   : Train -> Bool
  ops istep310 istep380 istep420 : Train -> Bool
  ops istep320 istep330 istep340 istep350 istep360 istep370 istep390
      istep400 istep410 istep430 : Train Train -> Bool

-- CafeOBJ variables
  vars T1 T2 : Train
-- define predicates to prove in inductive step
  eq istep100(T1) = inv100(s, T1) implies inv100(s', T1) .

-- lemmas should be proved for the above predicates
  eq istep310(T1)     = inv310(s, T1)     implies inv310(s', T1)     .
  eq istep320(T1, T2) = inv320(s, T1, T2) implies inv320(s', T1, T2) .
  eq istep330(T1, T2) = inv330(s, T1, T2) implies inv330(s', T1, T2) .
  eq istep340(T1, T2) = inv340(s, T1, T2) implies inv340(s', T1, T2) .
  eq istep350(T1, T2) = inv350(s, T1, T2) implies inv350(s', T1, T2) .
  eq istep360(T1, T2) = inv360(s, T1, T2) implies inv360(s', T1, T2) .
  eq istep370(T1, T2) = inv370(s, T1, T2) implies inv370(s', T1, T2) .
  eq istep380(T1)     = inv380(s, T1)     implies inv380(s', T1)     .
  eq istep390(T1, T2) = inv390(s, T1, T2) implies inv390(s', T1, T2) .
  eq istep400(T1, T2) = inv400(s, T1, T2) implies inv400(s', T1, T2) .
  eq istep410(T1, T2) = inv410(s, T1, T2) implies inv410(s', T1, T2) .
  eq istep420(T1)     = inv420(s, T1)     implies inv420(s', T1)     .
  eq istep430(T1, T2) = inv430(s, T1, T2) implies inv430(s', T1, T2) .
}
```

A.3 TCIcases.mod

```
-- modified by XJW040323
-- This file is to specify the s and s' for induction based on the
-- effective conditions of each transition rule.

mod ICASE1 {
  pr(ISTEP)
-- arbitrary trains
```

```
  op t10 : -> Train
-- assumptions for tr-send-secure
  eq t(s, t10) = sect0 .
-- successor state
  eq s' = tr-send-secure(s, t10) .
}

mod nonICASE1 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect0) = false .
-- successor state
  eq s' = tr-send-secure(s, t10) .
}

mod ICASE2 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions for le-get-secure
  eq t(s, t10) = sect1 .
  eq idling(s) = true .
-- successor state
  eq s' = le-get-secure(s, t10) .
}

mod nonICASE2-1 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect1) = false .
  eq idling(s) = true .
-- successor state
  eq s' = le-get-secure(s, t10) .
}

mod nonICASE2-2 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect1) = false .
  eq idling(s) = false .
-- successor state
  eq s' = le-get-secure(s, t10) .
}

mod nonICASE2-3 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect1) = true .
  eq idling(s) = false .
```

```
-- successor state
  eq s' = le-get-secure(s, t10) .
}

mod ICASE3 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions for le-send-release
  -- eq l(s) = state1 .
  eq l(s, t10) = state1 .
-- successor state
  eq s' = le-send-release(s, t10) .
}

mod nonICASE3 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  -- eq (l(s) = state1) = false .
  eq (l(s, t10) = state1) = false .
-- successor state
  eq s' = le-send-release(s, t10) .
}

mod ICASE4 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions for tr-get-release
  -- eq l(s) = state2 .
  eq l(s, t10) = state2 .
-- successor state
  eq s' = tr-get-release(s, t10) .
}

mod nonICASE4 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  -- eq (l(s) = state2) = false .
  eq (l(s, t10) = state2) = false .
-- successor state
  eq s' = tr-get-release(s, t10) .
}

mod ICASE5 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions for tr-send-passed
  eq t(s, t10) = critical .
-- successor state
  eq s' = tr-send-passed(s, t10) .
}
```

```
mod nonICASE5 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = critical) = false .
-- successor state
  eq s' = tr-send-passed(s, t10) .
}

mod ICASE6 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions for le-get-passed
  eq t(s, t10) = sect2 .
-- successor state
  eq s' = le-get-passed(s, t10) .
}

mod nonICASE6 {
  pr(ISTEP)
-- arbitrary trains
  op t10 : -> Train
-- assumptions
  eq (t(s, t10) = sect2) = false .
-- successor state
  eq s' = le-get-passed(s, t10) .
}
```

A.4 TCProof100.mod

```
-- modified by XJW040324
-- Case Analyses

-- In a proof score, we use constants to denote arbitrary objects.
--
-- Invariants and actions usually have arguments other than one denoting states
-- such as p, q, and t10 of inv200(s,p,q) and tr-send-secure(s,t10).
-- We have to show invariants for not only any state but also any such
-- arguments. Constants are usually used to denote arbitrary objects
-- such as states and processes.
--
-- Constants, such as s, denoting arbitrary states are declared in module ISTEP.
-- Constants, such as p and q, denoting arbitrary trains are declared in module INV.
-- Constants, such as t10, denoting arbitrary trains are declared in ICASE module.


-- 1) tr-send-secure(s,t10)
mod EICASE1 { pr(ICASE1)
-- Basic predicates
  op bp1 : -> Bool
  #define bp1 ::= (p = t10) .
```

```
-- Exhaustiveness check
  op check : -> Bool
  eq check = bp1 or not bp1 .
}

-- Case splitting
mod ICASE1-1 { pr(EICASE1)
  eq p = t10 .
}
--
mod ICASE1-2 { pr(EICASE1)
  eq bp1 = false .
}

-- 2) le-get-secure(s,t10)
mod EICASE2 { pr(ICASE2)
-- Basic predicates
  op bp1 : -> Bool
  #define bp1 ::= (p = t10) .
-- Exhaustiveness check
  op check : -> Bool
  eq check = bp1 or not bp1 .
}

-- Case splitting
mod ICASE2-1 { pr(EICASE2)
  eq bp1 = true .
}
--
mod ICASE2-2 { pr(EICASE2)
  eq bp1 = false .
}

-- 3) le-send-release(s,t10)
mod EICASE3 { pr(ICASE3)
-- Basic predicates
  op bp1 : -> Bool
  #define bp1 ::= (p = t10) .
-- Exhaustiveness check
  op check : -> Bool
  eq check = bp1 or not bp1 .
}

-- Case splitting
mod ICASE3-1 { pr(EICASE3)
  eq p = t10 .
}
--
mod ICASE3-2 { pr(EICASE3)
  eq bp1 = false .
}

-- 4) tr-get-release(s,t10)
mod EICASE4 { pr(ICASE4)
-- Basic predicates
  op bp1 : -> Bool
  #define bp1 ::= (p = t10) .
```

```
-- Exhaustiveness check
  op check : -> Bool
  eq check = bp1 or not bp1 .
}

-- Case splitting
mod ICASE4-1 { pr(EICASE4)
  eq p = t10 .
}
--
mod ICASE4-2 { pr(EICASE4)
  eq bp1 = false .
}

-- 5) tr-send-passed(s,t10)
mod EICASE5 { pr(ICASE5)
-- Basic predicates
  op bp1 : -> Bool
  #define bp1 ::= (p = t10) .
-- Exhaustiveness check
  op check : -> Bool
  eq check = bp1 or not bp1 .
}

-- Case splitting
mod ICASE5-1 { pr(EICASE5)
  eq p = t10 .
}
--
mod ICASE5-2 { pr(EICASE5)
  eq bp1 = false .
}

-- 6) le-send-release(s,t10)
mod EICASE6 { pr(ICASE6)
-- Basic predicates
  op bp1 : -> Bool
  #define bp1 ::= (p = t10) .
-- Exhaustiveness check
  op check : -> Bool
  eq check = bp1 or not bp1 .
}

-- Case splitting
mod ICASE6-1 { pr(EICASE6)
  eq bp1 = true .
}
--
mod ICASE6-2 { pr(EICASE6)
  eq bp1 = false .
}

-- Exhaustiveness Check
  red in EICASE1 : check .
  red in EICASE2 : check .
  red in EICASE3 : check .
  red in EICASE4 : check .
```

```
  red in EICASE5 : check .
  red in EICASE6 : check .
-- End of Exhaustiveness Check

-- Proof
-- I) Base case
  red in INV : inv100(init,p) .
-- II) Inductive cases
  red in ICASE1-1    : istep100(p) .
  red in ICASE1-2    : istep100(p) .
  red in nonICASE1   : istep100(p) .
  red in ICASE2-1    : istep100(p) .
  red in ICASE2-2    : istep100(p) .
  red in nonICASE2-1 : istep100(p) .
  red in nonICASE2-2 : istep100(p) .
  red in nonICASE2-3 : istep100(p) .
  -- red in ICASE3-1 : istep100(p) .
  red in ICASE3-1    : inv310(s,p) implies istep100(p) .
  red in ICASE3-2    : istep100(p) .
  red in nonICASE3   : istep100(p) .
  red in ICASE4-1    : istep100(p) .
  red in ICASE4-2    : istep100(p) .
  red in nonICASE4   : istep100(p) .
  red in ICASE5-1    : istep100(p) .
  red in ICASE5-2    : istep100(p) .
  red in nonICASE5   : istep100(p) .
  red in ICASE6-1    : istep100(p) .
  -- red in ICASE6-2 : istep100(p) .
  red in ICASE6-2    : inv320(s,t10,p) implies istep100(p) .
  red in nonICASE6   : istep100(p) .
```

A.5 Experimental Result

In this section, we present the experimental result of the proof of INV100. The results of the proof of other invariants are not listed here because of the same consideration, i.e., the proofs of those invariants are similar to the proof of INV100.

To execute the proof is very simply, just using a command "input" (or "in" for short) of CafeOBJ to read the above four files sequently, and the result is shown below.

```
CafeOBJ> in TrainCrossing.mod
processing input : d:\xjw\cafeobj\train-crossing\phd\TrainCrossing.mod
-- defining module! TRAIN..._.* done.
-- defining module! TSTATE....._.....* done.
-- defining module! LSTATE....._....* done.
-- defining module! BSTATE....._..* done.
-- defining module* TC_*............._.................................*
** system failed to prove =*= is a congruence of TC done.
CafeOBJ> in TCInvariants.mod
processing input : d:\xjw\cafeobj\train-crossing\phd\TCInvariants.mod
-- defining module INV................._............* done.
-- defining module ISTEP................._............* done.
CafeOBJ> in TCIcases.mod
```

```
processing input : d:\xjw\cafeobj\train-crossing\phd\TCIcases.mod
-- defining module ICASE1..._..* done.
-- defining module nonICASE1..._..* done.
-- defining module ICASE2..._...* done.
-- defining module nonICASE2-1..._...* done.
-- defining module nonICASE2-2..._...* done.
-- defining module nonICASE2-3..._...* done.
-- defining module ICASE3..._..* done.
-- defining module nonICASE3..._..* done.
-- defining module ICASE4..._..* done.
-- defining module nonICASE4..._..* done.
-- defining module ICASE5..._..* done.
-- defining module nonICASE5..._..* done.
-- defining module ICASE6..._..* done.
-- defining module nonICASE6..._..* done.
CafeOBJ> in TCProof100.mod

processing input : d:\xjw\cafeobj\train-crossing\phd\TCProof100.mod
-- defining module EICASE1..._..._.* done.
-- defining module ICASE1-1._.* done.
-- defining module ICASE1-2._.* done.
-- defining module EICASE2..._..._.* done.
-- defining module ICASE2-1._.* done.
-- defining module ICASE2-2._.* done.
-- defining module EICASE3..._..._.* done.
-- defining module ICASE3-1._.* done.
-- defining module ICASE3-2._.* done.
-- defining module EICASE4..._..._.* done.
-- defining module ICASE4-1._.* done.
-- defining module ICASE4-2._.* done.
-- defining module EICASE5..._..._.* done.
-- defining module ICASE5-1._.* done.
-- defining module ICASE5-2._.* done.
-- defining module EICASE6..._..._.* done.
-- defining module ICASE6-1._.* done.
-- defining module ICASE6-2._.* done.
-- reduce in EICASE1 : check
true : Bool
(0.000 sec for parse, 10 rewrites(0.000 sec), 38 matches)
-- reduce in EICASE2 : check
true : Bool
(0.000 sec for parse, 10 rewrites(0.000 sec), 38 matches)
-- reduce in EICASE3 : check
true : Bool
(0.000 sec for parse, 10 rewrites(0.000 sec), 38 matches)
-- reduce in EICASE4 : check
true : Bool
(0.000 sec for parse, 10 rewrites(0.000 sec), 38 matches)
-- reduce in EICASE5 : check
true : Bool
(0.000 sec for parse, 10 rewrites(0.016 sec), 38 matches)
-- reduce in EICASE6 : check
true : Bool
(0.000 sec for parse, 10 rewrites(0.000 sec), 38 matches)

-- reduce in INV : inv100(init,p)
```

```
true : Bool
(0.000 sec for parse, 10 rewrites(0.000 sec), 15 matches)
-- reduce in ICASE1-1 : istep100(p)
true : Bool
(0.015 sec for parse, 33 rewrites(0.000 sec), 158 matches)
-- reduce in ICASE1-2 : istep100(p)
true : Bool
(0.000 sec for parse, 32 rewrites(0.000 sec), 158 matches)
-- reduce in nonICASE1 : istep100(p)
true : Bool
(0.000 sec for parse, 24 rewrites(0.016 sec), 159 matches)
-- reduce in ICASE2-1 : istep100(p)
true : Bool
(0.000 sec for parse, 49 rewrites(0.000 sec), 99 matches)
-- reduce in ICASE2-2 : istep100(p)
true : Bool
(0.000 sec for parse, 48 rewrites(0.000 sec), 116 matches)
-- reduce in nonICASE2-1 : istep100(p)
true : Bool
(0.000 sec for parse, 28 rewrites(0.000 sec), 101 matches)
-- reduce in nonICASE2-2 : istep100(p)
true : Bool
(0.000 sec for parse, 30 rewrites(0.000 sec), 103 matches)
-- reduce in nonICASE2-3 : istep100(p)
true : Bool
(0.000 sec for parse, 30 rewrites(0.000 sec), 103 matches)
-- reduce in ICASE3-1 : inv310(s,p) implies istep100(p)
true : Bool
(0.000 sec for parse, 49 rewrites(0.000 sec), 133 matches)
-- reduce in ICASE3-2 : istep100(p)
true : Bool
(0.000 sec for parse, 34 rewrites(0.000 sec), 153 matches)
-- reduce in nonICASE3 : istep100(p)
true : Bool
(0.000 sec for parse, 24 rewrites(0.000 sec), 161 matches)
-- reduce in ICASE4-1 : istep100(p)
true : Bool
(0.000 sec for parse, 35 rewrites(0.000 sec), 95 matches)
-- reduce in ICASE4-2 : istep100(p)
true : Bool
(0.000 sec for parse, 32 rewrites(0.000 sec), 153 matches)
-- reduce in nonICASE4 : istep100(p)
true : Bool
(0.000 sec for parse, 24 rewrites(0.016 sec), 161 matches)
-- reduce in ICASE5-1 : istep100(p)
true : Bool
(0.000 sec for parse, 33 rewrites(0.000 sec), 166 matches)
-- reduce in ICASE5-2 : istep100(p)
true : Bool
(0.000 sec for parse, 32 rewrites(0.000 sec), 166 matches)
-- reduce in nonICASE5 : istep100(p)
true : Bool
(0.000 sec for parse, 24 rewrites(0.000 sec), 159 matches)
-- reduce in ICASE6-1 : istep100(p)
true : Bool
(0.000 sec for parse, 37 rewrites(0.000 sec), 119 matches)
-- reduce in ICASE6-2 : inv320(s,t10,p) implies istep100(p)
```

```
true : Bool
(0.000 sec for parse, 59 rewrites(0.016 sec), 281 matches)
-- reduce in nonICASE6 : istep100(p)
true : Bool
(0.000 sec for parse, 24 rewrites(0.000 sec), 159 matches)
CafeOBJ>
```

As shown above, the result of each sub-case for induction is `true`, therefore, we have proved that INV100 always hold in the system with two lemmas, INV310 (in ICASE3-1) and INV320 (in ICASE6-2).

Appendix B

Model-checking Crossing Control System with Maude

In this appendix, we list the model-checking codes of the crossing control system with Maude as for reference below.

```
fmod TRAIN is
  sort Train .
endfm

fmod TSTATE is --- train state
  sort TState .
  ops sect0 sect1 critical sect2 : -> TState .
endfm

fmod LSTATE is --- level crossing state
  sort LState .
  ops state0 state1 state2 : -> LState .
endfm

fmod BSTATE is --- barrier state
  sort BState .
  ops open close : -> BState .
endfm

--- module TrainCrossing (TC) System
mod TC is
  pr TRAIN .
  pr TSTATE .
  pr LSTATE .
  pr BSTATE .

  sorts TRule OValue Sys .
  subsorts TRule OValue < Sys .
  *** Configuration
  op none : -> Sys .
  op __ : Sys Sys -> Sys [assoc comm id: none] .
  *** Observable values
  op tr[_] =_   : Train TState -> OValue .
  op le[_] =_   : Train LState -> OValue .
  op ba =_      : BState       -> OValue .
  op idling =_  : Bool         -> OValue .
```

```
  *** Transition rules
  op tr-send-secure  : Train -> TRule .
  op le-get-secure   : Train -> TRule .
  op le-send-release : Train -> TRule .
  op tr-get-release  : Train -> TRule .
  op tr-send-passed  : Train -> TRule .
  op le-get-passed   : Train -> TRule .
  *** Maude variables
  var T  : Train .
  var TS : TState .
  var LS : LState .
  var BS : BState .
  var IS : Bool   .  --- system idling variant

  *** tr-send-secure
  crl [tr-send-secure] :
    tr-send-secure(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
    => tr-send-secure(T) (tr[T] = sect1) (le[T] = LS) (ba = BS) (idling = IS)
    if TS == sect0 .
  *** le-get-secure
  crl [le-get-secure] :
    le-get-secure(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
    => le-get-secure(T) (tr[T] = TS) (le[T] = state1) (ba = close) (idling = false)
    if TS == sect1 and IS == true .
  *** le-send-release
  crl [le-send-release] :
    le-send-release(T) (tr[T] = TS) (le[T] = LS ) (ba = BS) (idling = IS)
    => le-send-release(T) (tr[T] = TS) (le[T] = state2) (ba = BS) (idling = IS)
    if LS == state1 .
  *** tr-get-release
  crl [tr-get-release] :
    tr-get-release(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
    => tr-get-release(T) (tr[T] = critical) (le[T] = LS) (ba = BS) (idling = IS)
    if LS == state2 .
  *** tr-send-passed
  crl [tr-send-passed] :
    tr-send-passed(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
    => tr-send-passed(T) (tr[T] = sect2) (le[T] = LS) (ba = BS) (idling = IS)
    if TS == critical .
  *** le-get-passed
  crl [le-get-passed] :
    le-get-passed(T) (tr[T] = TS) (le[T] = LS) (ba = BS) (idling = IS)
    => le-get-passed(T) (tr[T] = sect0) (le[T] = state0) (ba = open) (idling = true)
    if TS == sect2 .
endm

mod TC-PREDS is
  pr TC .
  inc SATISFACTION .
  subsort Sys < State .
  op wait          : Train -> Prop .
  op tr-oncrossing : Train -> Prop .
  op ba-open       :       -> Prop .
  op cr-release    : Train -> Prop .
  op sys-idle      :       -> Prop .
```

```
  var T : Train .
  var S : Sys .
  eq (tr[T] = sect1)    S |= wait(T) = true .
  eq (tr[T] = critical) S |= tr-oncrossing(T) = true .
  eq (ba = open)        S |= ba-open = true .
  eq (le[T] = state2)   S |= cr-release(T) = true .
  eq (idling = true)    S |= sys-idle = true .
endm

mod TC-CHECK is
  inc TC-PREDS .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
  ops t1 t2 t3   : -> Train .
  op  init       : -> Sys .
  op  mutex      : -> Formula .
  ops root basic : -> Formula .

  eq init =
       tr-send-secure(t1)  tr-send-secure(t2)  tr-send-secure(t3)
       le-get-secure(t1)   le-get-secure(t2)   le-get-secure(t3)
       le-send-release(t1) le-send-release(t2) le-send-release(t3)
       tr-get-release(t1)  tr-get-release(t2)  tr-get-release(t3)
       tr-send-passed(t1)  tr-send-passed(t2)  tr-send-passed(t3)
       le-get-passed(t1)   le-get-passed(t2)   le-get-passed(t3)
       (tr[t1] = sect0) (tr[t2] = sect0) (tr[t3] = sect0)
       (le[t1] = state0) (le[t2] = state0) (le[t3] = state0)
       (ba = open)
       (idling = true) .

  eq basic = ([] ~(cr-release(t1) /\ sys-idle)) /\
             ([] ~(cr-release(t2) /\ sys-idle)) /\
             ([] ~(cr-release(t3) /\ sys-idle)) .
  eq root =  ([] ~(tr-oncrossing(t1) /\ ba-open)) /\
             ([] ~(tr-oncrossing(t2) /\ ba-open)) /\
             ([] ~(tr-oncrossing(t3) /\ ba-open)) .
  eq mutex = ([] ~(tr-oncrossing(t1) /\ tr-oncrossing(t2))) /\
             ([] ~(tr-oncrossing(t2) /\ tr-oncrossing(t3))) /\
             ([] ~(tr-oncrossing(t1) /\ tr-oncrossing(t3))) .
endm

  red modelCheck(init, basic) .
  red modelCheck(init, root) .
  red modelCheck(init, mutex) .
```

And the experimental results are listed below, which shows that the properties are all preserved in the OTS.

```
d:\XJW\maude\train-crossing> maude
 maude
            \||||||||||||||||||/
          --- Welcome to Maude ---
            /||||||||||||||||||\
        Maude 2.1.1 built: Jul 30 2004 09:23:31
         Copyright 1997-2004 SRI International
           Fri Apr  8 13:11:48 2005
Maude> in model-checker.maude
==========================================
```

```
fmod LTL
=========================================
fmod LTL-SIMPLIFIER
=========================================
fmod SAT-SOLVER
=========================================
fmod SATISFACTION
=========================================
fmod MODEL-CHECKER Maude> in TC.maude
=========================================
---> Formal Specification begins
=========================================
fmod TRAIN
=========================================
fmod TSTATE
=========================================
fmod LSTATE
=========================================
fmod BSTATE
=========================================
mod TC
=========================================
mod TC-PREDS
=========================================
mod TC-CHECK
=========================================
reduce in TC-CHECK : modelCheck(init, basic) . rewrites: 1638 in
6574226999ms cpu (0ms real) (0 rewrites/second) result Bool: true
=========================================
reduce in TC-CHECK : modelCheck(init, root) . rewrites: 1638 in
4290626999ms cpu (0ms real) (0 rewrites/second) result Bool: true
=========================================
reduce in TC-CHECK : modelCheck(init, mutex) . rewrites: 1636 in
4290626999ms cpu (999ms real) (0 rewrites/second) result Bool: true
```

Publications

[1] Jianwen Xiang, Kokichi Futatsugi and Yanxiang He: "Formal Fault Tree Construction Model and Specification", Proc. of The 8th IASTED International Conference Software Engineering And Application '04, ACTA press, pp.374-381 (Cambridge, MA, USA, Nov. 2004).

[2] Jianwen Xiang, Kokichi Futatsugi and Yanxiang He: "Fault Tree and Formal Methods in System Safety Analysis", Proc. of The 4th International Conference on Computer and Information Technology '04, IEEE Computer Society Press, pp.1108-1115 (Wuhan, China, Sep. 2004).

[3] Jianwen Xiang, Kokichi Futatsugi and Yanxiang He: "Formal Fault Tree Construction and System Safety Analysis", Proc. of IASTED International Conference on Software Engineering '04, ACTA press, pp.378-384 (Innsbruck, Austria, Feb. 2004).

[4] Jianwen Xiang, Kokichi Futatsugi and Yanxiang He: "Fault Tree Analysis of Software Reliability Allocation", Proc. of The 7th World Multiconference on Systemics, Cybernetics and Informatics '03, Volume II - Computer Science and Engineering, pp.460-465 (Orlando, USA, July 2003, **The Best Paper**).

[5] Jianwen Xiang, Yanxiang He, Kokichi Futatsugi, and Weiqiang Kong: "Constructing Projection Frequent Pattern Tree for Efficient Mining", Wuhan University Journal of Natural Sciences, Vol. 8 No. 2A, pp.351-357 (June 2003).

[6] Weiqiang Kong, Kazuhiro Ogata, Jianwen Xiang, and Kokichi Futatsugi: "Formal Analysis of an Anonymous Fair Exchange E-Commerce Protocol",Proc. of The 4th International Conference on Computer and Information Technology '04, IEEE Computer Society Press, pp.1100-1107 (Wuhan, China, Sep. 2004).

[7] Jing Tian, Yoshiteru Nakamori, Jianwen Xiang and Kokichi Futatsugi: "Knowledge Management in Academia: Survey, Analysis and Perspective", Proc. of The 17th International Conference on Multiple Criteria Decision Analysis '04, (Whistler, Canada, Aug. 2004).

[8] Jing Tian, Yoshiteru Nakamori, Jianwen Xiang and Kokichi Futatsugi: "Knowledge Management in Academia: Survey, Analysis and Perspective", Int. Journal Management and Decision Making (to publish).

[9] Jianwen Xaiang, Kazuhiro Ogata, and Kokichi Futatsugi: "Formal Fault Tree Analysis of State Transition Systems", The 5th International Conference on Quality Software (QSIC05), IEEE Computer Society Press (Melbourne, Australia, Sep. 2005, *to publish*).

Bibliography

[AC77] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. In *Proceedings COMPSAC 77*, pages 149–155, 1977.

[And90] T. E. Anderson. The performance of Spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 1(1):6–16, 1990.

[Avi85] A. Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, Dec 1985.

[BA93] G. Bruns and S. Anderson. Validating safety models with fault trees. In J. Górski, editor, *SafeComp'93: 12th International Conference on Computer Safety, Reliability, and Security*, pages 21–30. Springer-Verlag, 1993.

[Bah97] N. J. Bahr. *System Safety Engineering and Risk Assesment: A Practical Approach*, chapter Fault Tree Analysis. Taylor & Francis, 1997.

[bet96] Betriebliches lastenheft für funkfahrbetrieb. stand 1.10, 1996.

[BG80] Rod Burstall and Joseph Goguen. The semantics of clear, a specification language. In Dines Bjorner, editor, *Proceedings 1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer-Verlag, 1980.

[CA78] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. of The 8th International Symposium on Fault-Tolerant Computing*, pages 3–9, Toulouse, France, Jun 1978.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming language in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[CDE+03a] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude 2.0 Manual: Version 1.0*, June 2003.

[CDE+03b] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In *Proc. of the 14th International conference on Rewriting Techniques and Applications (RTA 2003)*, volume LNCS, pages 76–87. Springer, 2003.

[CELM96] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. In Jose Meseguer, editor, *Proceedings of The First International Workshop on Rewriting Logic and its Applications*, volume 4 of Electronic Notes in Theoretical Computer Science, Asilomar, California, Sep 1996. Elsevier Science.

[CGP01] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2001.

[CM98] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1998.

[CSD00] David Coppit, Kevin J. Sullivan, and Joanne Bechta Dugan. Formal semantics of models for computational engineering: A case study on dynamic fault trees. In *Proc. of The 11th International Symposium on Software Reliability Engineering*, pages 270–282, San Jose, California, USA, Oct 2000.

[DB93] J. B. Dugan and R. Van Buren. Reliability evaluation of flyby wire computer systems. *Journal of Systems and Safety*, Jun 1993.

[DBB92] J. B. Dugan, S. Bavuso, and M. Boyd. Dynamic fault tree models for fault tolerant computer systems. *IEEE Transactions on Reliability*, Sep 1992.

[DF96] Răzvan Diaconescu and Kokichi Futatsugi. Logical semantics of CafeOBJ. Technical Report IS-RR-96-0024S, Japan Advanced Institute of Science and Technology, 1996.

[DF98] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific Pub., 1998.

[DF00] Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 2000. First version appeared at JAIST Technical Report IS-RR-98-0017F, July 1998.

[Dia98] Răzvan Diaconescu. Extra theory morphisms for institutions: Logical semantics for multi-paradigm languages. *Journal of Applied Categorical Structures*, 6(4):427–453, 1998.

[Div87] Air Force Space Division. System safety handbook for the acuquisition manager. Technical Report SDP 127-1, January 12 1987.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 245–320. The MIT Press/Elsevier Science Publishers, 1990.

[DL95] J. B. Dugan and M. R. Lyu. *Dependability Modeling for Fault-Tolerant Software and Systems*, chapter 5 in Software Fault Tolerance. John Wiley & Sons, Chichester, UK, 1995.

[DT03] H. Dierks and J. Tapken. Moby/dc - a tool for model-checking parametric real-time specifications. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes of Computer Science*, pages 271–277. Springer, 2003.

[DVG97] J. B. Dugan, B. Venkataraman, and R. Gulati. Diftree: A software package for the analysis of dynamic fault tree models. In *Annual Reliability and Maintainability Symposium, 97RM-047*, pages 1–7, 1997.

[ea83] S. Hope et al. Methodologies for hazard analysis and risk assessment in the petroleum refining and storage industry. *journal of System Safety*, July/August:24–32, 1983.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer-Verlag, 1985.

[EMS02] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In *Proc. of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA 2002)*, volume ENTCS 71. Elsevier, 2002.

[FGJM85] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. of Symposia Applied Mathematics*, volume 19 of Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, 1967.

[FMNP85] P. Fenelon, J. McDermid, A. Nicholson, and D. Pumfrey. Experience with the Application of HAZOP to Computer-Based Systems. In *Proc. of The 10th Annual Conference on Computer Assurance*, pages 37–48, Gaithersburg, MD, USA, 1985.

[FN97] Kokichi Futatsugi and Ataru Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. 1st International Conference on Formal Engineering Methods*, pages 170–181. IEEE, 1997.

[FNT00] Kokichi Futatsugi, A. T. Nakagawa, and T. Tamai. *CAFE: an Industrial-Strength Algebraic Formal Method*. Elsevier Science, Amsterdam, 2000.

[FS95] Kokichi Futatsugi and Toshimi Sawada. Design considerations for Cafe specification environment. In *Proc. OBJ2 10th Anniversary Workshop*, Oct 1995.

[GB92] Joseph A. Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, Jan 1992.

[GD94a] Joseph A. Goguen and Răzvan Diaconescu. An oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, 1994.

[GD94b] Joseph A. Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Harmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*, pages 1–34. Springer-Verlag, 1994.

[GM89] Joseph A. Goguen and J. Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance, polymorphism, overloading and partial operations. Technical Report SRI-CSL-89-10, SRI International, 1989.

[GM97] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, University of California at San Diego, 1997.

[Gór94] J. Górski. Extending safety analysis techniques with formal semantics. In F. J. Redmill and T. Anderson, editors, *Technology and Assessment of Safety Critical Systems*, pages 147–163. Springer Verlag, London, 1994.

[Ham80] Willie Hammer. *Product Safety Management and Engineering*. Prentice-Hall, Englewood Cliffs, N.J., 1980.

[HH86] H. Hecht and M. Hecht. *Fault-Tolerant Computing: Theory and Techniques*, volume 2, chapter Fault-Tolerant Software, pages 658–696. Prentice-Hall, Englewood Cliffe, New Jersey, 1986.

[HK00] Ernest J. Henley and Hiromitsu Kumamoto. *Probabilistic Risk Assessment and Management for Engineers and Scientists*. Wiley-IEEE Press, 2nd edition, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HR98] Kirsten M. Hansen and Anders P. Ravn. From safety analysis to software requirement. *IEEE Transactions on Software Engineering*, 24(7):573–584, Jul 1998.

[HRS94] Kirsten M. Hansen, Anders P. Ravn, and V. Stavridou. From safety analysis to formal specification. Technical Report ID/DTH KMH 1/1, ProCoS II, ESPRIT BRA 7071, ID/DTH, Lyngby, Denmark, 1994.

[HWS+00] G. Helmer, J. Wong, M. Slagell, V. Honavar, L. Miller, and R. Lutz. Software fault tree and colored petri net based specification, design, and implementation of agent-based intrusion detection systems. Submitted to ACM Transactions on Information and System Security, 2000.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of European Association for Theoretical Computer Science*, 62:222–259, 1997.

[JWM+00] J.Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. *Software Engineering with OBJ: algebraic specification in action*, chapter Introducing OBJ, pages 3–167. Kluwer Academic Publishers, 2000.

[Klo87] Jan Willem Klop. Term rewriting systems: A tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 32:143–182, 1987.

[Koy92] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Springer-Verlag, New York, 1992.

[KW89] K. H. Kim and H. O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in realtime applications. *IEEE Transactions on Computers*, 38(5):626–636, May 1989.

[LABK90] Jean-Claude Laprie, J. Arlat, C. Béounes, and M. Kaaniche. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, Jul 1990. Reprinted in Fault-Tolerant Software Systems: Techniques and Applications, Hoang Pham(ed.), IEEE Computer Society Press, 1992, pp. 5-17.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lee80] Frank P. Lees. *Loss Prevention in the Process Industries*, volume 1 and 2. Butterworths, London, 1980.

[Lev87] Nancy G. Leveson. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, SE-13(3):386–397, March 1987.

[Lev91] Nancy G. Leveson. Safety verification of ada programs using software fault trees. *IEEE Software*, pages 48–59, Jul 1991.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Pub., Sep 1995.

[LH83] Nancy G. Leveson and Peter R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, Sep 1983.

[Liu00] Shaoyin Liu. Verifying formal specifications using fault tree analysis. In *Proc. of The International Symposium on Principles of Software Evolution*. IEEE, 2000.

[LM99] Axel Lankenau and Oliver Meyer. Formal methods in robotics: Fault tree based verification. In *Proc. of The Third International Software Quality Week Europe*, 1999.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[Lyu95] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1995.

[McC81] Norman J. McCormick. *Reliability and Risk Analysis*. Academic Press, New York, 1981.

[McI83] Jams W. McIntee. Fault tree techniques as applied to software (soft tree). Technical report, Dept. of Air Force, U.S., 1983.

[MDCS98] Ragavan Manian, J. B. Dugan, D. Coppit, and K. J. Sullivan. Combining various solution techniques for dynamic fault tree analysis of computer systems. In *The Third IEEE International High-Assurance Systems Engineering Symposium*, pages 21–28. IEEE, Nov 1998.

[Mes91] José Meseguer. Conditional rewriting logic: Deduction, models and concurrency. In *Proc. 2nd International CTRS Workshop*, volume 516 of *Lecture Notes in Computer Science*, pages 64–91. Springer-Verlag, 1991.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes97] José Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Pressice, editor, *Proc. of The 12th International Workshop on Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1997.

[MG86] José Meseguer and J. A. Goguen. Initiality, induction, and computability. In *Algebraic methods in semantics*, pages 459–541. Cambridge University Press, New York, NY, USA, 1986.

[MIO87] J. Musa, D. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Anolication.* McGraw-Hill, New York, 1987.

[MJC+99] Sang-Yoon Min, Yoon-Kyu Jang, Sung-Deok Cha, Yong-Rae Kwon, and Doo-Hwan Bae. Safety verification of ada95 programs using software fault trees. In M. Felici, K. Kanoun, and A. Pasquini, editors, *Proc. of The 18th International Conference on Computer Safety, Security, and Security*, volume 1689 of *Leture Notes in Computer Science*, pages 226–238. Springer, 1999.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MtSG96] Zohar Manna and the STeP Group. Step: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. CAV'96 - 8th Intl. Conf. on Computer-Aided Verification*, volume LNCS 1102, pages 415–418. Springer-Verlag, July 1996.

[OF99] Kazuhiro Ogata and Kokichi Futatsugi. Specification and verification of some classical mutual exclusion algorithms with CafeOBJ. In J. Meseguer K. Futatsugi, J.A. Goguen, editor, *OBJ/CafeOBJ/Maude Workshop at Formal Methods*, pages 159–178, Bucharest, 1999.

[OF02] Kazuhiro Ogata and Kokichi Futatsugi. Formal analysis of suzuki & kasami distributed mutual exclusion algorithm. In Bart Jacobs and Arend Rensink, editors, *Proc. of IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 181–195, Braunschweig, Germany, 2002. Kluwer.

[OF03] Kazuhiro Ogata and Kokichi Futatsugi. Proof scores in the OTS/CafeOBJ method. In *Proc. of The 6th IFIP WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pages 170–184. Springer, 2003.

[Pet81] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[Rei83] D. Reifer. Software failure modes and effects analysis. *IEEE Transactions on Reliability*, 28(3):147–249, August 1983.

[RST00] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Safety analysis of a radio-based crossing control system using formal methods. In *Proc. 9th IFAC Symposium Control in Transportations Systems 2000*, pages 289–294, Braunschweig, Germany, 2000.

[sch03] Andreas schäfer. Combining real-time model-checking and fualt tree analysis. In D. Mandrioli, K. Araki, and S. Gnesi, editors, *Proc. of The 12th International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 522–541, Pisa, Italy, Sep 2003. Springer.

[Sho90] M. L. Shooman. *Probabilistic Reliability: An Engineering Approach.* McGraw-Hill, New York, 2nd edition, 1990.

[STR02] Gerhard Schellhorn, Andreas Thums, and Wolfgang Reif. Formal fault tree semantics. In *Proc. of The 6th World Conference on Integrated Design and Process Technology*, Pasadena, CA, 2002.

[Tap01] J. Tapken. *Model-Checking of Duration Calculus Specifications.* PhD thesis, Carl von Ossietzky Universität Oldenburg, 2001.

[Tay82] J. R. Taylor. Fault tree and cause-consequence analysis for control software validation. Technical Report RISO-M-2326, Riso National Laboratory, Roskilde, Denmark, Jan 1982.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault tree handbook. Technical Report NUREG-0492, U.S. Nuclear Regulatory Commission, Washington, D.C, Jan 1981.

[vL91] Axel van Lamsweerde. Learning machine learning. In Andre Thayse, editor, *Introducing a logic based approach to artificial intelligence*, volume 3. Wiley, 1991.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–263, Toronto, August 2001.

[vLDL98] Axel van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[vLL00] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, 2000.

[WL61] H. A. Watson and Bell Telephone Laboratories. Launch control safety study. Technical report, Bell Telephone Laboratories, Murray Hill, NJ, 1961.

[XFH03] Jianwen Xiang, Kokichi Futatsugi, and Yanxiang He. Fault tree analysis of software reliability allocation. In *Proc. of The 7th World Multiconference on Systemics, Cybernetics and Informatics*, volume Volume II - Computer Science and Engineering, pages 460–465, Orlando, USA, Jul 2003. International Institute of Informatics and Systemics.

[XFH04a] Jianwen Xiang, Kokichi Futatsugi, and Yanxiang He. Fault tree and formal methods in system safety analysis. In *Proc. of The 4th International Conference on Computer and Information Technology*, pages 1108–1105. IEEE, Sep 2004.

[XFH04b] Jianwen Xiang, Kokichi Futatsugi, and Yanxiang He. Formal fault tree construction and system safety analysis. In *Proc. of The IASTED International Conference on Software Engineering*, pages 378–384, Innsbruck, Austria, Feb 2004. International Association of Science and Technology for Development, ACTA Press.

[XFH04c] Jianwen Xiang, Kokichi Futatsugi, and Yanxiang He. Formal fault tree construction model and specification. In *Proc. of The 8th IASTED International Conference on SOFTWARE ENGINEERING AND APPLICATIONS*, pages 374–381, Cambridge, MA, USA, Nov 2004. International Association of Science and Technology for Development, ACTA Press.

Index