Formal Fault Tree Analysis: Practical Experiences

Frank Ortmeier, Gerhard Schellhorn

Formal Fault Tree Analysis: Practical Experiences

Frank Ortmeier Gerhard Schellhorn

Lehrstuhl für Softwaretechnik und Programmiersprachen
Universität Augsburg
D-86135 Augsburg

Abstract

Safety is an important requirement for many modern systems. To ensure safety of complex critical systems, well-known safety analysis methods have been formalized. This holds in particular for automation sytsems and transportation systems. In this paper we present the formalization of one of the most wide spread safety analysis methods: fault tree analysis (FTA). Formal FTA allows to rigorously reason about completeness of a faulty tree. This means it is possible to prove whether a certain combination of component failures is critical for system failure or not. This is a big step forward as informal reasoning on cause-consequence relations is very error-prone.
We report on our experiences with a real world case study from the domain of railroads. The here presented case study is – to our knowledge – the first complete formal fault tree analysis for an infinite state system. Until now only finite state systems have been analyzed with formal FTA by using model checking.

Keywords: fault tree analysis, dependability, safety analysis, formal methods

1 Introduction

Many critical accidents in the last years show that the risk modern systems bring is rising (e.g. the recent accidents in china's chemical plants or the german ICE accident at Eschede). As a result safety is becoming a more and more important issue in system development . At the same time new systems become increasingly complex. This makes safety analysis both more important and more difficult. Therefore new and better analysis methods must be developed. One such technique is formal FTA. FFTA is a formal variant of well-known FTA. The benefit is, that cause-consequence relations between component failure and system failure can be rigorously proved. This is less error-prone than informal reasoning and yields much better results.

In this paper we present the first formal fault tree analysis of an infinite state system, the problems we faced and the lesson we learned. We use the formal fault tree semantics of [14]. Verification was done with the KIV system [2]. This case study can also be seen as a guideline on how to do formal FTA in an interactive verification environment. From a safety point of view the problems and solutions found in the presented case study are exemplary for a big group of safety critical systems.

A revised version of this paper will be electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

In Sect. 2 we give a brief introduction on FTA, summarize the formal semantics and revisit the semantics of Harel's state charts [9]. Sect. 3 presents a real world case study from the domain of railroads. Learned lessons are presented in Sect. 4. A conclusion and an outlook is given in Sect. 5.

2 Formal Fault Tree Analysis

A well-known safety analysis technique is fault tree analysis (FTA, [16]). FTA was developed for technical systems to analyze if they permit a hazard (top event). The top event is noted at the root of the fault tree. Events which cause the hazard are given in the child nodes and analyzed recursively, resulting in a tree of events. Each analyzed event (main event) is connected to its causes (sub-events) by a gate in the fault tree (see Fig. 1). An AND-gate indicates that all sub-events are necessary to trigger the main event, for an OR-gate only one sub-event is necessary. An INHIBIT-gate states that in addition to the cause stated in the sub-event the condition (noted in the oval) has to be true to trigger the main event. The inhibit gate is more or less an AND-gate, where the condition does not have to be a fault. The leaves of the tree (basic events) are failure modes at component level. These failures have to occur in certain combinations (corresponding to the AND/OR structure of the tree), before the top event can occur i.e. the system fails. An example fault tree is shown in Fig. 8.

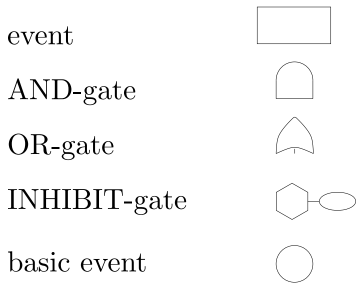


Fig. 1. Fault Tree Symbols

A combination of basic events which leads to the hazard is called a *cut set*. A *minimal cut set* is a cut set which can not lead to the top level hazard, if only one event of the set is prevented. A typical example is that in redundant systems for system failure it is necessary that the primary and the secondary unit fail (for e.g. the electric and the hydraulic braking module). This information helps to identify failure events whose exclusion secures the system. If for example one event occurs in different minimal cut sets, the probability of the top level hazard will strongly decrease, if this event can be excluded.

Minimal cut sets can be computed from fault trees by combining the primary events with boolean operators as indicated by the gates. A minimal cut set then consists of the elements of one conjunction of the disjunctive normal form of the resulting formula.

For formal FTA, each gate is represented by an interval temporal logic (ITL) formula. Temporal formulas in ITL are built from first-order formulas using propo-

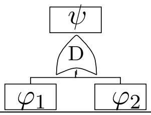
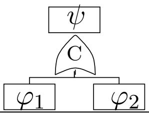
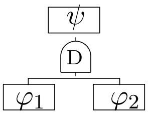
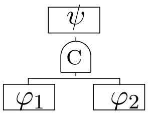
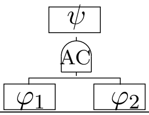
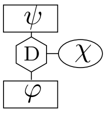
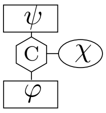
Gate	ITL-formula	Gate	ITL-formula
	$\boxtimes\ (\psi \rightarrow \varphi_1 \vee \varphi_2)$		$\neg\ (\neg \diamondtimes\ (\varphi_1 \vee \varphi_2)\ ;\ \Diamond_i\ \psi)$
	$\boxtimes\ (\psi \rightarrow \varphi_1 \wedge \varphi_2)$		$\neg\ (\neg \diamondtimes\ (\varphi_1 \wedge \varphi_2)\ ;\ \Diamond_i\ \psi)$
			$\neg\ (\neg \diamondtimes\ \varphi_1\ ;\ \Diamond_i\ \psi)$ $\wedge\ \neg\ (\neg \diamondtimes\ \varphi_2\ ;\ \Diamond_i\ \psi)$
	$\boxtimes\ (\psi \rightarrow \varphi \wedge \chi)$		$\neg\ (\neg \diamondtimes\ (\varphi)\ ;\ \Diamond_i\ \psi)$ $\wedge\ \neg\ (\neg \diamondtimes\ (\chi)\ ;\ \Diamond_i\ \psi)$

Fig. 2. Formal semantics of fault trees

sitional connectives and the following temporal operators[1]: $\boxed{i}\ \varphi$ ("in all initial intervals φ"), $\boxtimes\ \varphi$ ("in all subintervals φ"), $\Diamond_i\ \varphi$ ("in some initial interval φ"), $\diamondtimes\ \varphi$ ("in some subinterval φ"), and $\varphi\ ;\ \psi$ (read φ chop ψ: "the interval can be split, such that φ holds in the first part and ψ in the second").

The formalization of FTA showed, that defining the semantics of an OR-gate simply as a disjunction is insufficient, since it does not take into account that the sub-events (causes) usually happen *before* the main event (consequence), and that events may have duration. Therefore, it is necessary to distinguish decomposition gates (D-gates) with boolean semantics and cause-consequence gates (C-gates), which describe temporal dependencies. This results in 7 types of gates. The gates and there formalizations are shown in Fig. 2. For example the FTA formula for D-gates (left column) and C-gates (right column) are shown.

D-OR- and D-AND-gates (D, D) can be defined canonically: for example the D-AND-gate (D) states, that whenever the effect ψ happens, both causes φ_1 and φ_2 must happen as well. A C-OR-gates (C) states, that it must not be possible to split a run, such that none of the causes φ_1 and φ_2 ever happens in the first half, but the consequence ψ happens at the beginning of the second half. In other words: if the consequence happens, one of the causes must have happened before (*completely*, if it has duration, therefore the chop is necessary). Causes and consequences must not overlap. The asynchronous and synchronous C-AND-gates (C, AC) are similar, they require that both causes must have happened (at the same time) before the consequence. The conditions for D-INHIBIT- and C-INHIBIT-gates (D,C) are the same as for the D-AND-gate and AC-gate.

Hansen et al. [8] defines cause-consequence gates in Duration Calculus (DC, [17]), but their definition does not meet the requirement, that causes are completed

[1] ITL also defines quantification and many other derived operators not needed here. More information may be found in [1]

before the consequence. A subsequent publication [7] is restricted to decomposition gates. Bruns and Andersen [4] also define a fault tree semantics using μ-calculus. They also distinguish between cause-consequence and decomposition gates. Only events without duration are considered. For this special case, our semantics is equivalent (see [15] for details).

For the semantics in Fig. 2, the following theorem was proven:

Theorem 2.1 (minimal cut set theorem) *If all conditions of a fault tree are verified, and if for each minimal cut set at least one of its basic events is prevented from happening, then the top-level event will never happen.*

This means in practice, that if you verify for every gate in the fault tree the corresponding formula, then you can be sure, that you have not forgotten any branches in the fault tree (i.e. no combination of failure modes has been ”overlooked”).

In other words, it is sufficient to prevent only one primary event of each minimal cut set, to avoid system failure. A complete fault tree is therefore a partial proof for the safety of the system. It shows what combinations of component failures are necessary reason for system failure. The completeness theorem also gives formal justification for the use of minimal cut sets in quantitative safety analysis, even for cases where timing conditions are relevant [13].

The theorem is proved using structural induction over the size of the fault tree. The basic fact underlying the proof is transitivity of the cause-consequence relation. The proof was done formally with the KIV system ([2]), using an algebraic specification of the syntax and semantics of continuous Interval Temporal Logic.

2.1 State charts

In this paper systems are modeled as *State charts*. *State charts* exist in several different variations. The most commonly known are UML *State charts* and *Statemate State charts*. For *Statemate State charts* a formal semantics has been defined by Harel and Damm [6]. In this paper *Statemate State charts* are used as system models. This semantics has been integrated in a model checking extension to *Statemate* [3] and is also supported by the interactive theorem prover KIV [2].

State charts may be seen as an extension to traditional state-transition-systems. A single *state chart* comprise a set of (sub-)*State charts*, a set of labeled transitions and an initial state. Figure 3 shows a very basic *state chart*.

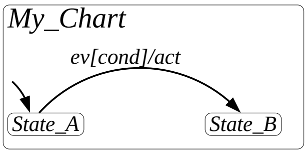


Fig. 3. A basic *state chart*

The *state chart My_Chart* has two (sub-)*State charts*: *State_A* and *State_B*. A transition from *State_A* to *State_B* happens if and only if, the event *ev* is triggered and the condition *cond* holds. If the transition is taken, the action *act* will be executed. The difference between events and conditions is that events are only one time step active (there is no event queue like in UML *State charts*) while conditions may be arbitrary (non-temporal) formulas. Thus conditions may in general hold several time steps. *Act* is a C-like program which is executed atomically when the transition is taken. If several transition are simultaneously possible, then one will be chosen indeterministically. A more detailed description of the semantics of *Statemate State charts* may be found in the tool's documentation or in [6].

For the purpose of formal proofs events and conditions may be treated analogously. From now on we write $event \wedge cond/act$ for $event[cond]/act$. If no action is defined we simply write $event \wedge cond$. If no condition and no event is guarding the transition (spontaneous transition) then we use $/act$. Unlabeled transitions are spontaneous transitions with no action defined.

3 A case study

As an example for the application of formal FTA, we present an analysis of a radio-based railroad crossing. The case study was done using the interactive theorem prover KIV [2] and the proof effort was about 1.5 person months. This case study is the reference case study of the german research councils (DFG) priority program 1064. This program aims at bringing together field-tested engineering techniques with modern methods of the domain of software engineering.

The German railway organization, Deutsche Bahn, prepares a novel technique to control railroad crossings: decentralized, radio-based railroad crossing control. This technique aims at medium speed routes, i.e. routes with maximum speed of 160 km/h. An overview is given in [10].

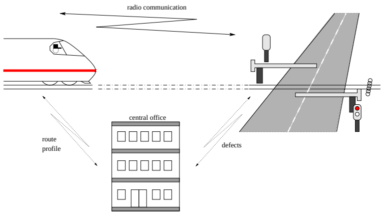


Fig. 4. Radio-based railroad crossing

The main difference between this technology and the traditional control of railroad crossings is that signals and sensors on the route are replaced by radio communication and software computations in the train and railroad crossing (see Fig. 4). This offers cheaper and more flexible solutions, but also shifts safety critical functionality from hardware to software.

Instead of detecting an approaching train by a sensor, sending this information to a central office which closes the railroad crossing, the train continously computes the position where it has to send a signal to secure the level crossing. This effectively saves money (not so much equipment on the track is needed) and removes the central control office (this is a single point of failure for all trains in the region). To calculate the activation point the train uses data about its position, maximum deceleration and the position of the crossing. Therefore the train has to know the position of the railroad crossing, the time needed to secure the railroad crossing, and its current speed and position. The first two items are memorized in a data store and the last two items are measured by an odometer. For safety reasons a safety margin is added to the activation distance. This allows compensating some deviations in the odometer. The system works as follows:

The train continuously computes its position. When it approaches a crossing, it broadcasts a 'secure'-request to the crossing. When the railroad crossing receives the command 'secure', it switches on the traffic lights, first the 'yellow' light, then the 'red' light, and finally closes the barriers. When they are closed, the railroad crossing is 'secured' for a certain period of time. The 'stop' signal on the train route, indicating an insecure crossing, is removed and substituted by computation and communication. Shortly before the train reaches the 'latest braking point' (latest point, where it is possible for the train to stop in front of the crossing), it requests the status of the railroad crossing. When the crossing is secured, it responds with a 'release' signal which indicates, that the train may pass the crossing. Otherwise the train has to brake and stop before the crossing. The railroad crossing periodically performs self-diagnosis and automatically informs the central office about defects and problems. The central office is also responsible for repair and provides route descriptions for trains. These descriptions indicate the positions of railroad crossings and maximum speed on the route. The safety goal of the system is clear: it must never happen, that the train passes a crossing which is not secured.

A well designed control system must assure this property at least as long as no component failures occur. The corresponding hazard is "a train passes the crossing and the crossing is not secured". This is the only hazard which we will consider in this case study

3.1 The formal model

In the following part a brief description of the *state chart* model of this system is given. Note, that the model not only includes intended behavior but failure modes as well. This is necessary for all types of formal safety analysis. Details on how such models may be derived from functional models of the intended behavior may be found in [12] and [11].

The model of the radio-based railroad crossing is split in three parallel charts. One chart models the crossing another one models the communication and a third models the train. These three charts are explained below.

3.1.1 Model of the crossing

The *state chart* in figure 5 shows the model of the crossing which is reacting to the signals sent by the train. Initially the crossing is in state *Opened*, which means the bars are open. When the crossing receives the signal *Close_Request_Rcv* from the train, it goes into state *Closing*. This activates a timer called *Closing_Count* which simulates the time needed for turning on the light signals at the crossing and the closing of the bars. This takes the time ($T\_Max\_Closing$). After the expiration of this time the crossing is closed (state *Closed*). Another timer *Closed_Count* is started to assure that the bars are not closed too long. This is a standard procedure in railroad organization. The crossing reopens if either the train passes the danger spot ($Pos > DS$) or the timer reached T_Max_Closed. The crossing also opens its bars if a fault in the sensor, which detects the passing of the train, occurs

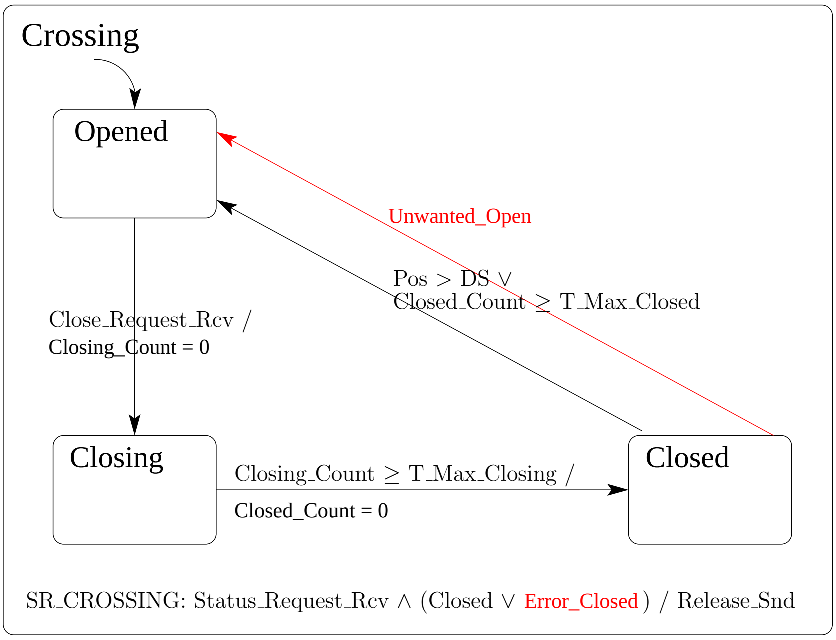

Crossing
Opened
Unwanted_Open
Pos > DS ∨
Closed_Count ≥ T_Max_Closed
Close_Request_Rcv /
Closing_Count = 0
Closing
Closing_Count ≥ T_Max_Closing /
Closed_Count = 0
Closed
SR_CROSSING: Status_Request_Rcv ∧ (Closed ∨ Error_Closed) / Release_Snd

Fig. 5. Model of the crossing-chart

($Unwanted\_Open$). The response of the crossing on the train's status request is modeled by a static reaction ($SR\_CROSSING$). If it receives a status request ($Status\_Request\_Rcv$), a release message ($Release\_Snd$) will be sent if the bars are closed (intended behavior) or if there is a faulty detection at the sensor for the bars' position ($Error\_Closed$).

3.1.2 Model of the train

The model of the train is divided into two parts: one for modeling the physics of the train and one for modeling the controller logic. From a theoretical point of view, it is advisable to model the control and the physics of the train separately. But in this example, the physical model consists only of some static reactions (see fig. 6). These static reactions basically state, that the position of the train updates according to the speed and that the speed updates according to the acceleration. So for an easier representation these two parts have been combined.

The train control supervises the position of the train, issues closing requests to the crossing and ultimately decides, if an emergency stop is necessary or not. The train control is implemented in software on-board the train. The formal model is given in figure 6. Starting from its initial state Idle the chart goes into state Wfc ('wait for close'), when the train approaches the crossing and the control sends a close request ($Close\_Request\_Snd$) to the crossing. The point when this signal is sent is continously calculated depending on the actual speed, estimated closing and communication time, and the maximum deceleration of the train. This is modeled in the predicate $Close(Pos, V, Acc_{MAX}, DS)$. Some time later, the train reaches another virtual control point which is also calculated continously and modeled in predicate $Request(Pos, V, Acc_{MAX}, DS)$. This is the position when the train sends a status request ($Status\_Request\_Snd$) to the crossing. The control is then in state Wfs ('wait for status answer'). If the train receives a release signal within the next

Wfs_*Count* time units the controller will go into state *Go* and the train may pass the crossing. Otherwise an emergency stop must be issued. In this case the brakes are activated ($A = Acc_{MAX}$) and the controller goes into state *Brake*. A failure of the brakes is also modeled. If the brakes fail, the controller will still go into state *Brake*, but there will be no real deceleration. The two states *Brake* and *Go* are final states of the chart, so they won't be left anymore.

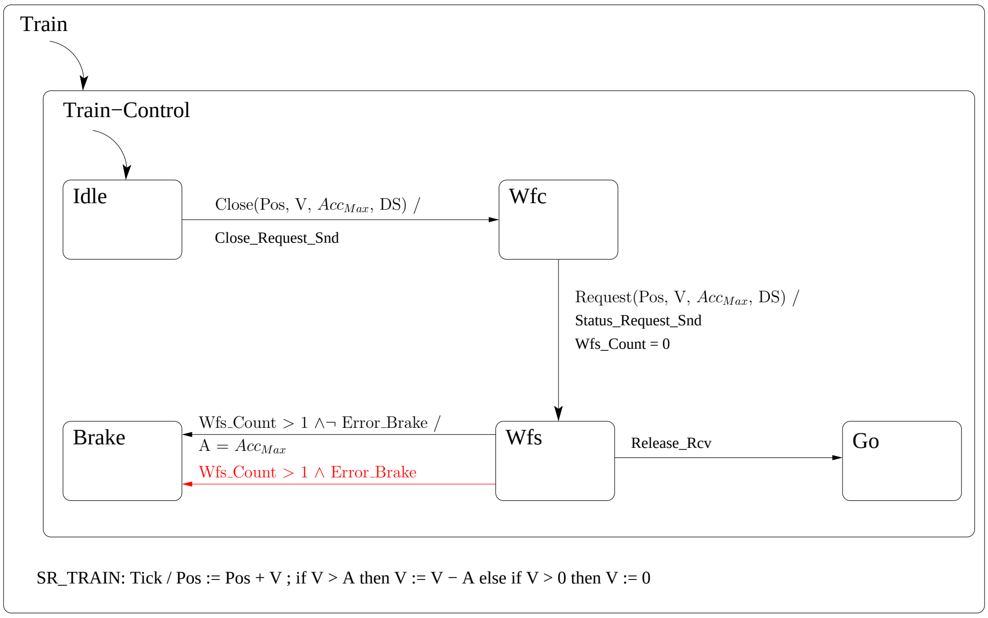

Train
Train−Control
Idle
Close(Pos, V, Acc_{Max}, DS) /
Close_Request_Snd
Wfc
Request(Pos, V, Acc_{Max}, DS) /
Status_Request_Snd
Wfs_Count = 0
Brake
Wfs_Count > 1 ∧¬ Error_Brake /
A = Acc_{Max}
Wfs_Count > 1 ∧ Error_Brake
Wfs
Release_Rcv
Go
SR_TRAIN: Tick / Pos := Pos + V ; if V > A then V := V − A else if V > 0 then V := 0

Fig. 6. Model of the train-control-chart

3.1.3 Model of the communication

The communication is modeled by three static reactions, see figure 7. These static reactions represent the function and disfunction of the communication. The functional communication relays all incoming messages, e.g. (SR_COMM1) the close request of the train ($Close\_Request\_Snd$) is forwarded to the crossing as $Close\_Request\_Rcv$. If the communication fails ($Failure\_Comm$) then no messages will reach their receiver. The other two static reactions represent the status request (SR_COMM2) and the release message (SR_COMM3).

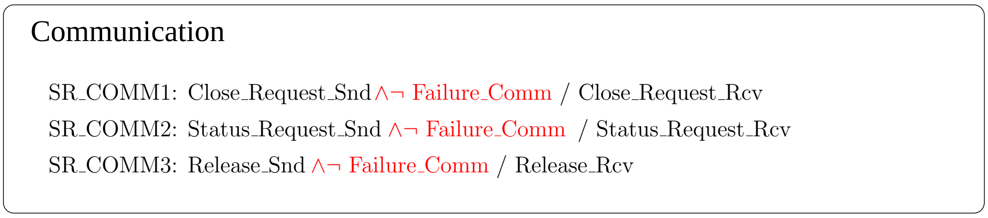

Communication
SR_COMM1: Close_Request_Snd ∧¬ Failure_Comm / Close_Request_Rcv
SR_COMM2: Status_Request_Snd ∧¬ Failure_Comm / Status_Request_Rcv
SR_COMM3: Release_Snd ∧¬ Failure_Comm / Release_Rcv

Fig. 7. Model of the communication-chart

3.2 Fault Tree Analysis

This model is now analyzed with formal FTA (see Sect. 2). The interesting hazard is a situation, where a train passes the crossing, while the bars are not closed. We will call this hazard "collision". The fault tree for this hazard is shown in figure 8.

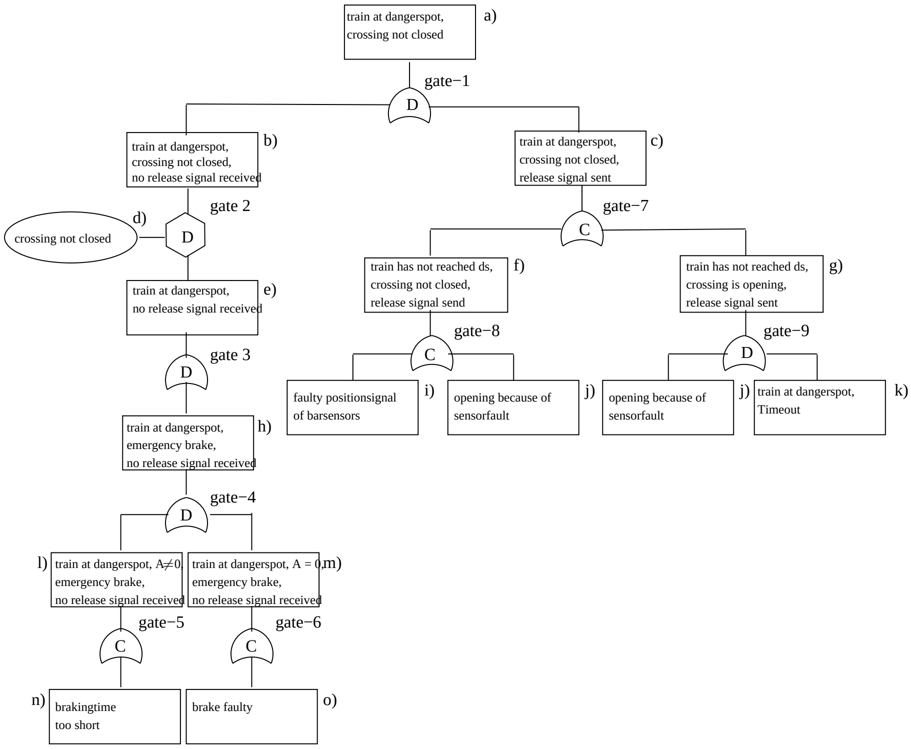


Fig. 8. Fault tree for hazard collision

The top event of the fault tree (collision) may have two different causes. One is that the train passes the crossing, while the bars are not closed, although no release signal has been sent. The other is a situation where the train passes the crossing, while the bars are not closed, but a release signal has been sent. The first cause corresponds to a misbehavior of the train and the second to one of the crossing. The "or" relationship is modeled by a decomposition gate. These two different situations must be further analyzed.

The left node — train passes the crossing (while the crossing is not closed) although no release signal has been received, is caused by a failure in the train's behavior, so no information about the crossing is needed. This is phrased by a D-INHIBIT-gate. The right node, train passing the not closed crossing and a release signal has been sent, can be caused by two different situations. One is given by the train approaching the not closed crossing and the release signal is being sent (while the crossing is not closed). The reason for this can be a fault in the position sensors of the bars[2]. The other possible reason is, that the bars open after a release signal has been sent but before the train has passed the crossing. The reason for this can be either a timeout or a faulty request to open the bars.The other cause is given by the train passing the opening/opened crossing and the signal has been sent some time before.

As an example, the formalization of the first three nodes is shown in table 1.

[2] or if SIMULTANEOUSLY with the train status request a (faulty) request to open the bars reaches the crossing. This branch of the faulty tree will usually only be found with formal FTA. It is not detected with informal FTA.

Informal node	formalization
train at danger spot and crossing not closed	$Pos < ds \wedge ds \leq Pos + V \wedge$ $\neg Closed$
train at danger spot and crossing not closed and no release signal received	$Pos < ds \wedge ds \leq Pos + V \wedge$ $\neg Closed \wedge$ $\neg Release\_Signal\_Rcv$
train at danger spot and crossing not closed and release signal sent	$Pos < ds \wedge ds \leq Pos + V \wedge$ $\neg Closed \wedge$ $Release\_Signal\_Snd$

Table 1
Formalization of fault tree nodes

The resulting proof obligation is then constructed by inserting these formal descriptions of the nodes into the D-OR-gate formula of Fig. 1. The other fault tree gates are handled analogously. The fault tree above has been proven complete. This means that for every gate the corresponding proof obligation has been shown. The conclusion is, that – for this example – all minimal cut sets are single-point-of-failures. So there is no redundancy in the system. On the other hand the fault tree also shows, that if these failures are prevented then the hazard will not occur. In other words if nothing fails, the system will work as intended or even shorter: the system is functionally correct.

4 Lessons learned

As already mentioned in the abstract the here presented case study is to our knowledge the first formal safety analysis of an infinite state system. In this section we will briefly present our experiences with proving FTA formulas over an infinite state model. To prove the correctness of the fault tree, we used KIV [2] as an interactive verification tool.

One big advantage of the KIV system is, that it natively supports state charts as specification mechanism. The state chart model shown in section 3 can be directly used as a system specification in KIV. The proof obligations are derived from the fault tree as shown in the previous section. They can also be generated by the fault tree module of KIV. KIV allows to prove temporal properties with symbolic execution and induction [1]. This means every temporal formula is split into a predicate logic part and some property which must hold from the next step onwards. In practice this results in stepping through all reachable states of the state chart until a loop is found and induction can be applied. State explosion can be avoided by generalization. Generalization means that instead of proofing a formula a more general theorem is proven. The starting formula is then a specialization of the more general theorem. This type of strategy often helps when verifying interactively. Altogether the case study required an effort of about one and a half person months.

We made the following experiences during this case study:

FFTA proofs are easy, but time consuming. Almost all proof steps can be done automatically. Only finding adequate generalizations and identifying the correct inductive argument (i.e. the corresponding state) requires human interaction and skill. In most cases, generalizations can only be found manually. In particular for big proofs it can be very time consuming to find this position (i.e. the part of the proof where a similar subgoal had already been proven) in the proof tree. For locating the correct spot it seems to be possible to use hash functions. This will make state chart proofs much easier and faster.

Generalization are a great help, but are not easy to be found (see above). It is clear that the more generalizations are made in the more possible successor states will be possible in and vice versa. For example if you analyze a deterministic state chart, then with no generalization each step in time will result in exactly one new state. If you generalize this state chart (i.e. you throw away all information on the current) state, then you will get all possible states as possible candidates for the next step in time. In many cases even this "brutal" generalization can be helpful (i.e. if you have to prove that the train moves in one direction). Although you can get as many as 200 case distinctions in your proof, the KIV system can close all of them with its built-in predicate logic simplifier.

This leads to two approaches to prove FTA properties: depth-first-search and breath-first-search. Depth-first-search is more useful as an strategy, if it is unclear if a proof obligation holds or not (i.e. if the nodes of the fault tree have been formalized correctly or not). This is useful in particular to validate a formula and find faults early. Breadth-first search is in general faster, but will only discover specification errors at the very end. But for some properties it is even possible to fully generalize the state of the system and close the proof in one step.

Formalizing FTA nodes is difficult. Even for simple systems it can be very hard to correctly formalize the nodes of the fault tree. This is because the informal understanding of a fault tree (decomposition of causes into components) is not enough for a formal description. This problem can be attenuated if all proof obligation are at the beginning validated with depth-first-search. It is our experience that this additional effort is really worth the time, because formalizing nodes of a fault tree is very error-prone.

5 Conclusion

We showed the first verification of an infinite state system with FTA. Our Experiences show, that formal FTA with interactive verification is a promising, but not an easy topic. Many problems arise from specification errors. These problems may be countered with good methodology. Compared to other formal safety analysis methods, formal FTA is the only one which has a human readable and understandable logic background structure and will thus be more easily accepted in industry than push-the-button techniques (like pure model checking).

References

[1] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction.* PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[2] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 363–366. Springer-Verlag, 2000.

[3] T. Bienmöller, W. Damm, and H. Wittke. The STATEMATE verification environment – making it real. In E. A. Emerson and A. P. Sistla, editors, *CAV'00: 12th international Conference on Computer Aided Verification*, number 1855 in LNCS, pages 561–567, Chicago, IL, USA, 2000. Springer.

[4] 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.

[5] A. Cau, B. Moszkowski, and H. Zedan. *ITL – Interval Temporal Logic.* Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK, 2002. www.cms.dmu.ac.uk/~cau/itlhomepage.

[6] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS' 97*, volume 1536 of *LNCS*, pages 186–238. Springer, 1998.

[7] K. Hansen, A. Ravn, and V. Stavridou. From safety analysis to software requirements. *IEEE Transactions on Software Engineering*, 24(7):573 – 584, July 1998.

[8] K. M. Hansen, A. P. Ravn, and V. Stavridou. From safety analysis to formal specification. ProCoS II document [ID/DTH KMH 1/1], Technical University of Denmark, 1994.

[9] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[10] J. Klose and A. Thums. The STATEMATE reference model of the reference case study 'Verkehrsleittechnik'. Technical Report 2002-01, Universität Augsburg, 2002.

[11] F. Ortmeier, A. Thums, G. Schellhorn, and W.Reif. Combining formal methods and safety analysis – the ForMoSA approach. In *Integration of Software Specification Techniques for Applications in Engineering*. Springer LNCS 3147, 2004.

[12] Frank Ortmeier and Wolfgang Reif. Formal safety analysis of transportation control systems. In *Proceedings of SEFM 2005*, 2005.

[13] G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Proceedings of The Sixth World Conference on Integrated Design & Process Technology*, Pasadena, CA, 2002.

[14] A. Thums. *Formale Fehlerbaumanalyse.* PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German).

[15] A. Thums, G. Schellhorn, and W. Reif. Comparing fault tree semantics. In D. Haneberg, G. Schellhorn, and W. Reif, editors, *FM-TOOLS 2002*, Technical Report 2002-11, pages 25 – 32. Universität Augsburg, 2002.

[16] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook.* Washington, D.C., 1981. NUREG-0492.

[17] Zhou Chaochen and M. R. Hansen. Duration calculus: Logical foundations. In *Formal Aspects of Computing*, pages 283–330, 1997.