Formal Safety analysis of a radio-based railroad crossing using Deductive Cause-Consequence Analysis (DCCA)

Frank Ortmeier, Wolfgang Reif and Gerhard Schellhorn

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg
{ortmeier, reif, schellhorn}@informatik.uni-augsburg.de

Abstract. In this paper we present the formal safety analysis of a radio-based railroad crossing. We use deductive cause-consequence analysis (DCCA) as analysis method. DCCA is a novel technique to analyze safety of embedded systems with formal methods. It substitutes error-prone informal reasoning by mathematical proofs. DCCA allows to rigorously prove whether a failure on component level is the cause for system failure or not. DCCA generalizes the two most common safety analysis techniques: failure modes and effects analysis (FMEA) and fault tree analysis (FTA).
We apply the method to a real world case study: a radio-based railroad crossing. We illustrate the results of DCCA for this example and compare them to results of other formal safety analysis methods like formal FTA.

Key words: formal methods, safety critical systems, safety analysis, failure modes and effects analysis, fault tree analysis, dependability

1 Introduction

The central question of safety analysis is to determine what components of a safety-critical system must fail to allow the system to cause damage. Most safety analysis techniques rely only on informal reasoning which depends heavily on skill and knowledge of the safety engineer. Some of these techniques have been formalized.

In this paper we present a new safety analysis technique: Deductive Cause-Consequence Analysis (DCCA). This technique is a formal generalization of well-known safety analysis methods like FMEA [10], FMECA [4] and FTA [3]. The logical framework of DCCA may be used to rigorously verify the results of these informal safety analysis techniques. It is also strictly more expressive

(in terms of what can by analyzed) than traditional FMEA. We show, that the results of DCCA have the same semantics as those of formal FTA [11]. Because of this DCCA may be used to verify fault trees without formalizing inner nodes of the tree.

In Sect. 2 the semantics of DCCA is presented. A comparison to FMEA and FTA is done in Sect. 3. We illustrate the application of DCCA and report on practical experiences in Sect. 4. In Sect. 5 some related approaches are discussed and Sect. 6 summarizes the results and concludes the paper.

2 DCCA

In this section we describe the formal semantics of DCCA. The formalization is done with Computational Tree Logic(CTL) [5]. We use finite automata as system models. The use of CTL and finite automata allows to use powerful model checkers like SMV [8] to verify the proof obligations.

In the following we assume that a list of hazards on system level and a list of possible basic component failures modes is given. Both data may be collected by other safety analysis techniques like failure-sensitive specification [9] or HazOp [6]. We assume that system hazards H and primary failures δ are described by predicate logic formula. This is true for most practical problems. We call the set of all failure predicates Δ.

2.1 Failure/hazard automata

For formal safety analysis failure modes must be explicitly modeled. We divide the modeling into two steps. First we model the occurrence pattern of the failure mode and second we model the failure mode itself. By "occurrence pattern" we understand how and when the failure mode occurs. For example does the failure mode occur indeterministically (like packet loss in IP traffic) or does it occur once and forever (like a broken switch) or does it occur only during certain time intervals (like until the next maintenance). To model this we use failure automata. Figure 1 shows two such failure automata.

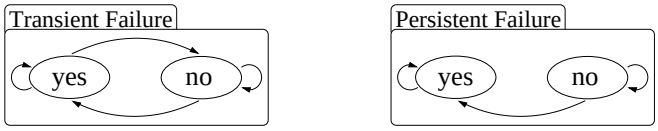

Fig. 1. Failure automata for transient and persistent failures

The left automaton models a transient failure which can indeterministically occur and disappear. The right one models a persistent failure, which happens once and stays forever (e.g. a broken relay). Maintenance etc. may be modeled

analogously. Failure predicates δ are then defined as "failure automaton for failure mode $\tilde{\delta}$ in state *yes*". For readability the symbol δ is used for both the predicate and the automaton describing the occurrence pattern.

The second step is to model the direct effects of failure modes. This is usually done by adding transitions to the model with conditions of the form $\varphi \wedge \delta$. This mean these additional transitions – which reflect erroneous behavior – may only be taken, when a failure automaton is in state *yes* i.e. when a failure occurs.

A similar approach may be used to define predicates for system hazards. If the system hazard can not be describe by a predicate logic formula directly, then often an observer automaton may be implemented such that whenever the automaton is in an accepting state, the hazard has occurred before [12]. This allows to describe the hazard as predicate logic formula on the states of the observer automaton. However in practical applications hazards may usually be described by predicate logic formulas.

2.2 Critical sets

The next step is to define a temporal logic property which says, whether a certain combination of failure modes may lead to the hazard or not. This property is called *criticality* of a set of failure modes.

Definition 1. *critical set / minimal critical set*
For a system SYS and a set of failure modes Δ a subset of component failures $\Gamma \subseteq \Delta$ is called critical for a system hazard, which is described by a predicate logic formula H if

$$SYS \models \mathbf{E}(\overline{\lambda}\ \mathbf{until}\ H) \ \ where\ \overline{\lambda} := \bigwedge_{\delta \in (\Delta \setminus \Gamma)} \neg\, \delta$$

We call Γ a minimal critical set if Γ is critical and no proper subset of Γ is critical.

Here, $\mathbf{E}(\varphi\ \mathbf{until}\ \psi)$ denotes the existential CTL-UNTIL-operator. It means there exists a path in the model, such that φ holds until the property ψ holds. The property *critical set* translates into natural language as follows: there exists a path such that the system hazard occurs without the previous occurrence of any failures except those which are in the critical set. In other words this means, it is possible that the systems fails, if only the component failures in the critical set occur. Intuitively, criticality is not sufficient to define a cause-consequence relationship. It is possible that a critical set includes failure modes, which have nothing to do with the hazard.

Therefore, the notion *minimal critical set* also requires that no proper subset of is critical. Minimal critical sets really describe what one would expect for a cause-consequence relationship in safety analysis to hold: the causes may - but not necessarily - lead to the consequence and second all causes are necessary to allow the consequence to happen. So the goal of DCCA is to find minimal critical sets of failure modes. Testing all sets by brute force would require an

effort exponential in the number of failure modes. However, DCCA may be used to formally verify the results of informal safety analysis techniques. This reduces the effort of DCCA a lot, because the informal techniques often yield good "initial guesses" for solutions. Note that the property critical is monotone with respect to set inclusion i.e. $\forall \Gamma_1, \Gamma_2 \subseteq \Delta : \Gamma_1 \subseteq \Gamma_2 \Rightarrow (\Gamma_1$ is critical set $\Rightarrow \Gamma_2$ is critical set). This helps to reduce proof efforts a lot.

3 Comparison to other safety analysis methods

DCCA formalizes different methods of formal safety analysis in a generic way. We can identify different cases according to the number of elements in the set of failure modes being analyzed and relate them to other existing safety analysis techniques.

$|\Gamma| = 0$

If the empty set of failure modes is examined, then the proof obligation of minimal criticality corresponds to the verification of functional incorrectness. Minimality is of course satisfied (the empty set does not have real subsets). The property of criticality states that there "exists a path where no component fails but eventually the hazard occurs" (in CTL: $EF\ H$). This is the negation of the standard property of functional correctness "on all paths where no component fails, the hazard will globally not occur" (in CTL: $AG\ \neg H$). In other words, if the empty set can be proven to be a critical set, then the system has design errors and is functionally incorrect.

$|\Gamma| = 1$

The analysis of single failure modes corresponds to traditional FMEA. Traditional FMEA analyzes the effects of a component failure mode on the total system in an informal manner. If the failure modes appears to be safety critical than this cause-consequence relationship is noted as one row of a (FMEA) spreadsheet. If a singleton set is minimal critical for a hazard H, then a correct FMEA must list the hazard H as effect of the analyzed failure mode. Note that functional correctness is a pre-condition for formal FMEA. If the system is not functionally correct, then there will be no singleton sets of failure modes which are minimal critical.

$|\Gamma| > 1$

This is a true improvement to FMEA. Combinations of component failure modes are traditionally only examined by FTA. FTA analyzes top-down the reasons of system failure. Cause and consequence are linked by certain *gates*. The *gates* of a fault tree state if all causes (AND-gate ) or any of the causes (OR-gate) are necessary to allow the consequence to occur. Iteration builds a tree like structure where the root is the system hazard and the leaves are component failures.

The result of FTA is a set of so called *minimal cut sets*. These sets may be generated automatically from the structure of the tree. Each *minimal cut set* describes a set of failure modes, which together may make the hazard happen. This corresponds to the definition of *minimal critical sets* obtained by DCCA.

So FTA may be seen as a special case of DCCA. An introduction to FTA may be found in [3].

FTA has been enhanced with formal semantics. Formal FTA [12] allows to decide whether failure modes have been forgotten or not. The idea is to assign a temporal logic formula to each gate. If this formula is proven correct for the system, then the gate is *complete*. This means no causes have been forgotten. An example is given in figure 2.

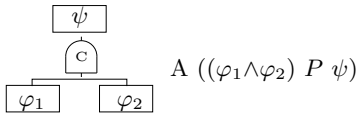

Fig. 2. Fault tree gate and formalization

The figure shows a synchronous cause-consequence AND-gate. The semantics is that both reasons φ_1 and φ_2 must occur simultaneously, before the consequence ψ may occur. Here, $A(\varphi\ P\ \psi)$ denotes the derived CTL-operator *PRECEDES*, which is defined as $\neg E(\neg\varphi\ \mathit{until}\ (\psi \wedge \neg\varphi))$. Informally *PRECEDES* means that whenever ψ holds, φ must have happened before.

Altogether formal FTA distinguishes 7 different types of gates, which reflect temporal ordering, environment constraints and synchronous vs. asynchronous dependencies between cause and consequence. A detailed description of formal FTA may be found in [13].

One of the main results of FTA is the minimal cut set theorem. This theorem states that for a complete fault tree the prevention of only one failure mode of every minimal cut set, assures that the system hazard will never occur. A fault tree is called complete, if all its gates have been proven to be complete.

DCCA may be used to verify the completeness of a fault tree analysis as well. To apply DCCA to FTA we must first introduce the notion of a *complete* DCCA. We call a DCCA complete if all minimal critical sets have been identified.

If a DCCA has been shown to be complete, then it is proven that the minimal critical sets of the DCCA have the same meaning as the minimal cut sets of a fault tree done with formal FTA. In particular the following theorem holds:

Theorem 1. *Minimal critical sets*
For a complete DCCA prevention of one element of every minimal critical set will prevent the hazard H from occurring.

The proof of this theorem is very easy. The statement may be directly derived from the definition of minimal critical sets and the semantics of CTL. In the following SYS denotes a CTL model, s_0 is the initial state of the model, minimal critical sets are called Γ, the set of all minimal critical sets is Ω and individual failure modes are labeled with δ. We define (CTL*-) formulas $[\Gamma]$ for "there exist

a path in the system, such that eventually a critical set of failures Γ occurs" and $[[\Omega]]$ for "there does not exist a path, such that any of the minimal critical sets $\Gamma \in \Omega$ occurs".

$$[\Gamma] := E \bigwedge_{\delta \in \Gamma} F\delta$$

$$[[\Omega]] := \bigwedge_{\Gamma \in \Omega} \neg[\Gamma] = \bigwedge_{\Gamma \in \Omega} \neg E \bigwedge_{\delta \in \Gamma} F\delta$$

With this abbreviations theorem 1 rewrites to $SYS \models [[\Omega]] \Rightarrow SYS \models AG\neg H$. In the following traces of system SYS are called σ and states s_i. The proof for theorem 1 is then as follows:

Proof.

$$\begin{aligned}
Assume: \quad & SYS \not\models AG\neg H \\
\Leftrightarrow\ & SYS, s_0 \not\models AG\neg H \\
\Leftrightarrow\ & SYS, s_0 \not\models \neg EF\neg\neg H \\
\Leftrightarrow\ & SYS, s_0 \models EFH \\
\Leftrightarrow\ & SYS, s_0 \models E(trueUH) \\
\Leftrightarrow\ & \exists \sigma = (s_0, s_1, ...) \in SYS : SYS, \sigma \models (trueUH) \\
\Leftrightarrow\ & \exists i :\ SYS, s_i \models H \wedge \forall j < i :\ SYS, s_j \models true \\
Let \quad & \Gamma_\dagger := \{\delta \in \Omega \mid \exists\, j < i\ : SYS, s_j \models \delta\} \\
\Rightarrow\ & SYS, \sigma \models (\overline{\Gamma_\dagger} UH) \\
\Leftrightarrow\ & \Gamma_\dagger \text{ is critical set} \\
\Rightarrow\ & \exists \tilde{\Gamma}_\dagger \subseteq \Gamma_\dagger :\ \tilde{\Gamma}_\dagger \text{ is minimal critical set} \\
Furthermore \quad & SYS, \sigma \models \bigwedge_{\delta_j \in \Gamma_\dagger} F\delta_j \\
\Rightarrow\ & SYS, \sigma \models \bigwedge_{\delta_j \in \tilde{\Gamma}_\dagger} F\delta_j \\
\Rightarrow\ & SYS, s0 \models E \bigwedge_{\delta_j \in \tilde{\Gamma}_\dagger} F\delta_j \\
\Leftrightarrow\ & SYS \models [\tilde{\Gamma}_\dagger] \\
\Rightarrow\ & SYS \not\models [[\Omega]],\ as\ \tilde{\Gamma}_\dagger \in \Omega,\ \text{because DCCA is complete} \\
\Rightarrow\ & \lightning \quad \square
\end{aligned}$$

For a complete DCCA even the following, stronger result holds:

$$SYS, s_0 \models A((\bigwedge_{\Gamma_i \in \Omega} \neg \bigwedge_{\delta_j \in \Gamma_i} F\delta_j) \rightarrow G\neg H)$$

This is the same property that holds for formal fault tree analysis with the semantics of [12]. However there is a difference as DCCA is more precise. Formal

FTA may yield weaker cut sets than DCCA. For example, assume a systems SYS has two redundant units A and B. The hazard H may only occur if both units fail. So the system has only one minimal critical set of failures: "*A fails AND B fails*". As an intuitively consequence the fault tree in figure 3 is correct and the fault tree in figure 4 is incorrect.

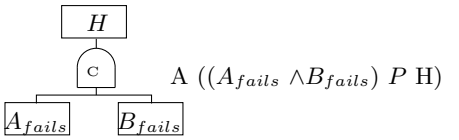

Fig. 3. Correct fault tree

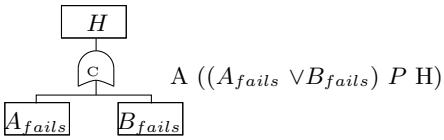

Fig. 4. Incorrect fault tree

With formal FTA both fault trees may be proven to be complete, as both formulas may be proven correct for this system. But the fault tree of figure 3 will yield only one minimal cut set $\Gamma = \{A_{fails}, B_{fails}\}$ while the fault tree of figure 4 will yield two singleton minimal cut sets $\Gamma_1 = \{A_{fails}\}$ and $\Gamma_2 = \{B_{fails}\}$. It is not possible to distinguish the two fault trees with formal FTA. On the other hand DCCA will discover that $\{A_{fails}\}$ resp. $\{B_{fails}\}$ is not critical, because the formula correspondind DCCA formula evaluates to false and thus the sets are not critical. The set $\Gamma = \{A_{fails}, B_{fails}\}$ can be proven to be minimal critical.

The reason for this difference is that formal fault tree semantics does NOT require that all causes must occur before the consequence, but only that prevention of causes prevents the consequence. Here, DCCA yields more precise results than formal FTA. A second advantage is that DCCA does not require inner nodes to be formalized. This is a big advantage in practical applications. Inner nodes are often very hard to formalize. For example an inner node of the fault tree of the example of Sect. 4 is "Release sent and barriers opening". Since "Release *sent*" refers to the past, this is not directly expressible in CTL. This problem was discovered in many case studies and was one of the motivating factors that led to the development of DCCA.

A problem of showing completeness of DCCA is of course the exponential growth of the number of proof obligations. However, only big minimal critical sets will result in a lot of proof effort. In many real applications minimal critical sets are rather small. In addition, informal safety analysis helps to find candidates for minimal cut sets in advance. FTA is one possibility, FMEA is another. This reduces the combinatorial effort of checking all possible sets of failure modes a lot. Finally, monotony of the property *critical* may be exploited; if e.g. a singleton set is minimal critical, then other minimal critical sets must not contain this element.

4 Application

As an example for the application of DCCA we present an analysis of a radio-based railroad crossing. This case study is the reference case study of the german research councils (DFG) priority program 1064. This programs 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: the 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 [7].

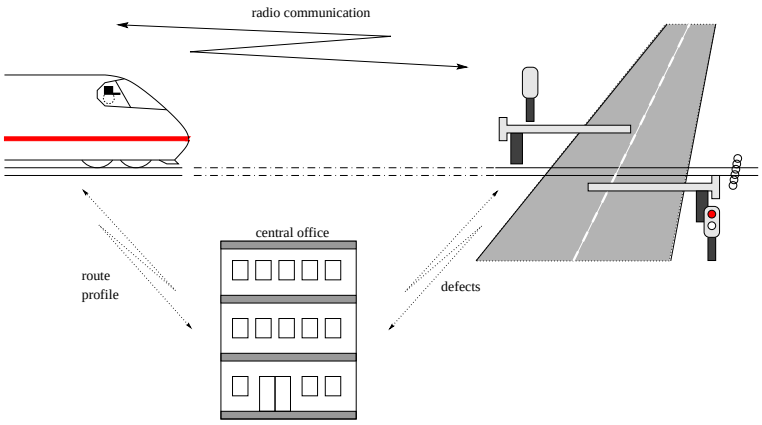

Fig. 5. Radio-based railroad crossing

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. 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, the train computes the position where it has to send a signal to secure the level crossing. 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 also 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 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 is on the crossing and a car is passing the crossing at the same time. A well designed control system must assure this property at least as long as no component failures occur. The corresponding hazard H 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

4.1 Formal model

We now give a brief description of the formal system model. We used SMV [8] as model checker. Altogether the system consists of 16 automata: two automata modeling the control of the crossing and the train, five timer automata, six failure automata and three automata modeling the physics of the train. Altogether the model has roughly 1100 states. In the following we give brief descriptions of the most interesting automata. For better readability - we use a graphical notation instead of SMV input language.

Primary failure and hazards We will now briefly explain the analyzed failure modes and hazards and how they are modeled. The modeling of failure modes generally splits into two different tasks: the modeling of the occurrence pattern and the direct effect of the failure mode. The occurrence pattern describes, when and how the failure occurs resp. when it does not occur. We model occurrence patterns with failure automata (see Sect. 2.1).

In the following we give a summary of the failure modes, which we analyzed. In this example only one hazard is interesting i.e. the train passes an insecure crossing. We call this hazard collision H_{Col}. This is modeled by the following formula:

$$H_{Col} := Pos \leq Pos_{ds} \wedge Pos + Speed > Pos_{ds} \wedge \neg Crossing = closed$$

In this formula Pos_{ds} is an abbreviation for the position of the crossing (ds = danger spot). It describes the location of the crossing. H_{Col} evaluates to true, iff the train passes the crossing and the barriers are not closed. We investigated the following six different types of component failures:

- **Failure of the brakes:** *error_brake* - This error describes the failure of the brakes. It has direct effects on automaton *Dec*.
- **Failure of the communication:** *error_comm* - This error describes the failure of the radio communication. It has direct effects on automata $timer_{close_{rcv}}$, $timer_{status_{rcv}}$, $timer_{ack_{rcv}}$.
- **Failure of the barriers closed sensor:** *error_closed* - This error describes that the crossing signals *closed*, although it is not closed. It has direct effects on automaton *crossing*.
- **Failure of the barriers' actuator:** *error_actuator* - This error describes that the actuator of the crossing fails. It has direct effects on automaton *crossing*.
- **Failure of the train passed sensor:** *error_passed* - This error describes that the sensor detecting trains which passed the crossing fails. It has direct effects on automaton *crossing*.
- **Deviation in the odometer:** *error_odo* - This error describes that the odometer does not give 100% precise data. It has direct effects on automaton *train_control*.

The occurrence of each of these failure modes is modeled by a failure automaton. All failure modes - except *error_actuator* - are assumed to be transient. As abbreviation we write $error\_brake$ for $error\_brake = yes$.

Model of the crossing The automaton in figure 6 shows the model of the crossing. Initially the barriers are *opened*. When the crossing receives a close request from an arriving train - i.e. condition *comm_close_rcv* becomes true, the barriers start *closing*. This process takes some time. This is modeled by timer automaton *timer_closing*. After a certain amount of time the barriers are *closed*. They will remain closed until the train has passed the crossing (detected by a sensor). The barriers reopen automatically after a defined time interval. This is a standard procedure in railroad organization, as car drivers tend to ignore closed barriers at a railroad crossing if the barriers are closed too long. So it is better to reopen the barriers, than having car drivers slowly driving around the closed barriers. The reopening is modeled using another timer automaton *timer_closed*.

A faulty signal from the sensor, which detects when the train has passed the crossing will also open the crossing. This is modeled by $error\_passed = true$. The barriers may get *stuck*, if the actuator fails (*error_actuator*).

Model of the train control 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 7. Starting from its initial state *idle* the automaton goes into state *wfc* ('wait for close'), if the train approaches the crossing (*pos_close_reached*). Simultaneously the train sends a signal requesting to close the barriers.

Some time later the train sends a status request message to the crossing and waits for an answer (state *wfs* - 'wait for status answer'). If the positive answer

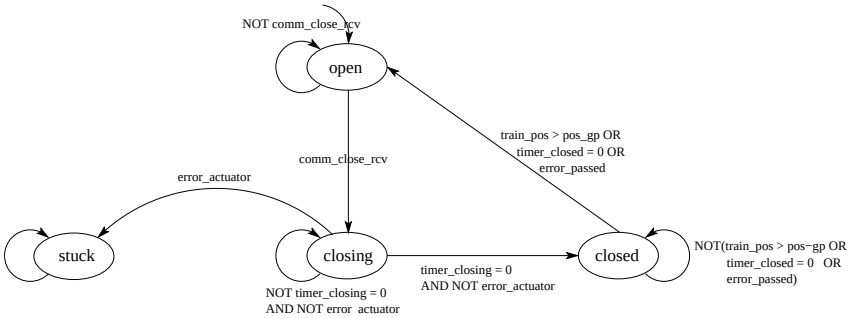

Fig. 6. Model of the crossing

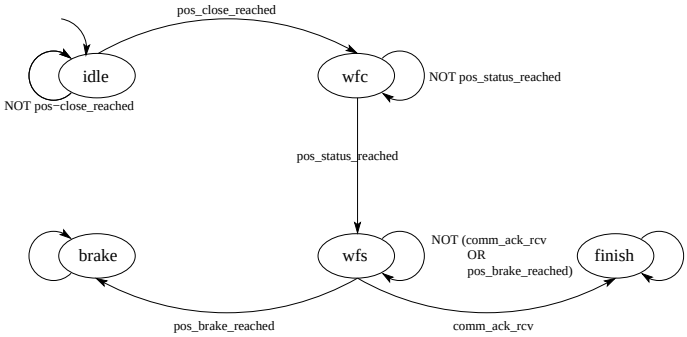

Fig. 7. Model of the train control

reaches the train in time, then the control allows passing the crossing and enters state *finish*. If no acknowledge is received, the control issues an emergency stop (state *brake*). In this case the crossing must be secured manually, before the train may pass the crossing.

The three important predicates *pos_close_reached, pos_status_reached* and *pos_brake_reached* are computed from position and speed data. Possible errors result by deviation of the speed sensor of the train (called odometer). This means the control might calculate braking distances etc. wrongly.

Model of the communication It is part of the case study to take communication delay explicitly into account. To model this three additional timer automata ($timer_{close_{rcv}}, timer_{status_{rcv}}, timer_{ack_{rcv}}$) are built. Timer automata allow to delay certain transitions for n steps and have respectively n states. If, for example, the train sends a close signal to the crossing, timer $timer_{close_{rcv}}$ starts a countdown (in every step it makes transition from state n to state n-1). When finally $timer_{close_{rcv}}$ reaches state 0, the condition *comm_close_rcv* becomes true which means close signal is received by the crossing. Failure of communication pre-

vents the signal *comm_close_rcv* from being received. The others communication request are modeled analogously.

Model of the train The physical train is modeled by three important properties: position, speed and acceleration. To improve readability, we give textual representation of these automata.

The position of the train is given as an integer value between 0 and Pos_{Max}. The automaton modeling the position of the train is defined as follows:

$$
\begin{aligned}
Pos &: 0..Pos_{Max} \\
Pos_{t=0} &:= 0 \\
Pos_{t=n+1} &:= \begin{cases} 0 \text{ or } 1, & if\ Pos_{t=n} = 0 \\ Pos_{t=n} + Speed_{t=n}, & if\ Pos_{t=n} + Speed_{t=n} \leq Pos_{Max} \\ Pos_{Max}, & otherwise \end{cases}
\end{aligned}
$$

This automaton models monotone movement of the train. State *Pos = 0* is an abstraction for "the train has not reached the crossing". At every step in time it is possible that the train either stays absent *Pos = 0* or enters the region in front of the crossing *Pos = 1*. Between 1 and Pos_{max} the train moves according to its speed. When the train reaches the upper bound of Pos_{Max}, we abstract this state to "train has passed the crossing".

The speed of the train is assumed to be constant, unless an emergency break is signaled. This is modeled as follows:

$$
\begin{aligned}
Speed &: 0..Speed_{Max} \\
Speed_{t=0} &:= Speed_{Max} \\
Speed_{t=n+1} &:= \begin{cases} Speed_{t=n} - dec_{t=n}, & if\ Speed_{t=n} - Dec_{t=n} \geq 0 \\ 0, & otherwise \end{cases}
\end{aligned}
$$

For the case study only deceleration is analyzed. The model can however easily extended to acceleration as well. Deceleration is controlled by *TrainControl*. It is by default 0 unless *TrainControl* is in state *brake*. *Error_brake* may prevent braking.

$$
\begin{aligned}
Dec &: 0..Dec_{Max} \\
Dec_{t=0} &:= 0 \\
Dec_{t=n+1} &:= \begin{cases} Dec_{Max}, & if\ TrainControl = brake \wedge \neg error\_actuator \\ 0, & otherwise \end{cases}
\end{aligned}
$$

4.2 DCCA

This model was used to analyze the system with DCCA as described in Sect. 2. All proofs were done using the SMV model checker tool. The proofs took less than 1 minute (for $Pos_{Max} = 1000$ and $Speed_{Max} = 16$).

First we proved that the system is functionally correct. We showed that the empty set of failure modes is not critical. The next step was to examine the singleton sets. We found that {*error_passed*} and {*error_odo*} were the only critical sets. Because the system is functional correct, these two are also minimal critical. To find minimal critical sets with two elements, we had to check only those sets, which do not include {*error_passed*} or {*error_odo*}. So 6 proofs of criticality were needed. Four of the examined sets them were found to be critical. No more sets of failure modes existed, which not already included on of the minimal critical sets. Altogether DCCA yielded the following complete list of minimal critical sets:

- {error_passed}
- {error_odo}[1]
- {error_comm, error_close}
- {error_comm, error_brake}
- {error_close, error_actuator}
- {error_brake, error_actuator}

This example shows that the effort for a complete DCCA does not grow exponentially in real applications, if monotony is used and an adequate methodology is used. This can be quickly computed with set algorithms. In conclusion, by use of monotony instead of 2^6 proof obligations only 13 proofs were necessary to determine all minimal, critical sets.

The results were very surprising for us. We already did a formal FTA with the semantics of [12] for this system using the interactive theorem prover KIV [1]. The fault tree we have proven to be complete consisted of only OR-gates. This means all leaves of the fault tree are single point of failures. DCCA now shows that only *error_passed* and *error_odo* are single points of failure. Other failure modes, which seem to be very safety critical - like for e.g *error_brake* - are only critical in conjunction with other failures. But this result is also intuitively correct. For example if only the brakes fail and everything else works correctly, then the crossing will be secured in time and there will be no need for an emergency stop at all. This means failure of the brakes is NOT a single-point-of-failure for this system. So this example is a proof of concept that the results of DCCA are not only in theory more precise than formal FTA but also in practice.

5 Related Work

There exist some other methods of formally verifying dependencies between component failures and system failure modes. One such technique is formal FTA [12]. Formal FTA requires, that all inner nodes of a fault tree are formalized. This can be very time consuming and difficult (see the example in Sect. 4). A second problem with formal FTA is, that it relies on universal theorems. But, proof

[1] This failure mode is only critical, if the safety margin in the calculation of *pos_closed_reached* is to small.

obligations for gates must be universal, since only universal properties can transitively lead to properties for the whole system. DCCA uses existential proof obligations. This allows to distinguish whether an (failure) event is a necessary or sufficient condition.

Another related approach has been developed in the ESACS project [2]. Here again model checking and FTA is used as basis. The ESACS approach does not require inner nodes of the fault tree to be formalized. However, the approach requires to adjust the model for different proofs. This can be time consuming (building BDDs for model checking is expensive) and

6 Conclusion

We presented a general formal safety analysis technique: DCCA. DCCA is a generalization of the most widely spread safety analysis techniques: FMEA and FTA. In the formal world, verification of functional correctness, formal FMEA and formal FTA may be found as special cases of DCCA. So DCCA may be used to verify different types of safety analysis techniques in a standardized way. The proof obligations of DCCA may be constructed automatically and the proofs can be done - for finite state systems - by model checking.

DCCA formalization is strictly more precise, than other formal formal safety analysis techniques like formal FTA. Theoretically, the effort for DCCA grows exponentially. But we have not found this case to happen in real world applications. The costs are more likely to grow linear (for non redundant systems) or polynomial by n (for systems with n-times redundancy), if monotony is used.

We showed the application of DCCA to a real world case study: the reference case study "radio-based railroad crossing" of german research foundations priority program 1064. DCCA has rigorously identified critical sets of failure modes and the results of the analysis were much more precise than what can be achieved with informal or formal FTA.

References

[1] 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.

[2] P. Bieber, C. Castel, and C. Seguin. Combination of fault tree analysis and model checking for safety assessment of complex systems. In *Dependable Computing EDCC-4: 4th European Dependable Computing Conference*, volume 2485 of *LNCS*, pages 19–31, Toulouse, France, 2002. Springer-Verlag.

[3] J. Fragole J. Minarik II J. Railsback Dr. W. Vesley, Dr. Joanne Dugan. *Fault Tree Handbook with Aerospace Applications*. NASA Office of Safety and Mission Assurance, NASA Headquarters, Washington DC 20546, August 2002.

[4] ECSS. Failure modes, effects and criticality analysis (FMECA). In European Cooperation for Space Standardization, editor, *Space Product Assurance*. ESA Publications, 2001.

[5] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.
[6] T. A. Kletz. Hazop and HAZAN notes on the identification and assessment of hazards. Technical report, The Institution of Chemical Engineers, Rugby, England, 1986.
[7] J. Klose and A. Thums. The STATEMATE reference model of the reference case study 'Verkehrsleittechnik'. Technical Report 2002-01, Universität Augsburg, 2002.
[8] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1990.
[9] F. Ortmeier and W. Reif. Failure-sensitive specification: A formal method for finding failure modes. Technical Report 3, Institut für Informatik, Universität Augsburg, 2004.
[10] Michael R. Beauregard Robin E. McDermott, Raymond J. Mikulak. *The Basics of FMEA*. Quality Resources, 1996.
[11] 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.
[12] A. Thums. *Formale Fehlerbaumanalyse*. PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German), (to appear).
[13] A. Thums and G. Schellhorn. Model checking FTA. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 739–757. Springer-Verlag, 2003.