Using Deductive Cause-Consequence Analysis (DCCA) with SCADE

Matthias Güdemann, Frank Ortmeier, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Güdemann, Matthias, Frank Ortmeier, and Wolfgang Reif. 2007. "Using Deductive Cause-Consequence Analysis (DCCA) with SCADE." In Computer Safety, Reliability, and Security: 26th International Conference, SAFECOMP 2007, Nurmberg, Germany, September 18-21, 2007, proceedings, edited by Francesca Saglietti and Norbert Oster, 465–78. Berlin: Springer. https://doi.org/10.1007/978-3-540-75101-4_44.

Using Deductive Cause-Consequence Analysis (DCCA) with SCADE

Matthias Güdemann, Frank Ortmeier, and Wolfgang Reif

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

Abstract. Esterel Technologies' SCADE Suite is one of the most important development tools for software for safety-critical systems. It is used for designing many critical components of aerospace, automotive and transportation applications. For such systems safety analysis is a key requirement in the development process.

In this paper we show how one formal safety analysis method – Deductive Cause-Consequence Analysis (DCCA) – can be integrated in the SCADE framework. This method allows for performing safety analysis largely automatically. It uses SCADE's semantical model and SCADE's built in verification engine *Design Verifier*. So the whole analysis can be done within one tool. This is of big importance, as a key feature for the acceptance of formal methods in broad engineering practice is, that they can be applied in an industrial development suite.

We illustrate the method on a real world case study from transportation domain and discuss possible next steps and limitations.

Keywords: formal methods, safety critical systems, deductive cause consequence analysis, dcca, safety analysis, dependability, SCADE.

1 Introduction

The role of software in embedded systems development is becoming more important, as more and more features are implemented in software instead of hardware. One example for this are the *x-by-wire* features in automotive application or avionics. Here, safety critical tasks like steering or breaking are controlled by electronic devices which are connected electrically. In contrast to traditional techniques, there no longer exists a physical link between controller and actuator. The result is that, the controlling software now becomes a safety critical system itself, as it takes over a large portion of safety critical tasks.

Systems for avionics must for example be DO178B [14] compliant. To achieve this, software must also be certified. For embedded systems SCADE Suite of Esterel Technologies is a widely accepted, state-of-the-art toolkit which allows to develop software graphically and has a certified development process for safety critical systems. The developed models are automatically converted into executable C code (for various target systems) using a certified code generator.

SCADE Suite also allows to automatically verify safety properties of the developed software using the built-in *Design Verifier* of Prover Technologies.

Traditionally SCADE models are mainly used for designing software. Hardware elements and safety analysis are not being modelled. On the other hand (formal) safety analysis methods need to reason about hardware and software, to give answers like "How many failures can be tolerated?" etc. In the last years a lot of advances in the domain of formal safety analysis have been made like formalisations of fault tree analysis (FTA) [15, 5, 2], fault injection (FI) [1] or the unifying methods of formal cause-consequence analysis (DCCA) [13, 12]. Most of these techniques are semantically grounded on either linear temporal logic (LTL) or computational temporal logic (CTL) and use some variation of finite automata for system modelling. Unfortunately, SCADE does not have proof support for LTL nor for CTL.

In this paper we show how one of the listed methods can be adopted, such that it can be used with SCADE's verification capabilities. This is of big importance for practical applications as it allows system developers to analyse the model which will be used for code generation (and do not need to manually translate it into another modelling language). The paper is structured as follows: Sect. 2 gives an introduction on DCCA. Sect. 3 shows how to conduct DCCA in SCADE. It also sketches a proof idea on why this adaptation of DCCA is semantically correct. Sect. 4 describes a case study and illustrates the application of DCCA in SCADE. Sect. 5 concludes the paper.

2 DCCA

This section describes briefly the formal semantics of DCCA. The formalisation is done with Computational Tree Logic(CTL) [4] using finite automata as system models. The use of CTL and finite automata allows to use powerful model checkers like SMV [9] to verify the proof obligations. The goal of DCCA is the following:

> Given an unwanted, hazardous situation H and a set of component failure modes Γ. Determine, which combinations of failures modes may (1) potentially cause an hazard and (2) are minimal in the sense, such that no proper subset of these failure modes can cause the hazard.

This is the standard question, which most safety analysis methods try to answer. In the following we assume that a set of hazards $\{H\}$ on system level and a set of possible basic component failures Δ modes is given. Both data may be collected by other safety analysis techniques like failure-sensitive specification [11] or HazOp [6].

2.1 Failure/Hazard Automata

For formal safety analysis failure modes must be explicitly modelled. The modelling can be naturally split into two parts. One part is modelling the occurrence pattern of the failure mode and the other is modelling the failure mode itself.

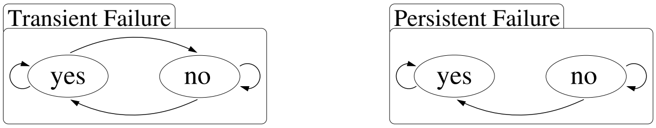

Fig. 1. Failure automata for transient and persistent failures

An "occurrence pattern" describes how and when the failure mode occurs. For example does the failure mode occur nondeterministically (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 failure automata are used. Figure 1 shows two such failure automata.

The left automaton models a transient failure which can nondeterministically 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 modelled 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 of the system with conditions of the form $\varphi \wedge \delta$. This assures, that these additional transitions – which reflect erroneous behaviour – may only be taken, when a failure automaton is in state *yes* i.e. when a failure occurs[1].

A similar approach may be used to define predicates for system hazards. If the system hazard can not be described 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 [15]. This allows to describe the hazard as a predicate logic formula on the states of the observer automaton. However in practical applications hazards may mostly be described by predicate logic formulas.

2.2 Critical Sets

The central part of the analysis is the definition of 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

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

[1] It can also be shown, that the integration of failure modes is monotone – wrt. to traces inclusion to the original model – if a certain set of modelling rules is followed (see [10] for details).

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 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. The goal of DCCA is to find minimal critical sets of failure modes. A DCCA is called complete if all minimal critical sets are found.

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 also, that the property critical is monotone with respect to set inclusion i.e. $\forall \Gamma_1, \Gamma_2 \subseteq \Delta : \Gamma_1 \subseteq \Gamma_2 \Rightarrow (\Gamma_1 \text{ is critical set} \Rightarrow \Gamma_2 \text{ is critical set})$. This helps to reduce proof efforts a lot.

3 DCCA in SCADE

Integrating DCCA in SCADE is not directly possible. This is because of two major problems. Firstly, SCADE does not allow for nondeterministic automata. The idea behind this "feature" is, that the toolkit was traditionally used for software development (where nondeterminism is mostly unwanted). The second (somewhat harder) problem is that SCADE only allows to verify properties of the form "on all paths it is at all times the case that φ holds" or short in CTL notation "AG(φ)". Therefore some adaptations are necessary and integration is only possible for a specific type of failure modes.

3.1 Semantics and Syntax of SCADE Models

The semantics of SCADE models is based on data flows. Each single data flow can be seen as a sequence of values for a variable. The set of all possible data flows defines the semantics of the model. So semantically this model is very similar to the set of traces defined by a Kripke structure.

On the other hand specification in SCADE is very different. Every SCADE model has a fixed set of input and output variables. A syntactic convention is that all inputs are drawn on the left side of a block and all outputs are drawn on the right side. SCADE models are built of blocks. Each block has a fixed input and output interface. Basic blocks are composed to larger models by connecting outputs to inputs. System inputs maybe be connected to any component's input and system outputs can be any component's output. Note that input data is processed *immediately* throughout the whole model. To avoid inconsistencies direct feedback is not allowed[2]. However, there exists a special operator **FBY** whose output is the input of the last step[3]. This operator is often used if data feedback is needed. A simple SCADE model is shown in Fig. 2.

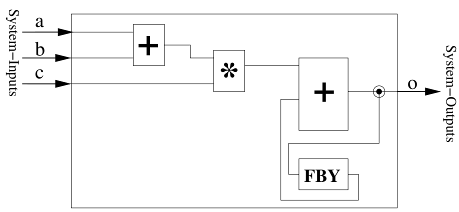

Fig. 2. A simple SCADE model

This model has three inputs a, b and c. The first two inputs are added ("+"-operator) and multiplied ("*"-operator) with the third input. The output o is the accumulation of all previous results ("FBY"-operator). An example data flow $(a, b, c, o)_i$ of the system is: $(1, 2, 3, 9), (1, 2, 2, 15), (2, 2, 2, 23), ...$[4].

SCADE also allows to embed state machines in single blocks. Here, the semantics is that the state machine executes *exactly* one step for every step of the data flow. Note that state machines in SCADE must always be deterministic. So direct modelling of failures as described in Sect. 2.1 is not possible.

3.2 Semantics of DCCA in SCADE

Occurrence patterns of failure modes are modelled in specific failure mode blocks and the hazard is modelled as a SCADE block as well. The output of these blocks is true if the failure mode/hazards occurs and false otherwise.

The basic architecture for applying DCCA to a SCADE model is shown in Fig. 3. Outputs of failure blocks are connected as inputs to the system model. The hazard is connected to the system outputs (outputs may also be some internal variables which are only used for defining the hazard). Failure blocks use as input specific (system) failure inputs.

In Fig. 4 a failure block is shown. The block contains a state machine which is similar to the failure automata shown in Fig. 1. However, there are slight

[2] This is checked by a syntactic analysis.

[3] For the initial state the output of this operator must be defined explicitly (for more details see the documentation of SCADE).

[4] Because: (1+2)*3 + 0=9; (1+2)*2+9=15; (2+2)*2+15=23.

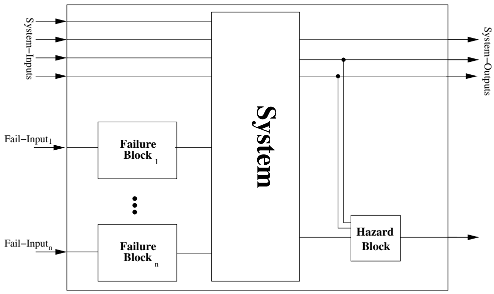

Fig. 3. System model

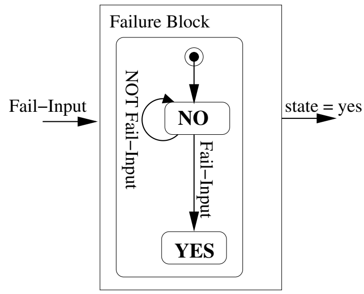

Fig. 4. Failure block

differences. First of all the block has an input, which indicates when the failure starts occurring ($Fail - Input$). This is not redundant as only system inputs are allowed to be nondeterministic in SCADE; state machine must be deterministic. The output of the block evaluates to true if and only if the state machine is in state "yes". This output is then triggering the direct effects of the failure mode as usual. Similar blocks for transient failures are possible.

For DCCA in SCADE two restrictions are necessary. The first one is, that failure blocks must not contain a feedback link from the system. So for example a failure block may not also depend on any of the systems outputs[5]. The second restriction is, that each failure block must have the choice to stay in state "no" whenever it is in state "no". Informally this means: "Every failure happens nondeterministically." The consequence is that at all times there exists some future in which the failure will not occur (if it hasn't already occurred).

It is clear that these two restriction are an abstraction of the real world. However this is not a real problem in the context of safety analysis, because the only relevant question in this domain is, which failures caused hazards (i.e. which components failed before the whole system failed). This approach and the architecture of Fig. 3 lead to the following definition for DCCA:

Definition 2. *critical set / minimal critical set (SCADE)*
For a system SYS and a set of failure mode inputs Δ a subset of failure mode inputs $\Gamma \subseteq \Delta$ is called critical for a system hazard, which is described by an hazard block H if the boolean output flow of block H is not always false *– under the restriction that all failure mode inputs of $\Delta \setminus \Gamma$ are permanently false. It is called minimal critical, if no proper subset of Γ is critical.*

This means in informal language: If no failures of $\Delta \setminus \Gamma$ occur and the hazard H occurs, then the failures modes of Δ are critical. So the process for DCCA is as follows: (1) Add failure mode blocks and hazard blocks, (2) connect failure mode blocks, hazard block and system block as shown in Fig. 3, (3) model direct effects of failure modes (this is analogous to modelling direct effects in finite

[5] An example where this could make sense, is that a crash of an aircraft deterministically triggers the failure mode "loss of power".

automata) and (4) for each set of failure modes check (with SCADE's *Design Verifier*) if the output of the hazard block is always false under the restriction that all other failure mode inputs are false.

3.3 Correctness of DCCA Implementation

This section outlines a sketch of a proof why this adaptation of DCCA is correct. CTL semantics are only where necessary. The following CTL-operators are used:

- "A" denotes the "on-all-paths"-operator
- "E" denotes the "there-exists-a-paths"-operator
- "G" means "at-all-times"
- "F" means "in-some-future-time"

Complete semantics may be found - for example - in [3]. The semantics of the CTL proof obligation for DCCA ($\mathrm{SYS} \models \mathbf{E}(\overline{\Gamma}\ \mathbf{until}\ H)$) is defined as follows:

$$\exists \pi \in \mathrm{SYS} : \exists k \geq 0 : \mathrm{SYS}, \pi_k \models H \text{ and } \forall j : 0 \leq j < k : \mathrm{SYS}, \pi_i \models \overline{\Gamma}$$

This means that there exists a trace π in the system on which at some point in time k the hazard H occurs and on all points in time before no failures in the set $\Delta \setminus \Gamma$ occurred.

As said above, this is not directly expressible in SCADE, as only safety properties of the form "on all paths at all times" can be expressed. Therefore the negated hazard is integrated as proof goal into a verification project as additional boolean output data flow and for every critical set Γ, the range of possible **failure inputs** is restricted, such that all failure inputs of $\Delta \setminus \Gamma$ are permanently set to false. Semantically this means that a system SYS_Γ is used for verification. The formal definition of this system is:

$$\pi \in \mathrm{SYS}_\Gamma := (\pi \in \mathrm{SYS} \wedge \pi \models \mathbf{G}\overline{\Gamma})$$

Here $\overline{\Gamma}$ is defined as in definition 1. The verification goal for showing the criticality of a set of failure modes in SCADE is then:

$$\mathrm{SYS}_\Gamma \not\models \mathbf{AG}\neg H$$

For correctness of this integration it must be shown, that proving the above formula is equivalent to proving the DCCA formula of definition 1. So the following equivalence must hold:

$$\mathrm{SYS}_\Gamma \not\models AG\neg H \Leftrightarrow \mathrm{SYS} \models \mathbf{E}(\overline{\Gamma}\ \mathbf{until}\ H)$$

This means informally: "A set of failure modes can be proven critical in SCADE if and only if it can be proven critical in a corresponding CTL proving environment.” The sketch of the proof is as follows (note that H and $\overline{\Gamma}$ propositional formulas):

$$\begin{gathered}
\mathrm{SYS}_\Gamma \not\models \mathbf{AG}\neg H \\
\Leftrightarrow \\
\exists \pi \in \mathrm{SYS}_\Gamma : \pi \models \mathbf{F}H \\
\Leftrightarrow (*) \\
\exists \pi \in \mathrm{SYS} : \pi \models \mathbf{F}H \text{ and } \pi \models \mathbf{G}\overline{\Gamma} \\
\Leftrightarrow (**) \\
\exists \pi \in \mathrm{SYS} : \pi \models (\overline{\Gamma} \textbf{ until } H) \\
\Leftrightarrow \\
\mathrm{SYS} \models \mathbf{E}(\overline{\Gamma} \textbf{ until } H)
\end{gathered}$$

For proving (*) the definition of SYS_Γ is needed. The downwards equivalence of (**) is trivial. Proof of the upwards direction of $(**)$ is a little tricky. Informally one must show, that for every path (which has a finite prefix where $\overline{\Gamma}$ holds in all states and on which H occurs) there also exists a continuation of this path such that $\overline{\Gamma}$ never occurs.

Formally:

$$\begin{gathered}
\pi \in \mathrm{SYS} : \exists i \in N : \forall j < i \; \pi_j \models \overline{\Gamma} \text{ and } \pi_i \models H \\
\Rightarrow \\
\exists \tilde{\pi} \in \mathrm{SYS} : \forall j \leq i : \pi_i = \tilde{\pi}_i \text{ and } \tilde{\pi} \models \mathbf{G}\overline{\Gamma}
\end{gathered}$$

This implication is true for the considered SCADE models. All variables that appear in the formula $\overline{\Gamma}$ are failure block outputs. More specifically they are of the form “¬(state machine in state yes)”. Initially the state machine is in state “no”. Because of the restrictions described in Sect. 3.2 a transition to state “yes” is only possible if the external input “Fail-Input” is true (see Fig. 4). This input is a nondeterministic system input. Therefore every trace where the state machine is in state “no” can be extended such that the state machine stays always in this state. This closes the proof of the upward direction of (**).

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 program aims at bringing together field-tested engineering techniques with modern methods of the domain of software engineering.

The German railway organisation, Deutsche Bahn, prepares a novel technique to control railroad crossings: the decentralised, 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].

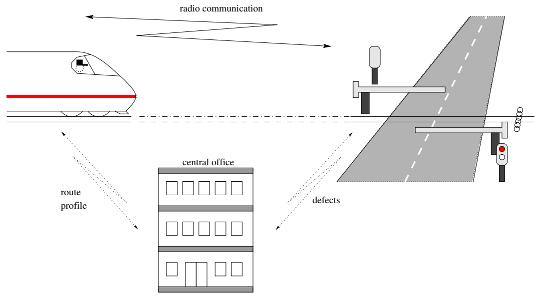

Fig. 5. Radio-based railroad crossing

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 memorised 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 for 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 in SCADE notation. In Fig. 6 the SCADE model of the described system is shown.

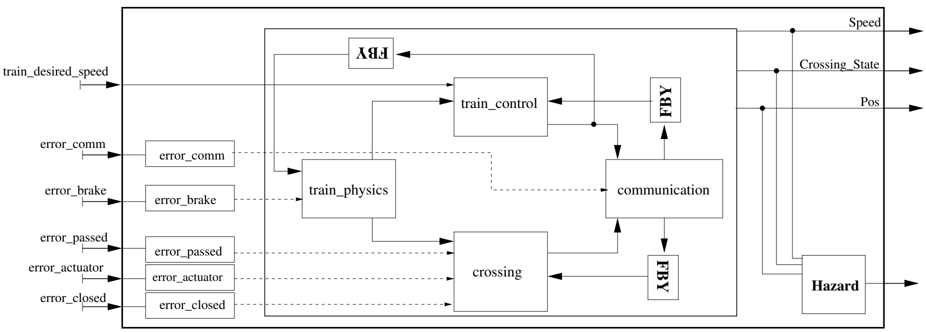

Fig. 6. SCADE model of the system

The system model itself is made of four blocks: one for modelling the physics of the train, one for modelling the railroad crossing, one for the communication between the train and the crossing and one for modelling the control logic of the train. To this system model failure mode blocks and a hazard block have been added.

The following five different types of failure modes were taken into account:

- **Failure of the brakes:** *error_brake* - This error describes the failure of the brakes. It's direct effects are modelled in block *train_physics*.
- **Failure of the communication:** *error_comm* - This error describes the failure of the radio communication. It's direct effects are modelled in block *communication*.
- **Failure of the barriers closed sensor:** *error_closed* - This error describes that the crossing signals *Crossing_Secured*, although it is not closed. It's direct effects are modelled in block *crossing*.
- **Failure of the barriers' actuator:** *error_actuator* - This error describes that the actuator of the crossing fails. It's direct effects are modelled in block *crossing*.
- **Failure of the train passed sensor:** *error_passed* - This error describes that the sensor detecting trains which passed the crossing fails. It's direct effects are modelled in block *crossing*.

The hazard of this system is, that the train passes an insecure crossing. We call this hazard collision H_{Col}. This is modelled by the following formula:

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

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, if and only if the train passes the crossing and the barriers are not closed. In Fig. 7 the SCADE block for this formula is shown.

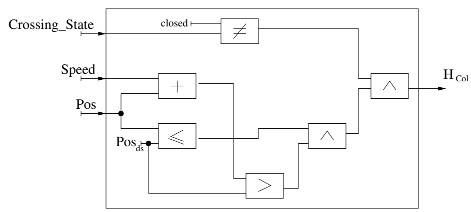

Fig. 7. Hazard block

Because of lack of space we will not show the whole model here, but only describe the block "crossing" in detail.

The block in Fig. 8 shows the model of the crossing. It has inputs for a received closing request (*comm_close_rcv*) and for data of the sensor on the track after the crossing (*sensor_passed*). Additionally, direct effects of failure modes "error_actuator", "error_passed" and "error_close" are modelled in this block. We will not explain how direct failure effects of failure modes can be modelled but refer to [10] where methodology and modelling rules are described.

The component works as follows: Initially the barriers are *opened*. When the crossing receives a close request from an arriving train - i.e. input *comm_close_rcv* becomes true, the barriers start *closing*. This process takes some time. This is modelled by timer block *Timer_Closing*. After a certain amount of time the barriers are *Closed*. They will remain closed until the train has passed the crossing (input *sensor_ passed*). The barriers reopen automatically after a defined time interval. This is a standard procedure in railroad organisation, 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[6]. The reopening is controlled by another timer *Timer_Closed*.

The direct effects of failures which are modelled in this component are marked with grey boxes in Fig. 8. A faulty signal from the sensor, which detects when the train has passed the crossing will also open the crossing. This is modelled by *error_passed*. The barriers may also get *stuck*, if the actuator fails (*error_actuator*).

[6] The target systems for this technologies are railroad crossings in rural areas where only one lane of the road is blocked by a barrier.

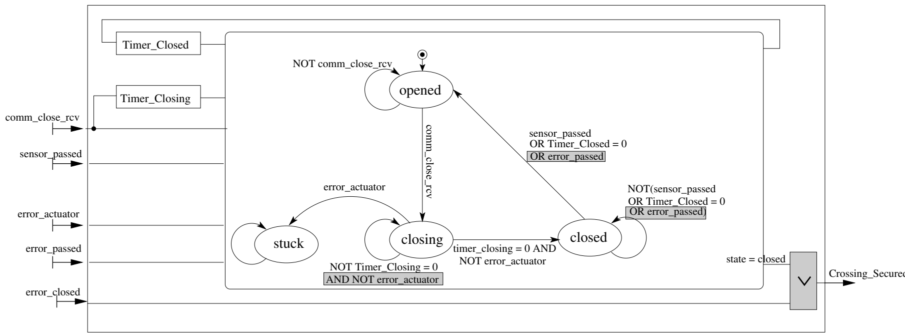

Fig. 8. Model of the crossing

Error_Closed models the failure of the sensor which is detecting the end position of the barrier. This may lead to a scenario, where the crossing signals *Crossing_Secured* although the barriers are not closed. The other components are modelled analogously.

4.2 Results of DCCA

This model was used to analyse the system with DCCA as described in Sect. 3. All proofs were done using the *Design Verifier* model checker from *Prover Technologies* which is integrated into SCADE. This verification engine allows for two different verification strategies: proof and debug. The first strategy can give rigorous proof of a property (by covering the complete state space of the model) while the debug-strategy is much faster but aims at finding counter examples. Unfortunately using the proof strategy for proving sets of failure modes to be non-critical did not work, as even for one failure mode, running time was more than three days. This is because linear time model checking has a higher complexity than for example CTL model checking [8] where the necessary proofs took less than one minute [12] with the SMV model checker.

Criticality of sets of failure modes can be shown with the *debug strategy*. The verification to show that a set is critical was really fast (a couple of seconds to a minute). For all of these critical sets a simulation file is generated which holds the counterexample with values for input variables that lead to the violation of the corresponding proof objective. This file can then be loaded and simulated on the given model to observe its behaviour. The design verifier could prove the following sets to be critical:

- {error_passed}
- {error_comm, error_brake}
- {error_closed, error_actuator}
- {error_brake, error_actuator}

It was not possible (in a runtime of three days before interruption) to show, that the other sets of failure modes are not critical. We did the same analysis with the SMV tool (by translating the model into an equivalent Kripke structure). The result was that (1) all critical sets could also be proven critical with SMV, (2) one additional critical set ({error_comm, error_closed}) was found[7] and (3) that all other sets were not critical. The runtime of SMV is – for this example – very fast. Building the BDD and verifying all 13 proof obligations[8] took less than one minute.

5 Conclusion

Integration of DCCA into an industrial system development environment is possible. However, due to the limited expressiveness of the verification engine restrictions to occurrence patterns of failure modes were necessary. On the other hand the possibility to carry out formal safety analysis without the need to translate system models by hand is a great step forward for integration of formal methods into an industrial development process. We showed the application to a real world case study, presented the results and compared them in terms of time needed for verification to CTL based verification methods.

Using the violation of certain safety properties is a rather natural way of expressing hazards in SCADE. Defining failure modes via nondeterministic failure blocks is a rather general concept for integration of failures into a system model, which integrates smoothly into SCADE. This process can easily be done by an experienced system engineer. The algorithmic capabilities of SCADE's verification engine and the temporal logic available is not as strong as that of state-of-the-art symbolic model checking tools. Nevertheless, formal safety analysis can be done in the framework and yields new results. It can be used within the SCADE framework with only little additional effort and can be used as help for system developers to rate their systems.

Future work will now investigate, if SCADE models can be automatically (and semantically equivalent) translated into Kripke structures. This will allow for much more powerful proof support as well as for more expressive logics.

References

[1] Abdulla, P.A., Deneux, J., Stalmarck, G., Agren, H., Akerlund, O.: Designing safe, reliable systems using SCADE. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, Springer, Heidelberg (2006)
[2] Coppit, D., Sullivan, K.J., Dugan, J.B.: Formal semantics of models for computational engineering: A case study on dynamic fault trees. In: International Symposium on Software Reliability Engineering, IEEE, Los Alamitos (2000)
[3] Peled, D.A., Clarke Jr., E.M., Grumberg, O.: Model Checking. The MIT Press, Cambridge (1999)

[7] For this set of failure modes the Design Verifier repeatedly crashed with an exception. This bug has been fixed now by the developers.

[8] In conclusion, by use of monotony instead of 2^5 proof obligations only 13 proofs were necessary to determine all minimal, critical sets.

[4] Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics, pp. 995–1072. North-Holland Pub. Co./MIT Press (1990)
[5] Górski, J., Wardziński, A.: Formalising fault trees. In: Redmill, F., Anderson, T. (eds.) Achievement and Assurance of Safety, Springer, Heidelberg (1995)
[6] Kletz, T.A.: Hazop and HAZAN notes on the identification and assessment of hazards. Technical report, Inst. of Chemical Engineers, Rugby, England (1986)
[7] Klose, J., Thums, A.: The STATEMATE reference model of the reference case study 'Verkehrsleittechnik'. Technical Report 2002-01, Universität Augsburg (2002)
[8] Kupferman, O., Vardi, M.Y.: Relating linear and branching model checking. Technical Report TR98-301, 18 (1998)
[9] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1990)
[10] Ortmeier, F.: Formale Sicherheitsanalyse. PhD thesis, Universität Augsburg (in German) (2006)
[11] Ortmeier, F., Reif, W.: Failure-sensitive specification: A formal method for finding failure modes. Technical Report 3, Institut für Informatik, Universität Augsburg (2004)
[12] Ortmeier, F., Reif, W., Schellhorn, G.: Formal safety analysis of a radio-based railroad crossing using deductive cause-consequence analysis (DCCA). In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, Springer, Heidelberg (2005)
[13] Ortmeier, F., Reif, W., Schellhorn, G.: Deductive cause-consequence analysis (DCCA). In: Proceedings of IFAC World Congress, Elsevier, Amsterdam (2006)
[14] RTCA. DO-178B: Software considerations in airborne systems and equipment certification (December 1st 1992)
[15] Thums, A.: Formale Fehlerbaumanalyse. PhD thesis, Universität Augsburg, Augsburg, Germany (in German) (2004)