Fault-aware modeling and specification for efficient formal safety analysis

Axel Habermaier, Alexander Knapp, Johannes Leupolz, Wolfgang Reif

Fault-Aware Modeling and Specification for Efficient Formal Safety Analysis

Axel Habermaier(✉), Alexander Knapp, Johannes Leupolz, and Wolfgang Reif

Institute for Software and Systems Engineering,
University of Augsburg, Augsburg, Germany
{habermaier,knapp,leupolz,reif}@isse.de

Abstract. Deductive Cause Consequence Analysis (Dcca) is a model checking-based safety analysis technique that determines all combinations of faults potentially causing a hazard. This paper introduces a new fault modeling and specification approach for safety-critical systems based on the concept of fault activations that decreases explicit-state model checking and safety analysis times by up to three orders of magnitude. We augment Kripke structures and LTL with fault activations and show how standard model checkers can be used for analysis. Additionally, we present conceptual changes to Dcca that improve efficiency and usability. We evaluate our work using our safety analysis tool S# ("safety sharp").

1 Introduction

Safety-critical systems have the potential to cause hazards, i.e., situations resulting in economical or environmental damage, injuries, or loss of lives [17]. *Deductive Cause Consequence Analysis* (Dcca) is a model-based safety analysis technique [7,12] that uses model checking to compute how faults such as component failures or environmental disturbances (the causes) can cause such hazards (the consequences): From a model of a safety-critical system that describes the system's nominal behavior as well as the relevant faults, Dcca determines all *minimal critical fault sets*, i.e., the smallest possible combinations of faults that can cause hazards, allowing the evaluation of the system's overall safety. Dccas are conducted automatically by tools like S# [13] or Vecs [20]; the Fsap/Compass toolsets [6,7], AltaRica [4], and *BT Analyser* [19] perform similar safety analyses. Alternatively, compositional safety analysis approaches and tools such as HiP-HOPS or *Component Fault Trees* [10,26] allow for faster analyses at the expense of requiring explicitly modeled fault propagations between components. Model checking-based techniques, by contrast, deduce these propagations automatically, albeit requiring additional states and transitions for each fault which reduce model checking efficiency exponentially [7,12].

This paper's first contribution is a fault-aware modeling and specification approach for safety-critical systems that decreases explicit-state analysis times by up to three orders of magnitude. We augment Kripke structures and *Linear*

Temporal Logic (LTL), making them aware of *fault activations* [2] within the analyzed systems: Faults are activated when they can in fact influence the system's behavior, preventing model checkers from considering possibly many situations with irrelevant active faults during analyses. We show how the extended formalisms can be mapped back to the classical ones for analysis with standard model checkers such as LTSmin [18]. We demonstrate the efficiency improvements over the traditional fault modeling approach, showing that explicit-state model checking becomes feasible for safety-critical systems that incorporate faults.

The second contribution is a conceptual change of Dcca formalized using LTL instead of *Computation Tree Logic* (CTL). It improves the model checking workflow as witnesses are generated to explain how critical fault sets *can* cause a hazard, which is more useful in practice than witnesses showing how non-critical fault sets *cannot* do so. We also formalize another Dcca variant that reduces analysis times in many cases.

2 Model-Based Safety Analysis

Throughout this section, we assume models of safety-critical systems to be given as Kripke structures $K = (P, S, R, L, I)$ consisting of a set of atomic propositions P, a set of states S, a left-total transition relation $R \subseteq S \times S$, a labeling function $L\colon S \rightarrow 2^P$, and a non-empty set of initial states $I \subseteq S$ [9]. Kripke structures are a well-known modeling formalism that established model checkers such as LTSmin, Spin, or NuSMV [8,14,18] are based on. Consequently, tools built for formal safety analyses [6,7,13,19,20] either implicitly or explicitly transform their models to Kripke structures for model checking, while their actual modeling formalisms are higher-level.

Figure 1 gives a description of the running example used throughout the paper that is based on the fault tree handbook's pressure tank case study [28]. The system is safety-critical because of the hazard of tank ruptures that might injure nearby people. Ruptures only happen when *both* suppression faults $\neg\mathsf{is\ full}$ and $\neg\mathsf{timeout}$ occur; consequently, there is only one minimal critical fault set for the hazard that consists of these two faults. For more complex systems with more faults, however, minimal critical fault sets are not as easily deduced. Instead, model-based safety analysis techniques such as Dcca are required to compute these critical sets automatically and rigorously.

2.1 Fault Terminology

Safety analyses consider situations in which faults cause system behavior that would not have occurred otherwise. In accordance with common terminology [2], these situations represent *fault activations*; that is, a fault is *activated* when it influences the system, affecting its behavior or state in a concealed or observable way. Faults are *dormant* until they are *activated* and become *active*, turning dormant again when they are *deactivated*. A fault's *persistence* constrains the

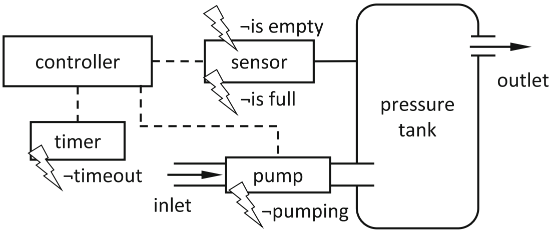


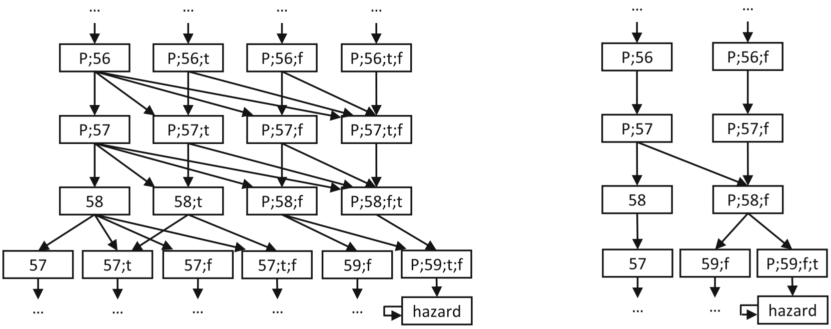
Fig. 1. A schematic overview of the running example: the fluid contained in the pressure tank is refilled by the pump that is activated and deactivated by a controller. The pressure sensor signals the controller when the pressure limit is reached or the tank is empty, causing the controller to deactivate or activate the pump, respectively; once the tank is empty, a new refill cycle begins. To tolerate pressure sensor faults, the controller disables the pump after 60 s of continuous operation as it would risk a tank rupture otherwise. For time measurements, the controller uses the timer. The pressure sensor has two suppression faults: It might not report that the pressure limit is reached (¬is full) or that the tank is empty (¬is empty). The timer might not signal a timeout (¬timeout) and a fault of the pump (¬pumping) prevents it from filling the tank.

transitions between its active and dormant states. *Transient* faults, for instance, are activated and deactivated completely nondeterministically, whereas *permanent* faults, while also activated nondeterministically, never become dormant again. Fault activations result in *effects* that change the internal state of affected components, thereby causing *errors*. Errors are deviations of the components' states from what they should have been. Errors *propagate* through the components, causing other errors. Eventually, errors might result in *failures* where the errors manifest themselves in a way that is externally observable. Failures either provoke faults in other components or they represent system *hazards*; safety analyses are conducted for the latter to determine all faults causing them.

2.2 State-Based Fault Modeling

Vecs, Compass, Fsap, and other safety analysis tools [6,7,19,20] share a common, state-based fault modeling approach: For each modeled fault, the tools' high-level modeling formalisms require at least one additional Boolean variable where changes of the variable's value represent fault activations and deactivations. These variables increase both the number of reachable states as well as the number of transitions of the Kripke structures generated from the high-level models [6,12]. Transient faults represent the worst case as they occur completely nondeterministically: n additional transient faults increase the generated Kripke structure's reachable state space by a factor of 2^n and each state has an additional 2^n successor states. Permanent faults, by contrast, have an overall lower number of possible successor states compared to transient faults, so the amount of reachable states and transitions might not increase as noticeably; model checking and safety analysis efficiency is reduced significantly with each additional fault in both cases.

The running example's pressure sensor modeled with S# in Listing 1, for instance, is a high-level representation that can either be checked using the classical state-based fault modeling approach or the fault-aware one introduced in this paper. In the former case, the Kripke structure generated for the model would be similar to the one shown in Fig. 2(a), whereas fault-aware modeling would eventually result in a significantly smaller classical Kripke structure similar to the one in Fig. 2(b). Figure 2(a) shows a part of the running example's Kripke structure shortly before the tank is fully filled and either the pump is shut off or the tank ruptures. After 56 s of pumping, neither fault has any observable effect on the system. During the next step, only the activation of ¬is full has an observable effect, namely that pumping continues even though it should have stopped. If pumping is not stopped by the sensor, the pump is shut off only if ¬timeout is not activated, otherwise the tank ruptures. The Kripke structure shown in Fig. 2(b) can be seen as an abstraction of the one in Fig. 2(a). It unifies states that are equivalent modulo active faults, thereby reducing both the state and the transition counts significantly; the states where ¬is full is active cannot be unified due to the cyclic nature of the model. The Kripke structure is *minimal* in the sense that irrelevant active faults are omitted while all *system* states remain reachable, including, in particular, the hazard. The notion of minimality is based on the observation that the exact points in time in which faults become active are irrelevant as long as they do so before or when they can affect the system. Inspired by partial order reduction [3], fault-aware modeling and specification is a fundamental change of model-based safety analysis that inherently considers only minimized Kripke structures similar to the one in Fig. 2(b).



(a) The Kripke structure resulting from state-based fault modeling has redundant states and transitions where faults are active without any observable effects.

(b) The activation-minimal Kripke structure has no state or transition redundancy by considering relevant active faults only.

Fig. 2. Partial view of the running example's Kripke structures resulting from state-based fault modeling (a) or fault-aware modeling (b). States are labeled with P when the pump is running; the number represents both the tank's pressure level and the timer's counter. The faults ¬is full and ¬timeout are active in states that show their respective labels f and t. For reasons of brevity, the other two faults are omitted and ¬is full and ¬timeout are assumed to be permanent.

3 Fault-Aware Modeling and Specification

Instead of the commonly used state-based fault modeling approach, we focus on fault activations, making them central to the models and specifications of safety-critical systems as well as the safety analysis techniques. Our approach is *event*-based where events are fault activations and deactivations; we exclusively consider the former only, as we have not yet found a use case that requires the latter. The significant state and transition count reductions demonstrated by Fig. 2(b) make the potential advantages for model checking efficiency evident. We augment the classical notion of Kripke structures and LTL to incorporate fault activations explicitly, allowing us to more conveniently formalize Dcca, fault injection, and fault removal in the remainder.

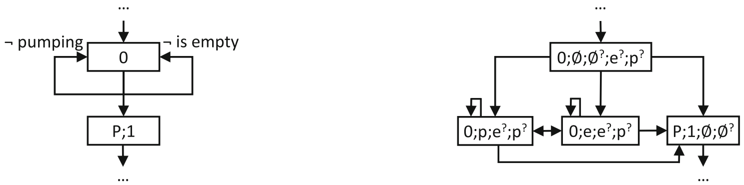
3.1 Fault-Aware Kripke Structures

Fault-aware Kripke structures explicitly denote the faults that can affect the system they represent. They highlight states in which faults can be activated by labeling their outgoing transitions with sets of activated faults as can be seen in Fig. 3(a): The transition relation of the running example's fault-aware Kripke structure is *activation-minimal* in the sense that no transitions can be removed without affecting the Kripke structure's behavior or losing system states; in particular, there are no transitions labeled with $\neg$is full or $\neg$timeout between the two shown states as these two faults obviously cannot be activated when the tank is empty. The Kripke structure in Fig. 2(a), on the other hand, has many superfluous states and transitions that can safely be removed without losing any system states or behavior. The actual state and transition count reductions made possible by activation minimality depend on how often a fault can be activated: $\neg$timeout, for instance, is only activatable right before the hazard occurs, resulting in a significant state space reduction; $\neg$pumping, by contrast, is activatable in roughly 50 % of all states and therefore does not profit as much from fault-aware modeling and specification.

Definition 1 (Fault-Aware Kripke Structures). *A* fault-aware Kripke structure $K = (P, F, S, R, L, I)$ *consists of a set of* atomic propositions P*; a set of* faults F*; a set of* states S*; a* transition *relation* $R \subseteq S \times 2^F \times S$ *labeled with fault activations that is*

- left-total*, i.e.,* $\forall s \in S \,.\, \exists s' \in S, \Gamma \subseteq F \,.\, (s, \Gamma, s') \in R$ *and*
- activation-minimal*, i.e.,* $(s_1, \Gamma, s_2) \in R \wedge (s_1, \Gamma', s_2') \in R \wedge \Gamma \subsetneq \Gamma' \rightarrow s_2 \neq s_2'$*;*

a labeling *function* $L : S \rightarrow 2^P$ *indicating the set of atomic propositions holding in a state; and a non-empty set of* initial fault activations and states $\emptyset \neq I \subseteq 2^F \times S$ *that is also* activation-minimal*, i.e.,* $(\Gamma_1, s_1) \in I \wedge (\Gamma_2, s_2) \in I \wedge \Gamma_1 \subsetneq \Gamma_2 \rightarrow s_1 \neq s_2$*. We also write* $P(K)$ *for* P*,* $F(K)$ *for* F*, etc. A fault-aware Kripke structure* K *is* finite *if* $P(K)$*,* $F(K)$*, and* $S(K)$ *are finite.*



(a) A fault-aware Kripke structure explicitly labels its transitions with the minimal amount of possible fault activations to avoid any state or transition redundancy.

(b) The classical Kripke structure generated from the fault-aware one to the left requiring additional states labeled with actual and potential fault activations.

Fig. 3. Part of the running example's fault-aware Kripke structure where the tank is empty, which the sensor should report to start the pump. Activations of either ¬pumping or ¬is empty prevent the system from doing so. State label P indicates that the pump is running; the number represents the pressure level. Labels e and p indicate activations of ¬is empty and ¬pumping, respectively, whereas e? and p? denote potential activations of these faults when leaving the state.

For a fault-aware Kripke structure K, a *path fragment* $\varsigma = \Gamma_0 s_0 \Gamma_1 s_1 \ldots$ of K is an infinite, alternating sequence of fault activations $\Gamma_i \subseteq F(K)$ and states $s_i \in S(K)$ such that $(s_i, \Gamma_{i+1}, s_{i+1}) \in R(K)$ for all $i \geq 0$. We write $\varsigma[i]$, $\varsigma_{\mathrm{F}}[i]$, and $\varsigma_{\mathrm{S}}[i]$ for (Γ_i, s_i), Γ_i, and s_i, respectively. A *path* of K is a path fragment ς of K with $\varsigma[0] \in I(K)$; the set of all paths of K is denoted by $\mathit{paths}(K)$. The *reachable* states $\mathcal{R}(K)$ are given by $\{\varsigma_{\mathrm{S}}[i] \mid \varsigma \in \mathit{paths}(K) \wedge i \geq 0\}$. The following notion of path equivalence *modulo* faults Γ allows us to compare the paths of two fault-aware Kripke structures, ignoring activations of faults $f \in \Gamma$:

Definition 2 (Path Equivalence). *Two fault-aware Kripke structures K_1 and K_2 are* path equivalent modulo faults F, *denoted as $K_1 \equiv_F K_2$, if for all $\varsigma = \Gamma_0 s_0 \Gamma_1 s_1 \ldots$ with $s_i \in S(K_1) \cup S(K_2)$, $\Gamma_i \subseteq F(K_1) \cup F(K_2)$, and $\Gamma_i \cap F = \emptyset$ for all $i \geq 0$, $\varsigma \in \mathit{paths}(K_1)$ if and only if $\varsigma \in \mathit{paths}(K_2)$. $K_1 \equiv_\emptyset K_2$ is abbreviated as $K_1 \equiv K_2$.*

In order to use standard model checkers such as LTSmin, a fault-aware Kripke structure $K = (P, F, S, R, L, I)$ can be converted to a classical Kripke structure $K' = (P', S', R', L', I)$: We encode *actual* and *potential* fault activations into atomic propositions Γ and $\Gamma^?$, respectively; the latter indicates that the faults Γ are activated by at least one outgoing transition of a state. We have $P' = P \cup 2^F \cup \{\Gamma^? \mid \Gamma \subseteq F\}$; $S' = 2^F \times S$; $R' = \{((\Gamma, s), (\Gamma', s')) \mid (s, \Gamma', s') \in R\}$; and $L'(\Gamma, s) = \Gamma \cup L(s) \cup \{\Gamma'^? \mid \exists s' \in S \,.\, (s, \Gamma', s') \in R\}$. While S' is much larger than S, most additional states are not reachable due to activation minimality and are thus irrelevant for explicit-state model checkers such as LTSmin; with state-based fault modeling, even more additional and often superfluous states would be introduced, most of which would be reachable and thus slow down model checking unnecessarily. Figure 3(b) shows the classical Kripke structure generated from the fault-aware one in Fig. 3(a). The additional states are required to support fault-aware LTL; they are unavoidable without fault-aware model checkers.

3.2 Fault-Aware Linear Temporal Logic

Fault-aware Kripke structures only model fault activations, disregarding any persistence constraints. Instead, we assume the constraints to be encoded into the checked LTL formulas to filter out paths violating any of them. The following definition of fault-aware LTL is based on the classical variant with both future- and past-time operators [3,21]. The past modalities do not increase the expressiveness of the logic, but make some formulas exponentially more succinct [21] while in many practical cases still allowing for efficient model checking [27]. Compared to classical LTL, fault-aware LTL provides two additional operators related to fault activations: Formula $\varGamma$ requires that at least the faults in $\varGamma$ were activated to reach a state, that is, it checks whether a state was reached because of the activations of all $f \in \varGamma$, and potentially more, during the last transition. Formula $\varGamma$ therefore allows a glimpse into the immediate past, whereas the other new operator supported by fault-aware LTL conceptually looks into the immediate future: Formula $\varGamma^?$ checks whether exactly the fault set $\varGamma$ might potentially be activated when leaving a state, i.e., it allows to check whether precisely the faults $f \in \varGamma$ can be activated to reach the next state. The $\varGamma^?$ operator therefore considers multiple distinct futures that are possible instead of one single future as is usually the case with LTL; the operator is conceptually similar to **EX** in CTL. Fault-aware LTL is unable to directly express that a fault is active or dormant, which we found of little practical use.

Definition 3 (Fault-Aware LTL). *Fault-aware LTL formulas $\varPhi$ over a set P of atomic propositions and a set F of faults are formed according to the following grammar, where φ, φ_1, and φ_2 are fault-aware LTL formulas over P and $F, p \in P$, and $\varGamma \subseteq F$:*

$$\varPhi ::= \mathrm{true} \mid p \mid \varGamma \mid \varGamma^? \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\,\varphi \mid \varphi_1 \mathbf{U} \varphi_2 \mid \mathbf{P}\,\varphi \mid \varphi_1\,\mathbf{S}\,\varphi_2$$

Propositional connectives are defined as usual; we write $\mathbf{F}\,\varphi$ for $\mathrm{true}\mathbf{U}\varphi$, $\mathbf{G}\,\varphi$ for $\neg\,\mathbf{F}\,\neg\varphi$, $\mathbf{O}\,\varphi$ for $\mathrm{true}\,\mathbf{S}\,\varphi$, and $\mathbf{H}\,\varphi$ for $\neg\,\mathbf{O}\,\neg\varphi$. Additionally, $\varphi_1\,\mathbf{U}^{=}\,\varphi_2$ abbreviates $\varphi_1\mathbf{U}(\varphi_1 \wedge \varphi_2)$. A fault-aware LTL formula $\varphi \in \varPhi$ is valid *at a position $i \geq 0$ of a path fragment ς of a fault-aware Kripke structure K, written as $\varsigma, i \models \varphi$, if:*

$\varsigma, i \models \mathrm{true}$ $\qquad$ $\varsigma, i \models p$ *iff* $p \in L(K)(\varsigma_{\mathrm{S}}[i])$

$\varsigma, i \models \varGamma$ *iff* $\varGamma \subseteq \varsigma_{\mathrm{F}}[i]$

$\varsigma, i \models \varGamma^?$ *iff* $(\varsigma_{\mathrm{S}}[i], \varGamma, s) \in R(K)$ *for some* $s \in S(K)$

$\varsigma, i \models \neg\varphi$ *iff* $\varsigma, i \not\models \varphi$ $\qquad$ $\varsigma, i \models \varphi_1 \wedge \varphi_2$ *iff* $\varsigma, i \models \varphi_1$ *and* $\varsigma, i \models \varphi_2$

$\varsigma, i \models \mathbf{X}\,\varphi$ *iff* $\varsigma, i + 1 \models \varphi$

$\varsigma, i \models \varphi_1\mathbf{U}\varphi_2$ *iff there is a* $k \geq i$ *with* $\varsigma, k \models \varphi_2$ *and* $\varsigma, j \models \varphi_1$ *for all* $i \leq j < k$

$\varsigma, i \models \mathbf{P}\,\varphi$ *iff* $i > 0$ *and* $\varsigma, i - 1 \models \varphi$

$\varsigma, i \models \varphi_1\,\mathbf{S}\,\varphi_2$ *iff there is a* $k \leq i$ *with* $\varsigma, k \models \varphi_2$ *and* $\varsigma, j \models \varphi_1$ *for all* $k < j \leq i$

$\varsigma \models \varphi$ abbreviates $\varsigma, 0 \models \varphi$. φ is valid in K, written as $K \models \varphi$, if $\varsigma \models \varphi$ for all $\varsigma \in \mathit{paths}(K)$.

For all faults $f \in F$, we generally require that they do not have to be activated in an initial state, i.e., $K \not\models f$ with f abbreviating $\{f\}$. Fault-aware Kripke structures violating this assumption always start with at least one activated fault, making their adequacy questionable. Additionally, a transient fault $f \in F$ must be activated completely nondeterministically. The corresponding persistence constraint is $\mathbf{G} \bigvee_{\Gamma' \subseteq F \setminus f} \Gamma'^{?}$: There always is a transition where f is not activated; otherwise, f's activation would be deterministically enforced. Permanent faults $\Gamma \subseteq F$ behave like transient ones until the first time they are activated: $\bigwedge_{\Gamma' \subseteq \Gamma} \mathbf{G}((\Gamma'^{?} \wedge \bigwedge_{f \in \Gamma'} \mathbf{O}\, f) \rightarrow \mathbf{X}\, \Gamma')$ ensures that all subsets $\Gamma' \subseteq \Gamma$ of the faults are indeed activated whenever they are activatable and all faults $f \in \Gamma'$ have already been activated at least once.

To determine whether a formula $\varphi \in \Phi$ holds for a fault-aware Kripke structure K with $K \not\models f$ for all $f \in F(K)$, we check $K \models (\bigwedge_{\psi \in \Psi} \psi) \rightarrow \varphi$ for a set of persistence constraints Ψ; such formulas are similar to fairness conditions in that they can only be expressed in LTL but not in CTL [3]. The extended formula might result in a more complex Büchi automaton; however, constraints for transient faults are both the common and the general case with a simple single-state Büchi representation. Therefore, transient faults no longer represent the worst case as with state-based fault modeling, but the best case instead. The transformation of fault-aware LTL to classical LTL is straightforward by making formulas Γ and $\Gamma^{?}$ propositions; the Kripke structures generated from fault-aware ones contain the required state labels for Γ and $\Gamma^{?}$.

3.3 Fault Injection

The intended behavior of a safety-critical system is commonly modeled first with the faulty behavior injected later in a separate step [6]. State-based fault modeling requires additional states, labels, and transitions for injected faults just to distinguish the faults' active and dormant states. For fault-aware Kripke structures, however, injecting a fault can only add new transitions when the fault is actually activated; additional states, labels, and transitions are only required to model a fault's effects on the system. Formally:

Definition 4 (Fault Injection). Injecting *the faults* F' *into* $K = (P, F, S, R, L, I)$ *yields the set of* extended *fault-aware Kripke structures* $K \lhd F'$, *where for all* $K' = (P', F \cup F', S', R', L', I') \in K \lhd F'$, $P \subseteq P'$, $S \subseteq S'$, $L(s) \subseteq L'(s)$ *for all* $s \in S$, $R \subseteq R'$ *such that for all* $(s, \Gamma, s') \in R' \setminus R$, $s \in S \rightarrow \Gamma \cap F' \neq \emptyset$, *and* $I \subseteq I'$ *such that for all* $(\Gamma, s) \in I' \setminus I$, $\Gamma \cap F' \neq \emptyset$.

The definition reflects the fact that there are many possible ways of injecting faults F' into a model by yielding a *set* of extended fault-aware Kripke structures incorporating F'. For reasons of adequacy, the model including the faults is required to be an *extension* of the model without these faults; that is, fault injection may add but can never remove behavior. We can formally show that the original Kripke structure and all possible extensions are path equivalent as long as the injected faults are never activated:

Proposition 1 (Conservative Extension). *For a fault set F and fault-aware Kripke structures K and $K_F \in K \lhd F$, $K \equiv_F K_F$.*

Fault injection is purely additive, as new behavior can only be reached by activations of injected faults; until then, the system behaves as before. The high-level models of the safety analysis tools typically guarantee conservative extension syntactically [6,13].

4 Deductive Cause Consequence Analysis

By model checking a series of formulas, DCCA uncovers cause consequence relationships between faults (the causes) and hazards (the consequences): For each hazard, DCCA computes all *minimal critical* fault sets $\varGamma$ that can cause the occurrence of the hazard. We assume a finite fault-aware Kripke structure K representing the system to be analyzed with a hazard H given as a propositional logic formula over K not referencing any faults $f \in F = F(K)$. A fault set $\varGamma$ is *critical* for a hazard H if and only if there is the possibility that H occurs and before that, at most the faults in $\varGamma$ have been activated. The following *original* definition [12] of the criticality property uses CTL, which could be extended to fault-aware Kripke structures similar to fault-aware LTL. The usage of CTL, however, limits DCCA's applicability to Kripke structures with a *single* initial state, since a CTL formula is only valid on a Kripke structure if it holds in *all* initial states.

Definition 5 (Minimal Critical Fault Sets). *Let $|I(K)| = 1$. A fault set $\varGamma \subseteq F$ is* critical *for hazard H if $K \models (only_\varGamma \, \mathbf{EU}^= H)$, where $only_\varGamma :\Leftrightarrow \bigwedge_{f \in F \setminus \varGamma} \neg f$. A critical fault set $\varGamma$ is* minimal *if no proper subset $\varGamma' \subsetneq \varGamma$ is critical.*

The CTL formula checks whether there is a path $\varsigma \in paths(K)$ on which the hazard H occurs and before that, none of the faults $f \notin \varGamma$ are activated; conversely, at most the faults $f \in \varGamma$ are activated. The use of the $\mathbf{EU}^=$ operator guarantees that the transition leading to the state where the hazard occurs still enables at most the faults $f \in \varGamma$; if $\mathbf{EU}$ was used instead, there could be activations of faults $f \in F \setminus \varGamma$ right before the hazard occurs, which is obviously unintended. Activation is only (implicitly) required for minimal criticality, but not for criticality: Any superset $\varGamma' \supseteq \varGamma$ of a critical fault set $\varGamma$ is also critical as additional fault activations cannot be expected to fix other faults and thus to improve safety. In practice, the criticality's *monotonicity* with respect to set inclusion [12] often allows for significant reductions in the number of checks required to find all minimal critical fault sets; otherwise, *all* subsets of F would have to be checked for criticality. DCCA's worst-case complexity, however, is in fact exponential.

DCCA determines all minimal critical fault sets for a hazard; it is a *complete* safety analysis technique [12] in the sense that the hazard cannot occur as long as at least one fault of each minimal critical fault set is never activated. Formally, $K \models (\bigwedge_{\varGamma \in \varLambda} \neg \bigwedge_{f \in \varGamma} \mathbf{F}\, f) \rightarrow \mathbf{G}\, \neg H$ always holds where $\varLambda$ is the set of all minimal critical fault sets for H determined by conducting a DCCA for H over K. DCCA is always complete, regardless of whether faults were injected into the Kripke structure in accordance with Definition 4 or added in some other, arbitrary way. Adequacy, however, is only guaranteed for fault-aware Kripke structures with injected faults. Only then is the check for criticality of the empty fault set equivalent to a check of functional correctness, namely whether the hazard can already occur even without any fault activations.

4.1 Conceptual Improvement: Safe Fault Sets

Similar to fairness conditions, persistence constraints cannot be enforced while checking the criticality of a fault set due to the use of CTL [3]. We therefore base DCCA on the dual of critical fault sets instead, which can indeed be formalized using fault-aware LTL:

Definition 6 (Safe Fault Sets). *A fault set* $\Gamma \subseteq F$ *is* safe *for hazard* H *if and only if* $K \models \neg(only_\Gamma\ \mathbf{U}^{=}\ H)$ *with* $only_\Gamma$ *as in Definition 5.*

A fault set Γ is considered safe for a hazard H if it is impossible that at most the activations of faults $f \in \Gamma$ result in an occurrence of H. Intuitively, a fault set should be critical if and only if it is not safe. This correlation, however, does *not* hold as criticality assumes Kripke structures to have a single initial state only, whereas for safe fault sets, we do not make this assumption. For multiple initial states, the original definition would classify critical sets as safe in certain cases. To establish the correlation, we from now on consider a fault set $\Gamma \subseteq F$ to be critical for H if and only if there is some path $\varsigma \in paths(K)$ with $\varsigma \models only_\Gamma\ \mathbf{U}^{=}\ H$. The monotonicity property and DCCA's completeness guarantee continue to hold for the new definition of criticality; the proofs are similar to the original ones [12]. Additionally, for any safe fault set Γ, any subsets $\Gamma' \subseteq \Gamma$ are safe as well: Under no circumstances is it possible that less fault activations make a system less safe.

DCCA is model checker-independent, whereas other techniques similar to DCCA are tied to certain model checkers such as NUSMV or *BT Analyser* [5,7,19]. In contrast to DCCA, some of these techniques [5,7] are only able to assume monotonicity or require permanent persistence [19]; with DCCA, monotonicity is guaranteed regardless of the analyzed model and all kinds of persistence constraints are supported. Using safe fault sets to conduct DCCAs results in three notable usability improvements: Firstly, DCCA now supports multiple initial states, avoiding workarounds that construct a unique pseudo initial state by changing both the model and the analyzed formula. Ironically, LTSMIN only supports a unique initial state, forcing us to work around this limitation nevertheless. Secondly, both LTL and CTL can be used to conduct DCCAs, enabling broader model checker support; with CTL, a fault set is safe if and only if $K \models \neg(only_\Gamma\ \mathbf{EU}^{=}\ H)$. Thirdly, the model checker now generates a counter example when a fault set Γ is not safe, i.e., it constructs a witness that explains why Γ is critical and how it causes the hazard. Consequently, the model checking

$$\begin{tikzcd}
K \arrow[r, Rightarrow, "K \lhd F_1 \uplus F_2"] \arrow[d, Rightarrow, "K \lhd F_1"'] & K_{F_1 \uplus F_2} \arrow[d, Rightarrow, "K_{F_1 \uplus F_2} \mathbin{\setminus\!\!\setminus} F_2"] \\
K_{F_1} \arrow[r, dash, "\equiv"'] & (K_{F_1 \uplus F_2})_{\setminus F_2}
\end{tikzcd}$$

Fig. 4. Illustration of the relation of fault injection and fault removal that enables fault removal Dcca: starting with a fault-aware Kripke structure K and two disjoint fault sets F_1 and F_2, path equivalent fault-aware Kripke structures can be obtained by either injecting the relevant faults F_1 only or by first injecting all faults $F_1 \uplus F_2$ and subsequently removing the irrelevant faults F_2.

workflow is improved as witnesses for safe fault sets showing how a hazard is *not* caused are typically of no interest in practice.

4.2 Efficiency Improvement: Fault Removal

Formal safety analysis tools can automatically conduct complete Dccas. Instead of using a series of LTL formulas to check for safe faults sets within a model, the tools could alternatively change the model to make the checks more efficient: Faults $F \setminus \Gamma$ are not allowed to be activated during a check of Γ due to $only_\Gamma$, so they could just as well be *removed* from the model entirely as outlined by Fig. 4, reducing the model's state space. Thus, instead of checking multiple formulas on the same model, the same simplified formula can be checked for multiple reduced models. We formalize a fault removal variant of Dcca based on this idea, which generalizes an ad hoc approach to conduct Dccas within the Scade tool [11]. We first define the notion of *activation independence* that is required to show that Dcca always computes the same minimal critical fault sets, regardless of whether the multiple formulas or multiple models approach is used. We consider a fault set to be activation independent if activations are never forced, that is, there is always an alternative future or initial state in which none of the faults are activated; trivially, transient and permanent faults are activation independent.

Definition 7 (Activation Independence). *A fault set $\Gamma \subseteq F(K)$ of a fault-aware Kripke structure K is* activation independent *in K if $K \not\models \Gamma \neq \emptyset \land \Gamma$ and $K \models \mathbf{G} \bigvee_{\Gamma' \subseteq F(K) \setminus \Gamma} \Gamma'^{?}$.*

In particular, the following set of *reduced* fault-aware Kripke structures is non-empty if and only if an activation independent fault set is removed, as activation independence preserves left-totality for all reachable states $\mathcal{R}(K)$ during fault removal:

Definition 8 (Fault Removal). Removing *the fault set $F' \subseteq F$ from a fault-aware Kripke structure $K = (P, F, S, R, L, I)$ yields the set of* reduced *fault-aware Kripke structures $K \mathbin{\setminus\!\!\setminus} F'$, where for all $K' = (P', F \setminus F', S', R', L', I') \in K \mathbin{\setminus\!\!\setminus} F'$, $P' \subseteq P$, $\mathcal{R}(K) \subseteq S' \subseteq S$, $R' = \{(s, \Gamma, s') \in R \mid s, s' \in S' \land \Gamma \cap F' = \emptyset\}$, $L'(s) = L(s)$ for all $s \in S'$, and $I' = \{(\Gamma, s) \in I \mid s \in S' \land \Gamma \cap F' = \emptyset\}$.*

In contrast to fault injection which is a creative activity, fault removal is mechanic and can therefore be done automatically by a tool. All $K' \in K \setminus\!\!\setminus F$ have identical sets of paths as they can only differ in irrelevant details such as unreachable states or transitions. Similar to fault injection, fault-aware Kripke structures are path equivalent before and after fault removal as long as the removed faults are never activated:

Proposition 2. *For fault set $F \subseteq F(K)$ and fault-aware Kripke structures K and $K_{\setminus F} \in K \setminus\!\!\setminus F$, $K \equiv_F K_{\setminus F}$.*

In general, we can only infer that $K \setminus\!\!\setminus \Gamma \neq \emptyset$ when removing an activation independent fault set $\Gamma \subseteq F(K)$ from a fault-aware Kripke structure K. When removing previously injected faults, we trivially obtain a fault-aware Kripke structure that is path equivalent to the original one. Fault-aware LTL can also be used to effectively remove an activation independent fault set $\Gamma' \subseteq F(K)$ from a fault-aware Kripke structure K, similar to how persistence constraints suppress undesirable fault activations or deactivations: For $K_{\setminus \Gamma'} \in K \setminus\!\!\setminus \Gamma'$ and $\varphi \in \Phi$ expressible over both K and $K_{\setminus \Gamma'}$, $K \models (\mathbf{G} \bigwedge_{f \in \Gamma'} \neg f) \rightarrow \varphi$ if and only if $K_{\setminus \Gamma'} \models \varphi$ by Proposition 2. In particular, while conducting a Dcca, all faults whose activations are suppressed by the $only_\Gamma$ part of the safe fault sets formula can be removed from the checked model, thereby replacing checks of multiple LTL formulas on a model with all faults by a series of reachability checks of multiple reduced models:

Theorem 1 (Fault Removal Dcca). *Let K be a fault-aware Kripke structure with faults $F = F(K)$, $\Gamma \subseteq F$ be a fault set, and $K_{\setminus (F \setminus \Gamma)} \in K \setminus\!\!\setminus (F \setminus \Gamma)$. Γ is safe for hazard H if and only if $K_{\setminus (F \setminus \Gamma)} \models \mathbf{G} \neg H$.*

The proof of Theorem 1 generalizes and completes the one given for Scade-Dcca [11]. Overall potential analysis time reductions depend on the model adaptation overhead and the size of the minimal critical fault sets; as the latter are usually rather small, efficiency can improve significantly. The following proposition establishes the adequacy of the fault removal optimization for injected, activation-independent faults $\Gamma' \subseteq F(K)$, i.e., the criticality of $\Gamma = F(K) \setminus \Gamma'$ can be determined by either removing Γ' or by injecting only Γ in the first place. That is, it is not even necessary to construct the complete fault-aware Kripke structure containing all analyzed faults and subsequently to remove the faults that the analyzed Dcca formula would suppress anyway; instead, only the analyzed faults can be injected into the model, avoiding any potential analysis tool overhead when carrying out the fault removals. More generally, the following proposition summarizes the formal justification for the equivalence at the bottom of Fig. 4, allowing only the smaller fault set to be injected in the first place; a similar result can be obtained for the reverse direction.

Proposition 3. *For fault sets F and $F' \subseteq F$ and all fault-aware Kripke structures K, $K_F \in K \triangleleft F$, and $(K_F)_{\setminus F'} \in K_F \setminus\!\!\setminus F'$, there is a $K_{F \setminus F'} \in K \triangleleft (F \setminus F')$ such that $K_{F \setminus F'} \equiv (K_F)_{\setminus F'}$.*

5 Tool Support and Evaluation

Fault-aware modeling and specification is implemented in the S# modeling and analysis framework for safety-critical systems [13]. The following gives a brief overview of S#'s high-level modeling language, its fault modeling capabilities, and its integration with the explicit-state model checker LTSmin [18]. S#'s efficiency is contrasted with the explicit-state and symbolic model checkers Spin and NuSMV [8,14] as well as the safety analysis tools Vecs and Compass [20,23]. In particular, we highlight S#'s analysis efficiency improvements over previous versions of S# for three case studies which result from the use of fault removal Dcca as well as fault-aware Kripke structures. The latest version of S# as well as documentation about its installation and usage are available at http://safetysharp.isse.de. Detailed descriptions of the case studies as well as the S# case study models are also available there, including interactive, S#-based visualizations that support visual replays of model checking counter examples.

5.1 The S# Modeling and Analysis Framework for Safety-Critical Systems

The S# modeling and analysis framework conducts Dccas fully automatically for system models authored in the ISO-standardized C# programming language and .Net runtime environment [15,16]. Its modeling language and the underlying model of computation put particular emphasis on flexible system design variant modeling and composition capabilities as well as support for fault modeling and automated fault injection which guarantees conservative extension. While S# models are *represented* as C# programs, they are still models of the safety-critical systems to be analyzed; the running example's tank, for instance, is part of the model even though it is not software-based in the real world. Even the software parts of S# models are not intended to be used as the actual implementations; these are typically done in C or C++ for reasons of efficiency. Thus, S# is best regarded as an executable, text-based extended subset of SysML [24] even though no automated transformations between the two exist. The underlying model of computation is a series of discrete system steps, where each step takes a clock tick. As shown by Listing 1, S# components are represented by C# classes, instances of which correspond to S# component instances. Methods are considered to be either required or provided ports; inheritance, interfaces, generics, lambda functions, etc. are fully supported by S#. To instantiate a model, the appropriate component instances must be created, their initial states and subcomponents must be set, and their required and provided ports must be connected.

```
class PressureSensor : Component { // incomplete due to space restrictions
   int Max;
   public PressureSensor(int max) { Max = max; }

   public extern int GetPhysicalPressure();
   public virtual bool MaxReached() { return GetPhysicalPressure() >= Max; }

   PermanentFault NotIsFull = new PermanentFault();

   [FaultEffect(Fault = nameof(NotIsFull))]
   class NotIsFullEffect : PressureSensor {
      public override bool MaxReached() { return false; }
   }
}
```

Listing 1. A partial S# model of the running example's pressure sensor. The provided port `MaxReached` checks the required port `GetPhysicalPressure` against the `Max` value set via the constructor to determine whether the maximum pressure level is reached. The permanent fault ¬is full is represented by `NotIsFull`; its effect is modeled by the nested class `NotIsFullEffect` marked with the `FaultEffect` attribute that links the effect to the fault. The effect overrides the original behavior of `MaxReached` such that it always returns `false` when the fault is activated, regardless of the actual pressure level; the port's original implementation is invoked only when the fault is dormant.

S#'s unified model execution approach [13] integrates the explicit-state model checker LTSmin [18]: Instead of model transformations typically employed by safety analysis tools such as Vecs, Compass, and AltaRica [4,20,23], S# unifies simulations, visualizations, and fully exhaustive model checking by executing the models with consistent semantics regardless of whether a simulation is run or some formula is model checked. During model checking, all combinations of nondeterministic choices and fault activations within a model are exhaustively enumerated, the generated transitions are minimized with regard to the faults they activate, and a fault-aware Kripke structure is generated on-the-fly and subsequently transformed into a classical one for LTSmin. However, S# is not a software model checker such as Java Pathfinder or Zing [1,29] as it does not analyze states after every instruction; only state changes between individual, more coarse-grained system steps are considered. Additionally, heap allocations or threads are unsupported during model checking.

5.2 Evaluated Case Studies

S#'s analysis efficiency with fault removal Dcca and fault-aware Kripke structures is evaluated with three case studies. The first two case studies were previously analyzed using hand-written NuSMV models [12,25]; the safety analysis results obtained with S# match those from previous analyses, the main improvements over them lie in S#'s modular, high-level modeling language and flexible model composition capabilities based on C# that, for instance, no longer require manual work for composing system design variants. Additionally, S#'s unified model execution approach not only generates and checks the required Dcca formulas fully automatically, but also allows for interactive visualizations and visual replays of model checking counter examples based on the same underlying S# model without sacrificing analysis efficiency unacceptably.

Radio-Controlled Railroad Crossing. The radio-controlled railroad crossing replaces sensors on the track by onboard computations of the train position and radio-based communication between the train and the crossing [12]. The hazard is a train passing an unsecured crossing, potentially resulting from faults such as lost communication messages or the crossing's barrier getting stuck. Being model checking-based, DCCA is automatically able to cope with temporal dependencies inherent to the case study: Simultaneous occurrences of multiple faults might be safe, while consecutive occurrences might not due to the communication interplay between the train and the crossing.

Height Control System. The height control system [25] of the Elbe Tunnel in Hamburg, Germany, tries to prevent overheight vehicles from entering the tunnel at unsuitable locations to avoid collisions with the tunnel's ceiling. The antagonistic hazards of collisions and false alarms must be balanced, taking failures of various sensors into account. The system's design space is restricted by the physical properties of the sensors as well as the road layout; the "best" designs strike a balance between the two aforementioned hazards. S# supports modular modeling of different design variants and their composition in order to analyze the safety of all modeled design variants.

Hemodialysis Machine. The third case study is a hemodialysis machine [22], consisting of several physical components such as tubing valves, pumps, drip chambers, and the dialyzer itself. To adequately express the causal dependencies between these components, it is necessary to model the fluid flows that interconnect them. The analyzed hazard is that of contaminated blood entering the patient's vein.

5.3 Evaluation Results

S#'s latest version makes use of fault removal DCCA and automatically generates fault-aware Kripke structures for explicit-state model checking with LTSMIN. Compared to previous versions of S# that employed explicit-state model checking with state-based fault modeling, analysis efficiency improves by up to almost four orders of magnitude depending on the case study as outlined by Table 1. S# is generally faster than the established explicit-state model checker SPIN: In the worst case of valid formulas where the model's entire state space must be enumerated, S# and LTSMIN take 68.8 s for the height control model. SPIN, by contrast, takes 553 s to check a hand-optimized, non-modular version of the model that semantically corresponds to the S# version. On a quad-core CPU, LTSMIN achieves a speedup of 3.7x, bringing the analysis time down to 18.6 s whereas SPIN scales by a factor of 1.5x only. Fault awareness makes S# more efficient than SPIN, causing it to compute less transitions while still finding all reachable states. For the height control case study, activation minimality is partially encoded into the SPIN model; general and automated support would require changes to SPIN's model checking algorithms, however.

For the height control case study, BDD-based symbolic analysis with NUSMV is faster than using S#: For a hand-written, very low-level and

Table 1. The results of the S#-based evaluation of the explicit-state efficiency improvements, comparing fault-aware modeling and specification with state-based fault modeling in (a). A comparison of both DCCA variants is shown in (b). Three S# case studies were evaluated on a 3.4 GHz quad-core CPU: The height control system (T), the radio-controlled railroad crossing (R), and the hemodialysis machine (H), checking the hazards of tunnel collisions, trains on unsecured crossings, and contaminated blood entering the patient's vein, respectively.

	State-Based			Fault-Aware			
	States	Trans	Time	States	Trans	Time	
T	249	1219724	1.5d	1.3	57	14.2s	9127x
R	116	10564	12m	2.5	8.5	1.9s	379x
H	0.6	152	3m	0.05	0.9	10.1s	18x

(a) Comparison of both fault modeling approaches. The "States" columns show the models' approximate amount of reachable states in millions, "Trans" the approximate amount of reachable transitions in millions, and "Time" the time to enumerate all states. The last column shows the analysis speed-up.

	Fault-Aware				
	Faults	MCS	Std Time	FR Time	
T	11	3	3010s	33.1s	91x
R	7	6	23s	1.4s	16x
H	8	4	1040s	15.9s	65x

(b) Comparison of both DCCA variants. "Faults" lists the number of faults, "MCS" the amount of minimal critical sets. The time columns show the times of standard and fault removal DCCA; the speed-up is shown on the right.

non-modular NuSMV model that is approximately equivalent to the S# model, the entire state space is generated almost instantly, despite state-based fault modeling. By contrast, the railroad crossing case study is more efficiently checked by S# or Spin than by NuSMV, so the relative efficiency of explicit-state and symbolic model checking is case study-specific and independent from S#. In general, highly nondeterministic models seem to profit more from symbolic techniques. Fault awareness can partially be encoded into NuSMV models using input variables [5], slowing down analysis noticeably in some cases, however.

6 Conclusion and Future Work

Fault-aware modeling and specification of safety-critical systems has two main advantages over the commonly used state-based fault modeling approach: Explicitly denoting faults and their activations simplifies the descriptions and formal definitions of safety analysis techniques like DCCA and of safety-related concepts such as fault injection and fault removal. Moreover, model checking efficiency in general and safety analysis times in particular are improved significantly such that explicit-state model checking becomes competitive with symbolic techniques when analyzing safety-critical systems. Some case studies still have higher analysis times with S# compared to NuSMV; this tradeoff seems acceptable, however, when considering the step-up in modeling flexibility and expressiveness as well as the guarantees of semantic consistency and conservative fault injection that S# provides over Spin, NuSMV, or, in parts, Vecs. Compared to other approaches for safety modeling and analysis like Compass, Vecs, AltaRica, or HiP-HOPS, S# has a competitive edge by tightly integrating the development, debugging, and simulation of models with their formal analysis with no or only minor sacrifices in analysis efficiency. In general, however, fair comparisons between these tools and S# are hard to achieve due to their different models of computation. For instance, it took us about 740 lines to create a scaled down COMPASS version of the railroad crossing model that is semantically similar to the S# version written in 400 lines of C# code. COMPASS performs a safety analysis that is equivalent to DCCA in 21 min using NUSMV instead of the 1.4 s it takes S# to do the same. Of course, the comparison is unfair as forcing COMPASS semantics onto S# might likewise slow down analyses.

While S# and DCCA only compute the minimal critical fault sets for a hazard, the actual hazard probability is also of interest. We are therefore working on fault-aware probabilistic model checking; preliminary results are promising, bringing the achieved analysis time reductions for non-probabilistic analyses to probabilistic ones. Additionally, we plan to explore (semi-)automatic abstractions from irrelevant environment states to decrease analysis times similar to partial order reduction [3]: S# models always contain parts of the system's physical environment for reasons of adequacy [13], for which the system's sensors might readily serve as abstraction functions.

References

1. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: a model checker for concurrent software. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 484–487. Springer, Heidelberg (2004)
2. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. Dependable Secure Comput. **1**(1), 11–33 (2004)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Batteux, M., Prosvirnova, T., Rauzy, A., Kloul, L.: The AltaRica 3.0 project for model-based safety assessment. In: Industrial Informatics, pp. 741–746. IEEE (2013)
5. Bozzano, M., Cimatti, A., Griggio, A., Mattarei, C.: Efficient anytime techniques for model-based safety analysis. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 603–621. Springer, Heidelberg (2015)
6. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: The COMPASS approach: correctness, modelling and performability of aerospace systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 173–186. Springer, Heidelberg (2009)
7. Bozzano, M., Cimatti, A., Tapparo, F.: Symbolic fault tree analysis for reactive systems. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 162–176. Springer, Heidelberg (2007)
8. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
9. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008)

10. Grunske, L., Kaiser, B.: An automated dependability analysis method for COTS-based systems. In: Franch, X., Port, D. (eds.) ICCBSS 2005. LNCS, vol. 3412, pp. 178–190. Springer, Heidelberg (2005)
11. Güdemann, M., Ortmeier, F., Reif, W.: Using deductive cause-consequence analysis (DCCA) with SCADE. In: Saglietti, F., Oster, N. (eds.) SAFECOMP 2007. LNCS, vol. 4680, pp. 465–478. Springer, Heidelberg (2007)
12. Habermaier, A., Güdemann, M., Ortmeier, F., Reif, W., Schellhorn, G.: The ForMoSA approach to qualitative and quantitative model-based safety analysis. In: Railway Safety, Reliability, and Security, pp. 65–114. IGI Global (2012)
13. Habermaier, A., Knapp, A., Leupolz, J., Reif, W.: Unified simulation, visualization, and formal analysis of safety-critical systems with S#. In: ter Beek, M., Gnesi, S., Knapp, A. (eds.) FMICS-AVoCS 2016. LNCS, vol. 9933, pp. 150–167. Springer, Heidelberg (2016)
14. Holzmann, G.: The SPIN Model Checker. Addison-Wesley, Reading (2004)
15. ISO: ISO/IEC 23270: Information technology - Programming languages – C# (2006)
16. ISO: ISO/IEC 23271: Information technology – Common Language Infrastructure (2012)
17. ISO/IEC/IEEE: ISO 24765: Systems and software engineering – Vocabulary (2010)
18. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015)
19. Kromodimoeljo, S., Lindsay, P.A.: Automatic Generation of Minimal Cut Sets. In: Engineering Safety and Security Systems, pp. 33–47 (2015)
20. Lipaczewski, M., Struck, S., Ortmeier, F.: Using tool-supported model based safety analysis – progress and experiences in SAML development. In: High-Assurance Systems Engineering, pp. 159–166. IEEE (2012)
21. Markey, N.: Temporal logic with past is exponentially more succinct. In: EATCS Bulletin, vol. 79, pp. 122–128. European Association for Theoretical Computer Science (2003)
22. Mashkoor, A.: The hemodialysis machine case study. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 329–343. Springer, Heidelberg (2016). doi:10.1007/978-3-319-33600-8_29
23. Noll, T.: Safety, dependability and performance analysis of aerospace systems. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2014. CCIS, vol. 476, pp. 17–31. Springer, Heidelberg (2015)
24. Object Management Group: OMG Systems Modeling Language (OMG SysML), Version 1.4 (2015)
25. Ortmeier, F., Schellhorn, G., Thums, A., Reif, W., Hering, B., Trappschuh, H.: Safety analysis of the height control system for the Elbtunnel. In: Anderson, S., Bologna, S., Felici, M. (eds.) SAFECOMP 2002. LNCS, vol. 2434, pp. 296–308. Springer, Heidelberg (2002)
26. Papadopoulos, Y., Walker, M., Parker, D., Rüde, E., Hamann, R., Uhlig, A., Grätz, U., Lien, R.: Engineering failure analysis and design optimisation with HiP-HOPS. Eng. Fail. Anal. **18**(2), 590–608 (2011)
27. Pradella, M., San Pietro, P., Spoletini, P., Morzenti, A.: Practical model checking of LTL with past. In: Automated Technology for Verification and Analysis (2003)
28. Vesely, W., Dugan, J., Fragola, J., Minarick, J., Railsback, J.: Fault tree handbook with aerospace applications. Technical report, NASA, Washington, DC (2002)
29. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)