OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

Qualitative and quantitative analysis of safety-critical systems with S

Johannes Leupolz, Alexander Knapp, Axel Habermaier, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Leupolz, Johannes, Alexander Knapp, Axel Habermaier, and Wolfgang Reif. 2018. "Qualitative and quantitative analysis of safety-critical systems with S#." International Journal on Software Tools for Technology Transfer 20 (4): 359–77. https://doi.org/10.1007/s10009-017-0464-3.

Nutzungsbedingungen / Terms of use: licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under th conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Qualitative and quantitative analysis of safety-critical systems with S#

Johannes Leupolz[1] · **Alexander Knapp**[1] · **Axel Habermaier**[1] · **Wolfgang Reif**[1]

Abstract We give an overview of the S# (pronounced "safety sharp") framework for rigorous, model-based analysis of safety-critical systems. We introduce S#'s expressive modeling language based on the C# programming language, showing how S#'s fault modeling and flexible model composition capabilities can be used to model a case study from the transportation sector with multiple design variants. A formal semantics for executable probabilistic models is given. Fully automated qualitative and quantitative safety analyses are conducted for the case study using algorithms of the model checkers LTSmin and MRMC. The results of the quantitative analyses are discussed in comparison with results obtained by using traditional techniques.

Keywords Safety analysis · Model checking · Quantitative analysis · Executable models · Formal methods

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 [19]. Deductive Cause Consequence Analysis (DCCA) is a model-based safety analysis technique that uses model checking to compute how faults such as component failures or environmental disturbances (the causes) can cause such hazards (the consequences) [13]. From a model of a safety-critical system that not only describes the system's nominal behavior but also the relevant faults, the qualitative analysis technique DCCA determines all minimal critical fault sets, that is, combinations of faults that can cause hazards, allowing the evaluation of the system's overall safety. Traditional safety techniques describe a way how these minimal critical sets can be used to estimate an upper bound of the hazard probability using the probabilities of the component faults. But by using these fault probabilities directly when analyzing the system traces (instead of using the detour of the minimal critical sets), a better estimate of the hazard probability may be expected.

The S# modeling and analysis framework [12] can conduct DCCAs fully automatically for system models authored in the ISO-standardized C# programming language and .NET runtime environment [18,20]. Furthermore, S# can directly compute the hazard probabilities in these models using algorithms of the Markov chain model checker MRMC. This paper provides an overview of modeling and analyzing safety-critical systems with S#, using a well-known case study from the transportation sector [30]. It discusses the core concepts of S#'s modeling language and the underlying model of computation; particular emphasis is placed on S#'s flexible system design variant modeling and composition capabilities as well as its support for fault modeling. Additionally, this paper introduces S#'s unified model execution approach based on an integration of the explicit-state model checker LTSmin [21] into S#: Instead of model transformations typically employed by safety analysis tools such as VECS, Compass, and AltaRica [5,27,28], S# unifies simulations, model-based tests, visualizations, and fully exhaustive model checking by executing the models with consistent

✉ Johannes Leupolz
leupolz@isse.de

Alexander Knapp
knapp@isse.de

Axel Habermaier
habermaier@isse.de

Wolfgang Reif
reif@isse.de

[1] Institute for Software & Systems Engineering, University of Augsburg, Augsburg, Germany

semantics regardless of whether a simulation is run or some formula is model checked. By using explicit model traversal techniques of LTSmin, S# can also create low-level Markov chains which can be analyzed quantitatively by the model checker MRMC. To justify our probabilistic analysis approach, we give a formal semantics of a simplified imperative modeling language based on operational semantics and Markov chains. We present the results of the fully automated qualitative and quantitative safety analyses for the case study. To demonstrate the usefulness, we compare the quantitative results computed directly by S# with results obtained by using traditional techniques.

The main contribution of this paper is to show how systems can be modeled in a way that is amenable to automatic safety analysis and how these analyses can be performed technically. This paper extends reference [15] which treated the automatic calculation of qualitative results in the form of minimal cut sets. This paper extends the original paper by a quantitative analysis and especially the calculation of hazard probabilities from executable models. This includes a formal semantics, proof of feasibility using the case study, and a detailed discussion of the obtained results in the case study.

The remainder of this paper is structured as follows: In the next section, we give a brief introduction to the case study. Section 3 gives an overview of modeling safety-critical systems with S#. Section 4 gives the formal foundation of analyzing probabilistic executable models. How model analysis is performed technically is discussed in Sect. 5. Qualitative and quantitative analyses of the case study are presented in Sect. 6. Finally, Sect. 7 concludes with a brief discussion of related and future work.

S# and its usage documentation are available at our Web site http://safetysharp.isse.de.

2 Case study: height control system

Figure 1 shows a schematic overview of the height control system of the Elbe Tunnel in Hamburg which raises alarms and closes the tunnel when it detects overheight vehicles trying to enter the old tube, risking collisions with the tunnel's ceiling. Overall, the height control consists of five sensors: Two light barriers lb_1 and lb_2 as well as three overhead detectors od_r, od_l, and od_f; the sensors are grouped into the pre, main, and end control. The light barriers span the entire width of both lanes, whereas each overhead detector is positioned hovering above only one of the lanes. Consequently, the light barriers can only report that an overheight vehicle passes by, but cannot determine the lane they drive on; it is physically impossible to install the light barriers in a way that would allow this distinction. The overhead detectors, on the other hand, can in fact distinguish between the lanes, but they cannot differentiate between overheight vehicles and regular,

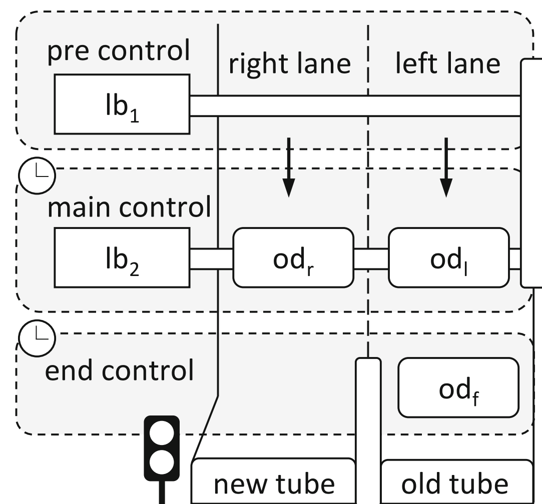

Fig. 1 A schematic overview of the case study: There are two lanes entering and exiting the two tunnel tubes at the bottom, with the arrows indicating the driving direction; overheight vehicles are only allowed to enter the new tube on the right lane. The height control consists of a pre, main, and end control that use light barriers and overhead detectors to monitor approaching vehicles

non-overheight ones; they are, however, not triggered by passenger cars. By contrast, the light barriers are positioned high enough to ensure that they are only triggered by passing overheight vehicles. The height control therefore has to combine the data of both types of sensors to determine the positions of overheight vehicles in the observed area.

Height control mechanism. When no overheight vehicles approach the tunnel, only the pre control is active, that is, the sensors of the main and end controls are deactivated. When lb_1 detects an overheight vehicle, the main control is activated, enabling its sensors and starting its timer. Additionally, a counter is increased that counts the number of overheight vehicles assumed to be between the pre and main control. The main control is deactivated when a vehicle is reported by lb_2 and od_r or od_l and the counter reaches zero, or the main control's timer times out. If the main control discovers an overheight vehicle driving on the left lane, the tunnel is closed immediately. Otherwise, the end control is activated, enabling its sensor and starting its timer. When the end control does not detect a high or overheight vehicle before its timer runs out, it is deactivated; otherwise, the tunnel is closed. Due to the road layout, vehicles cannot switch lanes after passing od_f.

Faults & hazards. Two failure modes are considered for each sensor: misdetections and false detections. Misdetections are false negatives, that is, omission faults preventing a sensor from reporting a vehicle passing by that it should detect. False detections, by contrast, are false positives, i.e., a sensor detects something that is not a vehicle, but, say, a bird. There

are two antagonistic hazards: On the one hand, the height control system is designed to prevent collisions by closing the tunnel whenever an overheight vehicle is about to enter the wrong tube (hazard *collision*). On the other hand, false alarms should be prevented, as unnecessary closures cause traffic jams and economical losses (hazard *False Alarm*). The system design is intended to strike an acceptable balance that minimizes both hazards as far as reasonable.

Design variants. Previous analyses revealed that collisions and false alarms can happen without any sensor fault occurrences [30]. Design alternatives were proposed to fix the problem, necessitating additional safety analyses to check for newly introduced safety issues. While discussing the results of our qualitative and quantitative analyses, we have a closer look at four design variants described here briefly. The variant *Original* denotes the original design. *PreImproved* is a variant where two additional sensors have been added to the pre control to improve its detection rate. Both *NoCounter* and *NoCounterT* remove the main control's counter to reduce false alarms. This requires only a change of the logic of the controller and does not require adding or removing any sensors. The logic of NoCounterT is designed to be more tolerant to the sensor input than NoCounter. Prior work [30] discusses the design variants in greater detail, with each analyzed variant requiring manual changes to a copy of the model. In this paper, by contrast, S#'s support for variant modeling and automated composition of different design variants can be leveraged to more conveniently model the different and partially orthogonal variants in a modular way, automatically composing all combinations together for fully automated safety analyses based on DCCA.

Sensor quality. Besides the concrete design of a height control, the quality of the sensors has a major influence on how likely a false alarm is raised and how likely a collision occurs or is prevented. In a perfect world, there is a perfect sensor and every system would only use this best sensor, preferably even redundantly. But in the real world, two sensors measuring the same data might have different characteristics and in some situations one sensor might be better and in others the other, e.g., one light barrier might be more prone to misdetections and the other more to false detections. The challenge is always to find the best trade-off. Being able to play with different fault probabilities when analyzing a system during the system design is definitively a plus for an analysis approach.

3 Modeling safety-critical systems with S#

Safety-critical systems typically follow the control-theoretical system partitioning into plants and controllers [26]: The controllers constantly and continuously interact with their plants to prevent potentially bad plant behavior that might result in hazards. A controller internally has an implicit or explicit model of its plant, using sensors to predict and actuators to affect the plant's state and future behavior. Discrepancies can emerge between the controller's perceived plant state and the plant's actual state: Due to faults such as component failures or environmental disturbances, sensors can provide incorrect data or actuators can have unintended effects on the plant. Subsequently, the controller is likely to mispredict the plant's future behavior, omitting control actions or unknowingly issuing destructive actions that potentially result in hazards. Models of safety-critical systems must contain both plants and controllers in order to adequately represent such control failures for formal safety analyses. In the case study, for instance, the vehicles constitute the plants with the hazards of collisions and false alarms specified over the vehicles' positions as well as tunnel closures; false alarms are control failures that the height control is unaware of, making it necessary to model the vehicles: The hazard of false alarms can only be adequately expressed over the state of the vehicles as the height control is completely unaware of its control failure; had it known that no overheight vehicles are on the left lane, it would not have closed the tunnel in the first place.

The case study model is iteratively decomposed into less complex subcomponents to increase modularity and composability, also enabling variant modeling; Fig. 2a gives an overview using SysML block definition diagrams [29]. The leaves of the hierarchy represent components for which further decomposition is not required: Either the components are modeled in sufficient detail for implementation in hardware or software, or they are standard off-the-shelf components such as light barriers that can be bought from third-party vendors and incorporated into the final system. Dependencies between a component and its parent, siblings, or subcomponents are broken up with behavioral encapsulation: Components expose provided and required ports that allow component interactions but hide actual component implementations. For the case study, the vehicle detectors are abstracted away behind a common VehicleDetector block to increase modularity and the PreControl, MainControl, and EndControl abstract blocks are introduced in order to facilitate variant modeling. The SysML internal block diagram in Fig. 2b illustrates the interdependencies between the vehicles and the preControl.

3.1 Model of computation

S# models are discrete-state, discrete-time. While the case study's controller is software-based and thus inherently discrete, the vehicles, by contrast, move continuously in reality; for the case study, standard numerical procedures for solving ordinary differential equations such as the Euler method [7] can adequately discretize vehicle behavior. Such discretiza-

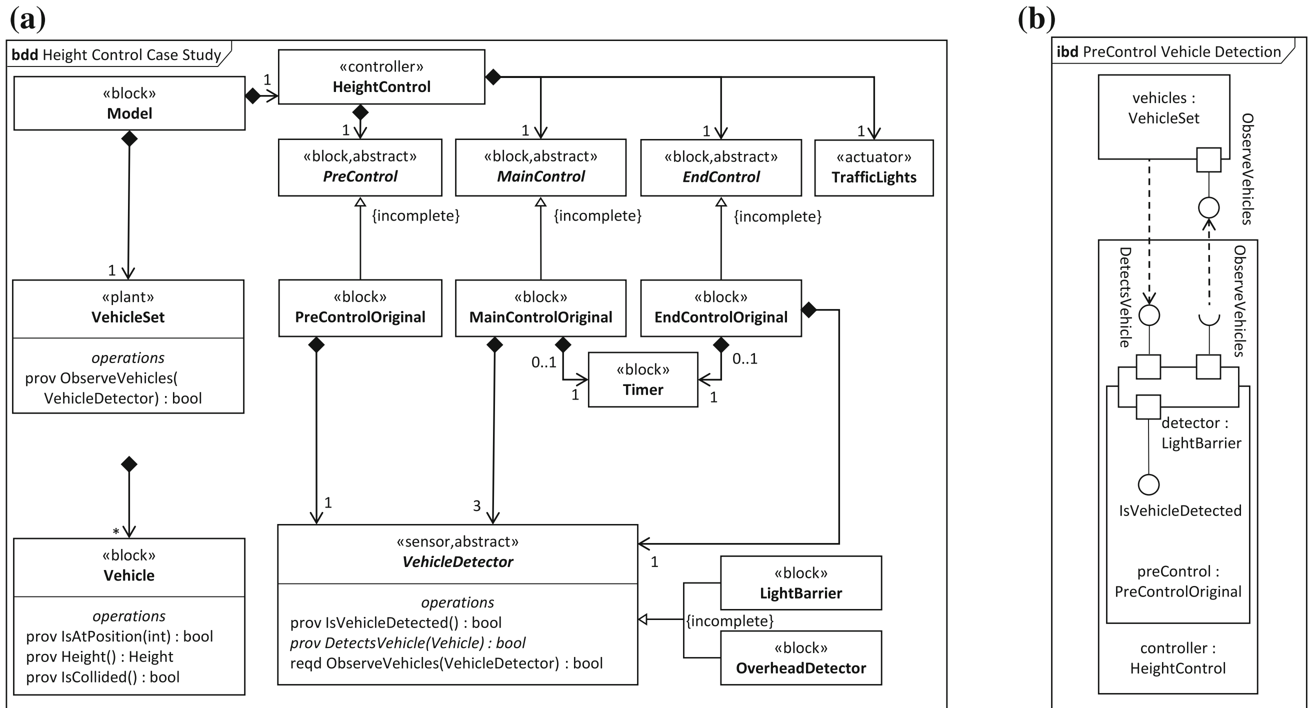

Fig. 2 A partial overview of the case study's structure and composition using SysML notation. **a** A partial block definition diagram of the case study showing some of the blocks' ports and operations. The model consists of the plant, i.e., a set of vehicles, and the actual height control system. The latter is subdivided into three subcontrollers which are abstract to support variant modeling. Only the blocks for the original controller designs are shown for reasons of brevity. **b** A partial internal block diagram showing the connection of the preControl's detector to the vehicles. While the ObserveVehicles ports are connected, the VehicleSet directly calls the DetectsVehicle port on the VehicleDetector instance passed to ObserveVehicles as shown in Listing 1

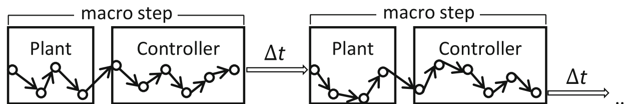

Fig. 3 Each macro step is not only subdivided into a finite sequence of micro steps, it also separates plant behavior from controller behavior, with the plant behavior always executed first. The controller's last micro step ends the macro step, causing time to pass and a new macro step to begin; no time passes between the plant and controller parts of a macro step

tions are a form of abstraction that is often possible for safety-critical systems; for the case study, in particular, the sensors can only observe the vehicles at very few locations. The model of computation embraces the zero execution time assumption for reactive systems [23]: Systems execute a sequence of macro steps at fixed points in time $t_0, t_1, t_2, \ldots$, with each macro step taking zero time to execute a finite amount of micro steps. Macro steps describe externally visible system behavior, while the intermediate micro steps are internal and thus unobservable from the outside. Between two consecutive macro steps, time Δt passes such that $t_i = t_0 + i \cdot \Delta t$ as illustrated by Fig. 3. However, S# completely abstracts from time, allowing the models to assume a Δt to pass between two consecutive macro steps that suits them best.

S#'s model of computation implicitly considers two separate components synchronously concurrent when their actions have no effect on each other within the same macro step, like the vehicles in the case study; as asynchronous concurrency can be modeled explicitly, neither modeling flexibility nor adequacy is limited. As illustrated by Fig. 3, however, macro steps linearize plant and controller execution, conceptually allowing the controllers to immediately react to changes in their plants' states: During a macro step at time t_n, the plants change their state in zero time through a sequence of micro steps. Within the same macro step, the controllers observe these changes through their sensors, compute the appropriate control actions, and update their actuators, all in zero time as well through multiple micro steps. Subsequently, the macro step ends and a new macro step begins at time t_{n+1} in which the plants are influenced by the control actions from the previous step. Sensors therefore observe the most recent plant states within a macro step, whereas actuator effects are delayed to the next step. Algorithm 1 conceptually illustrates macro step execution, sending Update signals to all components; Update signals

trigger a component's autonomous macro step behavior, if it has any.

Algorithm 1 Macro Step Execution in Two Phases

1: **function** MACROSTEP(plants : Component[*], controllers : Component[*])
2: **for** p **in** plants **do** SIGNAL(p, Update) **end for**
3: **for** c **in** controllers **do** SIGNAL(c, Update) **end for**
4: **end function**

3.2 The S# modeling language

S# provides a domain-specific modeling language embedded into the C# programming language and the .NET runtime environment [18,20]. While S# models are *represented* as C# programs, they are still models of the safety-critical systems to be analyzed; for the case study, for instance, the vehicles are part of the model even though they are not software-based in the real world. Even the software parts of S# models such as the preControl of Fig 2b are not intended to be used as the actual implementations; these are typically done in C or C++ for reasons of efficiency. Additionally, the S# models are typically abstractions of the real controller software to make model checking-based safety analysis feasible. Thus, S# is best regarded as an executable, text-based extended subset of SysML, though there currently is no automatic conversion between the two.

S# inherits C#'s language features and expressiveness and can use third-party .NET libraries and tools, in particular during model composition and initialization. However, some restrictions apply during simulations and model checking: No heap allocations are allowed, for instance. The only source of nondeterminism can be S#'s own `Choose` function; threads, in particular, are unsupported. For quantitative analyses, the concrete probabilities of the options need to be provided as parameters of the `Choose` function. When no probability is provided, S# assumes that the options are distributed uniformly, i.e., they are equally probable. S# components are represented by C# classes, instances of which correspond to S# component instances. Listing 1 declares the abstract `VehicleDetector` component from Fig. 2a as a class derived from S#'s `Component` base class. All of its methods are considered to be either required or provided ports; required ports are marked as `extern` and have no implementation. Class inheritance, interfaces, generics, lambda functions, etc. are fully supported; for example, `LightBarrier` derives from `VehicleDetector`, overriding the abstract provided port `DetectsVehicle` as necessary using C#'s shorthand syntax `=>` for simple expression-returning methods. The hierarchical system structure is established by defining fields that are of a `Component`-derived type; in Listing 1, for instance, the `VehicleSet` component has multiple subcomponents of type `Vehicle` because of the `_vs` array field.

```
enum Lane { Left, Right }
enum Height { Regular, High, Over }

abstract class VehicleDetector : Component {
  public Fault Misdetection = new
      TransientFault();
  public Fault FalseDetection = new
      TransientFault();

  public virtual bool IsVehicleDetected() =>
    ObserveVehicles(this);
  public abstract bool DetectsVehicle(Vehicle
      vehicle);
  public extern bool
      ObserveVehicles(VehicleDetector
      detector);

  [FaultEffect(Fault = nameof(Misdetection))]
  abstract class MisdetectionEffect :
      VehicleDetector {
    public override bool IsVehicleDetected()
        => false;
  }
  [FaultEffect(Fault = nameof(FalseDetection))]
  abstract class FalseDetectionEffect :
      VehicleDetector {
    public override bool IsVehicleDetected()
        => true;
  }
}
class LightBarrier : VehicleDetector {
  int _pos;
  public LightBarrier(int pos) { _pos = pos; }
  public override bool DetectsVehicle(Vehicle
      v) =>
    v.Height == Height.Over &&
      v.IsAtPosition(_pos);
}
class VehicleSet : Component { // other
    members omitted
  Vehicle[] _vs;
  public bool ObserveVehicles(VehicleDetector
      d) =>
    _vs.Any(d.DetectsVehicle);
}
class Vehicle : Component { // other members
    omitted
  int _pos; int _speed;
  Lane _lane; const int StepTime = 1;

  public extern bool IsTunnelClosed();
  public bool IsAtPosition(int pos) => _pos >
      pos &&
    _pos - _speed * StepTime <= pos;
  public bool IsCollided() => Height() ==
      Height.Over &&
    _pos >= Model.TunnelPosition && _lane ==
        Lane.Left;
  protected virtual Lane ChooseLane() =>
      Lane.Right;
  protected virtual int ChooseSpeed() =>
      MaxSpeed;

  public override void Update() {
    if (IsTunnelClosed())
      return;
    if (_pos < Model.EndControlPosition)
      _lane = ChooseLane();
    _speed = ChooseSpeed(); _pos += _speed *
        StepTime;
  }

  [FaultEffect]
  public class DriveLeft : Vehicle {
    protected override Lane ChooseLane() =>
        Choose(Lane.Right, Lane.Left);
  }
  [FaultEffect]
  public class SlowTraffic : Vehicle {
    protected override int ChooseSpeed() =>
        ChooseFromRange(MinSpeed, MaxSpeed);
  }
}
```

Listing 1 Parts of the S# model for Fig. 2a. The abstract `VehicleDetector` base type declares two provided ports `IsVehicleDetected` and `DetectsVehicle`. The former simply passes

the detector instance to the required port `ObserveVehicles` that is connected to the `ObserveVehicles` provided port of a `VehicleSet` instance (cf. Listing 2 and Fig. 2b). `Bind(nameof(d.ObserveVehicles), nameof(v.ObserveVehicles))` during model composition (not shown) where `d` is the `VehicleDetector` instance. The `VehicleSet` uses .NET's standard `Any` function to invoke the given `detector`'s `DetectsVehicle` port for each `Vehicle` instance in `_vs. LightBarriers`, for instance, detect such a `Vehicle` if it is overheight and passes the light barrier's position; the position the detector is installed at is specified via the component's constructor. The `Vehicle`'s `IsAtPosition` provided port hides the effects of positional discretization, because of which the vehicles might never reach a detector's exact position. Vehicles, by default, drive on the right lane with their maximum speed; different `Vehicle` instances execute their discretized movement behavior concurrently as they have no interdependencies. The `IsCollided` port is used to check for collision hazards.

To instantiate a S# 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. All C# language features and .NET libraries can be used to compose model instances; S#'s limitations for heap allocations, etc., only apply during simulations and model checking. The case study uses reflection to automatically instantiate all design variants of the model as shown in Listing 2; alternatively, valid model configurations could also be read from a database, for instance. A total of 16 different design alternatives result from the four main control variants and the two variants of the pre and end controls each; of these 16 variants, four are not analyzed in detail as their main controls ignore the improved detection capabilities of their pre controls, which makes them unrealistic. While the model supports an arbitrary amount of vehicles, their number has to remain constant during model checking, i.e., a model instance cannot create or remove vehicles while it is analyzed. Therefore, a fixed amount of `Vehicle` instances must be created and initialized during model composition. By default, model instances contain two overheight vehicles and one high vehicle which turned out to be sufficient to find all minimal critical fault sets for the analyzed hazards.

```
IEnumerable<Model> CreateVariants() { var
    preControls =
GetVariants<PreControl>(); var mainControls =
GetVariants<MainControl>(); var endControls =
GetVariants<EndControl>(); return from
    preControl in preControls
from mainControl in mainControls from
    endControl in endControls
where IsRealisticCombination(preControl,
    mainControl, endControl)
select new Model(preControl, mainControl,
    endControl); }

IEnumerable<Type> GetVariants<T>() => from
    type in
typeof(T).Assembly.GetTypes()
  where type.IsSubclassOf(typeof(T)) &&
     !type.IsAbstract select type;

void BindDetectors(VehicleSet s,
    VehicleDetector[] ds) {foreach(var
d in ds) Bind(nameof(s.ObserveVehicles),
    nameof(d.ObserveVehicles));
}
```

```
void VehicleFaults(VehicleSet s, Fault
    leftOHV, Fault leftHV, Fault
slowTraffic) {
    leftOHV.AddEffects<Vehicle.DriveLeft>(s.
Where(v =>
v.Height() == Height.Over));
leftHV.AddEffects<Vehicle.DriveLeft>(s.Where(v
    => v.Height() ==
Height.High));
    slowTraffic.AddEffects<Vehicle.SlowTraffic
>(s); }
```

Listing 2 Partial overview of model initialization; the full code is available online: The `CreateVariants` method instantiates all 12 realistic design variants of the case study using reflection and C#'s language integrated query functionality, filtering out unrealistic variants using the `IsRealisticCombination` method (not shown). `BindDetectors` sets up the connections between the vehicles and the detectors as illustrated by Fig. 2b. `VehicleFaults` programmatically adds the two `Vehicle` fault effects for slow-moving and left-driving vehicles to three faults using S#'s `AddEffects` method and .NET's array filter method `Where` in combination with some C# lambda functions: When either `leftOHV` or `leftHV` is activated, overheight or high vehicles are allowed to drive on the left lane, respectively. `slowTraffic` allows all vehicles to drive slower than assumed during system design.

3.3 Fault modeling

Safety analyses consider situations in which faults cause system behavior that would not occur otherwise. Fault behavior must therefore be part of the analyzed models as illustrated by the `Misdetection` and `FalseDetection` faults in Listing 1, for example. In accordance with common terminology [2], faults are activated when they somehow affect and influence actual system behavior. They are dormant until they are activated and become active, turning dormant again when they are deactivated. A fault's persistence constrains the 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. In the case study, all faults are modeled with transient persistence.

Fault activations trigger effects, represented by the nested classes `MisdetectionEffect`, `FalseDetectionEffect`, `DriveLeft`, and `SlowTraffic` in Listing 1, which cause errors or failures, i.e., internal or externally observable deviations of the components' behaviors from what they should have been, respectively. Faults therefore affect the internal state of a component or the behavior of one or more of its ports. The two fault effects of the `VehicleDetector` component, for instance, immediately result in component failures whenever their corresponding faults are activated.

Failures either provoke faults in other components or they represent system hazards; S# deduces such propagations automatically using DCCA.

False detections of the `VehicleDetector` component in Listing 1 cause the detector to incorrectly report the pres-

ence of a `Vehicle`: The field `FalseDetection` of type `Fault` is initialized with a `TransientFault` instance, activating and deactivating the fault completely nondeterministically. The type `Fault` itself has a field `Probability`. This field must be set for each instance of `Fault` with the requested value for quantitative analysis.

The fault's local effect on the component is modeled by adding the nested class `FalseDetectionEffect` that is marked with the `FaultEffect` attribute to link the effect to the fault. The effect overrides the original behavior of the `IsVehicleDetected` provided port; when the fault is activated, the port always returns `true`, regardless of the actual `Vehicle` positions. The port's original implementation is invoked only when the fault is dormant; if both the false detection and misdetection faults of a detector are activated simultaneously, S# chooses one of the fault effects nondeterministically. In the case of a quantitative analysis, each of the choices for fault effects is currently distributed uniformly. We plan to integrate a way to annotate such distributions directly in the model. As high or overheight vehicles on the left lane violate traffic laws and slow-moving vehicles violate basic design assumptions about traffic flow that influence the choice for the durations of the timers, left- and slow-driving vehicles are modeled using faults. To demonstrate S#'s flexibility in fault modeling, these faults affect multiple `Vehicle` instances: It is generally irrelevant which overheight vehicles drives on the left, and hence there is only a single fault, `leftOHV` in Listing 2, whose activation allows all overheight vehicles to switch lanes. For false alarms, it is important to differentiate between high and overheight vehicles on the left lane; however, hence there is also a `leftHV` fault; `slowTraffic`, by contrast, can affect all kinds of vehicles. Due to the use of S#'s nondeterministic `Choose` function in `DriveLeft` and `SlowTraffic`, each `Vehicle` instance decides independently whether it is actually affected by a fault activation.

4 Labeled Markov chain semantics of S#

Instead of providing a formal semantics of the complete S# modeling language, we introduce a simplified imperative modeling language which includes all necessary language features to demonstrate our approach. This modeling language abstracts away all high-level features like classes and methods and other convenience functions, which facilitate modeling. Every S# model can still be reduced to the simplified language. In the following, we focus on the probabilistic case using a structural operational semantics; nondeterminism in the simplified language for S# has been considered in [11] using a denotational approach. We first introduce formal probabilistic programs for micro steps before providing these and whole executable models with a Markov chain semantics.

4.1 Formal probabilistic programs and executable models

We replace the nondeterministic choices from formal programs of [11] by probabilistic choices. In a similar way, probabilistic choices have been added to the guarded command language by Gretz et al. [10] and to Promela by Baier et al. [3].

The valuation of variables is stored in *variable environments* $\sigma \in \Sigma \triangleq V_L \cup V_S \rightarrow Val$. We differentiate between *local variables* V_L, i.e., variables which are not persisted between macro steps, and *state variables* V_S. A variable environment maps each variable $v \in V_L \cup V_S$ to basic *values* $\nu \in Val$, e.g., integers, booleans, and doubles, which can be serialized into a finite, fixed amount of memory.

We leave the *expression language Expr* intentionally underspecified. We only assume that the evaluation of $e \in Expr$ is side-effect-free and its semantics is given by a function $\mathcal{E}[\![e]\!] : \Sigma \rightarrow Val$.

The *statements* of formal programs contain the usual statements skip, the sequential composition, the conditional statement, the while-loop, and the standard variable assignment. In addition to that, it contains the probabilistic variable assignment `choose`, which adds to each option a probability. For each probabilistic choice, the probabilities have to sum up to 1, i.e., $\sum_{q_i} q_i = 1$. The statement v `:= choose(`$e_1, \ldots, e_n$`)` is syntactic sugar for v `:= choose((`$^1/_n, e_1$`), ..., (`$^1/_n, e_n$`))`.

$$
\begin{aligned}
\rho \in Stm ::= \;& \texttt{skip} \\
& \mid \rho_1 \texttt{ ; } \rho_2 \\
& \mid \texttt{if } e \texttt{ then } \rho_1 \texttt{ else } \rho_2 \texttt{ fi} \\
& \mid \texttt{while } e \texttt{ do } \rho \texttt{ od} \\
& \mid v \texttt{ := } e \\
& \mid v \texttt{ := choose((}q_1\texttt{,}e_1\texttt{), ..., (}q_n\texttt{,}e_n\texttt{))}
\end{aligned}
$$

Figure 4 provides the inference rules for the structural operational semantics.

An example of a formal probabilistic program is shown in Fig. 6.

Based on the statements, executable models can be formalized as follows. An *executable model M* is represented by the tuple $(E, V_S, V_L, \rho_E, \rho_I)$, which contains a finite set of *state expressions* $E \subseteq Expr$, a finite set of state variables V_S, a finite set of local variables V_L such that $V_L \cap V_S = \emptyset$, a terminating *execution* probabilistic program $\rho_E \in Stm$, and a terminating *initialization* probabilistic program $\rho_I \in Stm$. The execution program is executed in each macro step. The state expressions correspond to properties to be checked, and

(skip)	$\langle \texttt{skip}, \sigma \rangle \rightarrow_M \langle \mathit{step}, \sigma \rangle$
(seq)	$\dfrac{\langle \rho_1, \sigma \rangle \rightarrow_M \mu_1}{\langle \rho_1 \texttt{ ; } \rho_2, \sigma \rangle \rightarrow_M \mu}$ with $\mu(\langle \rho_1' \texttt{ ; } \rho_2, \sigma' \rangle) = \mu_1(\langle \rho_1', \sigma' \rangle)$, $\mu(\langle \rho_2, \sigma' \rangle) = \mu_1(\langle \mathit{step}, \sigma' \rangle)$, and $\mu(s) = 0$ otherwise
(if $^{\text{true}}$)	$\langle \texttt{if } e \texttt{ then } \rho_1 \texttt{ else } \rho_2 \texttt{ fi}, \sigma \rangle \rightarrow_M \langle \rho_1, \sigma \rangle$, if $\mathcal{E}[\![e]\!]\sigma = \text{true}$
(if $^{\text{false}}$)	$\langle \texttt{if } e \texttt{ then } \rho_1 \texttt{ else } \rho_2 \texttt{ fi}, \sigma \rangle \rightarrow_M \langle \rho_2, \sigma \rangle$, if $\mathcal{E}[\![e]\!]\sigma = \text{false}$
(while $^{\text{true}}$)	$\langle \texttt{while } e \texttt{ do } \rho \texttt{ od}, \sigma \rangle \rightarrow_M \langle \rho \texttt{ ; while } e \texttt{ do } \rho \texttt{ od}, \sigma \rangle$, if $\mathcal{E}[\![e]\!]\sigma = \text{true}$
(while $^{\text{false}}$)	$\langle \texttt{while } e \texttt{ do } \rho \texttt{ od}, \sigma \rangle \rightarrow_M \langle \mathit{step}, \sigma \rangle$, if $\mathcal{E}[\![e]\!]\sigma = \text{false}$
(assign)	$\langle v \texttt{ := } e, \sigma \rangle \rightarrow_M \langle \mathit{step}, \sigma[v \mapsto \mathcal{E}[\![e]\!]\sigma] \rangle$
(por)	$\langle v \texttt{ := choose((} q_1 \texttt{,} e_1 \texttt{), ..., (} q_n \texttt{,} e_n \texttt{))}, \sigma \rangle \rightarrow_M \mu$ with $\mu(\langle \mathit{step}, \sigma[v \mapsto \mathcal{E}[\![e_i]\!]\sigma] \rangle) = q_i$, $1 \leq i \leq n$ and $\mu(s) = 0$ otherwise

Fig. 4 Structural operational semantics defining the inference rules. The usual definitions are adopted to fit into the quantitative setting. Consider for the sequential composition (seq): Assume that a distribution μ_1 for state $\langle \rho_1, \sigma \rangle$ exists. By applying (seq), we can infer the distribution μ of state $\langle \rho_1 \texttt{ ; } \rho_2, \sigma \rangle$. Indeed, when ρ is the first statement in a sequence of statements and the execution of ρ on its own in a certain variable environment is resulting in a distribution of variable environments, the subsequent statement of the sequence of statements can be executed in the resulting variable environments. How to apply the rules is illustrated by an example in Fig. 5

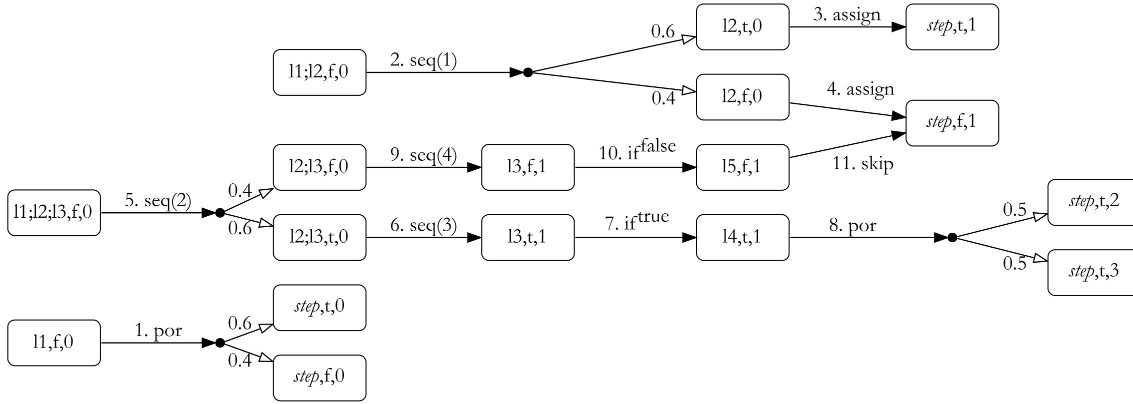

Fig. 5 Inference rules of Fig. 4 applied to example in Fig. 6. Each node represents a statement with a variable environment. To keep the node labeling concise, we write the reference numbers of the statements and abbreviate false to `f` and true to `t`, respectively. The distributions are depicted by (distribution) *arrows* with solid heads. When a distribution has only one target state with probability 1, the *solid arrow* ends directly in the node representing the state. Otherwise, a set of (probability) *arrows* with empty heads start at the head of the distribution *arrow* and end in the target state. Distribution *arrows* are numbered and contain the name of the inference rules by which they are justified and, if required, the *arrow* number of the premise's distribution

```
l1: l := choose((0.6, true), (0.4, false));
l2: y := 1;
l3: if l then
l4:     y := choose(2, 3)
    else
l5:     skip
    fi
```

Fig. 6 An example for a formal probabilistic program which contains two probabilistic choices. Each statement is numbered to allow easy referencing, e.g., skip has the reference `l5`. These references are not part of the program

the state variables represent the state vector to be stored during model checking.

4.2 Markov chains

We briefly recall the terminology of Markov chains and their probability space [4]. Since we will make use of two different flavors, we introduce the basic notions for a slightly generalized form of Markov chains.

A *generic Markov chain* $\mathcal{M} = (S, \Theta, R, \mu_0)$ consists of a finite set S of *states*, a finite set Θ of *targets* with a *target state* function $\tau : \Theta \rightarrow S$, a *transition distribution* function $R : S \rightarrow \mathit{Dists}(\Theta) = \{\mu : \Theta \rightarrow [0, 1] \mid \sum_{\theta \in \Theta} \mu(\theta) = 1\}$, and the initial probability distribution $\mu_0 \in \mathit{Dists}(\Theta)$. We also write $S^{\mathcal{M}}$ for $\mathcal{M}$'s states, $\Theta^{\mathcal{M}}$ for $\mathcal{M}$'s targets, etc. A standard Markov chain is represented in our terms by choosing $\Theta^{\mathcal{M}} = S^{\mathcal{M}}$ and $\tau^{\mathcal{M}}$ as the identity. When $\Theta^{\mathcal{M}}$ is chosen to be $L \times S^{\mathcal{M}}$ for some set of *labels* L with $\tau^{\mathcal{M}}(\ell, s) = s$, then we obtain an *$L$-labeled Markov chain*.

A *finite path* $\pi = \theta_0 \theta_1 \ldots \theta_n \in (\Theta^{\mathcal{M}})^*$ of $\mathcal{M}$ is a sequence of targets where $\mu_0^{\mathcal{M}}(\theta_0) > 0$ and $R(\tau^{\mathcal{M}}(\theta_i))(\theta_{i+1}) > 0$ for all $0 \leq i < n$; we write $\mathit{Paths}_{\mathit{fin}}$ for the set of all finite paths of $\mathcal{M}$. A *path* $\widehat{\pi} = \theta_0 \theta_1 \theta_2 \ldots \in (\Theta^{\mathcal{M}})^{\omega}$ of $\mathcal{M}$ is an infinite sequence of targets where $\mu_0^{\mathcal{M}}(\theta_0) > 0$ and $R(\tau^{\mathcal{M}}(\theta_i))(\theta_{i+1}) > 0$ for all $0 \leq i$, and the set of all (infinite) paths of $\mathcal{M}$ is denoted by *Paths*.

The *cylinder set* $Cyl^{\mathcal{M}}(\pi)$ of a finite path $\pi \in Paths_{fin}$ contains all paths of $\mathcal{M}$ that start with π, i.e., $Cyl^{\mathcal{M}}(\pi) = \{\widehat{\pi}' \in Paths \mid \pi \in Pref(\widehat{\pi}')\}$ where $Pref(\widehat{\pi})$ denotes all finite paths that are prefixes of $\widehat{\pi}$. The *probability space* $(\Omega^{\mathcal{M}}, \mathfrak{E}^{\mathcal{M}}, Pr^{\mathcal{M}})$ of a generic Markov chain $\mathcal{M}$ is given by the sample space $\Omega^{\mathcal{M}} = Paths$, the events $\mathfrak{E}^{\mathcal{M}}$ as the smallest σ-algebra on $\Omega^{\mathcal{M}}$ that contains $\{Cyl^{\mathcal{M}}(\pi) \mid \pi \in Paths_{fin}\}$, and the probability measure $Pr^{\mathcal{M}}$ with

$$Pr^{\mathcal{M}}(Cyl^{\mathcal{M}}(\theta_0 \ldots \theta_n)) \\ = \mu_0^{\mathcal{M}}(\theta_0) \cdot \prod_{0 \le i < n} R(\tau^{\mathcal{M}}(\theta_i))(\theta_{i+1}).$$

For a countable $\Pi \subseteq Paths_{fin}^{\mathcal{M}}$ such that $Cyl^{\mathcal{M}}(\pi) \cap Cyl^{\mathcal{M}}(\pi') = \emptyset$ for all $\pi \neq \pi' \in \Pi$ (ensuring that each path is only counted once), we then obtain

$$Pr^{\mathcal{M}}(\Pi) = \sum_{\pi \in \Pi} Pr^{\mathcal{M}}(Cyl^{\mathcal{M}}(\pi)).$$

In particular, we can consider reachability of a target $\theta \in \Theta^{\mathcal{M}}$ by computing the probability of the set of paths

$$Paths_{\mathbf{F}\theta}^{\mathcal{M}} = \{\theta_0 \ldots \theta_n \in Paths_{fin}^{\mathcal{M}} \mid \\ \theta_n \models \theta \wedge \forall 0 \le i < n \,.\, \theta_i \not\models \theta\}$$

where $\theta' \models \theta$ if, and only if, $\theta' = \theta$. We abbreviate $Pr^{\mathcal{M}}(Paths_{\mathbf{F}\theta})$ by $Pr^{\mathcal{M}}(\mathbf{F}\theta)$. The probability of reaching a particular state $s \in S^{\mathcal{M}}$ is defined analogously by considering

$$Paths_{\mathbf{F}s}^{\mathcal{M}} = \{\theta_0 \ldots \theta_n \in Paths_{fin}^{\mathcal{M}} \mid \\ \theta_n \models s \wedge \forall 0 \le i < n \,.\, \theta_i \not\models s\}$$

where now $\theta \models s$ if, and only if, $\tau^{\mathcal{M}}(\theta) = s$.

4.3 Labeled Markov chains of executable models

Given an executable probabilistic program ρ with initial variable environment σ_0, we now define a standard Markov chain representing the operational semantics with its probability distributions. The states of the Markov chain will be the possible states the program might be in. In fact, we cannot simply choose $(Stm \cup \{step\}) \times \Sigma$ as the states because Stm is infinite; we assume Σ, however, to be finite by only considering finite variables domains. Thus, we focus on those statements Stm_ρ relevant for ρ as the set of all sub-programs of ρ and for each loop `while` e `do` ρ' `od` in ρ the unwound loop, i.e., ρ' `;` `while` e `do` ρ' `od`.

Consequently, we define the *Markov chain semantics* of a single formal probabilistic program ρ with initial variable environment σ_0 as the standard Markov chain $\mathcal{M}$ with states $S^{\mathcal{M}} = (Stm_\rho \cup \{step\}) \times \Sigma$; transition distributions $R(\langle \rho, \sigma \rangle) = \mu \iff \langle \rho, \sigma \rangle \rightarrow_M \mu$ and $R(\langle step, \sigma \rangle)(\langle step, \sigma \rangle) = 1$ and 0 otherwise; and initial distribution $\mu_0^{\mathcal{M}}(\langle \rho, \sigma_0 \rangle) = 1$ and 0 otherwise. We write $\mathcal{M}[\![\rho]\!]\sigma_0$ for this macro step Markov chain $\mathcal{M}$.

Based on the Markov chains induced by formal probabilistic programs representing single macro steps, we now move to a complete executable model $M = (E, V_S, V_L, \rho_E, \rho_I)$. For notational succinctness, let us write $\Sigma(\Lambda, \sigma_S)$ for the variable environments where exactly the set of expressions $e \in \Lambda \subseteq E$ are true and the state variables coincide with σ_S, i.e., $\Sigma(\Lambda, \sigma_S) = \{\sigma_L \cup \sigma_S \mid \forall e \in E \,.\, e \in \Lambda \leftrightarrow \mathcal{E}[\![e]\!](\sigma_L \cup \sigma_S) = \text{true}\}$; and σ_S^L for adding the initialization of local variables to a state variable environment, i.e., $\sigma_S^L = \iota_L \cup \sigma_S$. With these preliminaries, the *Markov chain semantics* of M is defined as the 2^E-labeled Markov chain $\mathcal{M}$ with states $S^{\mathcal{M}} = V_S \rightarrow Val$, transition distributions

$$R(\sigma_S)(\Lambda', \sigma_S') = \textstyle\sum_{\sigma' \in \Sigma(\Lambda', \sigma_S')} Pr^{\mathcal{M}[\![\rho_E]\!]\sigma_S^L}(\mathbf{F}\langle step, \sigma' \rangle)$$

and initial distribution

$$\mu_0^{\mathcal{M}}(\Lambda', \sigma_S') = \textstyle\sum_{\sigma' \in \Sigma(\Lambda', \sigma_S')} Pr^{\mathcal{M}[\![\rho_I]\!]\iota_S^L}(\mathbf{F}\langle step, \sigma' \rangle).$$

In fact, $R^{\mathcal{M}}$ is only well defined if ρ_E always terminates when started in σ_S^L, a requirement which we have to assume for computing successors (see Sect. 5.1). Note that the local variable environment produced in a macro step is not recorded in the states of the labeled Markov chain, but contributes to the evaluation of the expressions.

5 Analyzing safety-critical systems with S#

S# unifies LTSmin-based, fully exhaustive, explicit-state model checking and MRMC-based probabilistic model checking as well as non-exhaustive simulations as shown in Fig. 7: In all cases, the S# runtime executes a model compiled with the S# compiler, ensuring the correct execution semantics of faults and required ports. During model checking, all combinations of nondeterministic or probabilistic choices and fault activations within a model are exhaustively enumerated. S# is not a software model checker such as Java Pathfinder or Zing [1,34], however, as it does not analyze states after every instruction; only state changes between macro steps are considered.

5.1 Execution semantics of S# models

The Model class shown in Fig. 8 captures S#'s model execution semantics. It consists of a hierarchy of Component instances, each having fields that form the component's state. Fields are allowed to be of most .NET types, including arrays, delegates, object references, and classes comprised of any of these like `List<T>`; e.g., the state of a `Vehicle` instance from Listing 1 consists of `_pos`, `_speed`, and `_lane`. For

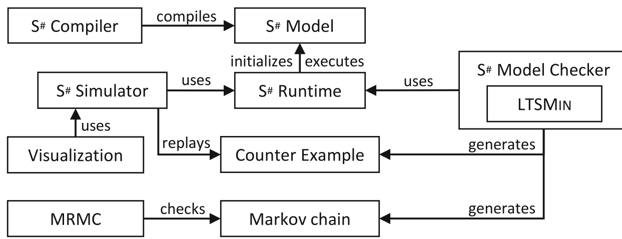

Fig. 7 Illustration of S#'s execution-centric architecture: The runtime initializes S# models compiled by a slightly extended version of the C# compiler to ensure the desired S# semantics of required ports and faults. Both the simulator and the model checker use the runtime to execute a model. The only difference between simulation and model checking is that the latter is exhaustive, checking all combinations of nondeterministic choices within a model, whereas the former considers a single combination only. Counter examples generated by the model checker can be replayed by the simulator for debugging purposes. Model visualizations build upon the simulator. For quantitative analysis, the S# model checker generates a Markov chain using state space exploration algorithms based on LTSmin. The generated Markov chain is checked using MRMC

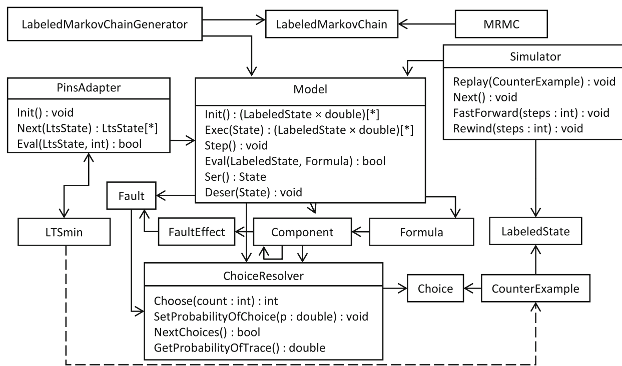

Fig. 8 A UML class diagram showing the classes required to simulate and model check S# models. The Model class is responsible for Formula evaluation and model execution with the intended S# semantics; in particular, it uses a ChoiceResolver instance to determine all combinations of nondeterministic choices and fault activations. For quantitative analyses, Model passes ChoiceResolver the probability of each of its taken choices directly after making the choice. The Simulator class, the LabeledMarkovChainGenerator, and LTSmin are decoupled from Model execution semantics, only requiring them to take care of State management

efficient storage and comparison, Component states are serialized and deserialized to and from fixed-sized byte arrays, represented by the State class. The set of State instances has to be finite; therefore, object creation and other forms of heap allocations during model checking and simulation are unsupported; during model initialization, on the other hand, no such restrictions exist. The method Model::Ser stores Component states in a State instance, whereas Model::Deser does exactly the opposite. S# generates these two methods dynamically at runtime via reflection, tailoring them to a specific Model instance to guarantee maximum efficiency with respect to serialization time and state storage size.

The method Model::Init generates all initial labeled states of a Model instance, while Model::Exec computes all successors of a state. In the case of quantitative analysis, these labeled states are accompanied by probabilities. Details are shown by Algorithm 2: For all combinations of nondeterministic choices and fault activations determined by ChoiceResolver::NextChoices, the given state is deserialized using Model::Deser so that Model::Step can allow all Component instances to compute their successor states, which are subsequently serialized using Model::Ser. At this point of time, the local variables are still set in the memory and state formulas which might reference some of these are evaluated. In case of a quantitative analysis, the probability of the micro step trace is retrieved from the ChoiceResolver. It may occur that two combinations of nondeterministic choices lead to the same labeled state. We assume that the += operator merges such probabilities on-the-fly. Finally, the computed successors are returned. Model::Step is conceptually equivalent to Algorithm 1. The Formula class represents state formulas that evaluate arbitrary Boolean C# expressions over Component instances. The evaluation is expected to be terminating, deterministic, and side-effect-free; otherwise, the exact behavior is unspecified.

Model::Eval evaluates a Formula instance for a given serialized LabeledState, which is needed for compatibility with LTSmin.

Algorithm 2 Model::Exec (s : State)

```
1:  var successors : double × LabeledState[*] = []
2:  while choiceResolver.NextChoices() do
3:      Deser(s);
4:      Step();
5:      var state = Ser();
6:      var label = EvaluateStateFormulas();
7:      var labeledState = (label,state);
8:      var prob = choiceResolver.GetProbabilityOfTrace();
9:      successors += (labeledState,prob);
10: end while
11: return successors
```

The only allowed source of nondeterminism within a Model are Fault instances and invocations of ChoiceResolver::Choose; the latter records the number of choices that can be made at a specific point during the execution of Model::Step and returns the index of the chosen value. Based on this index, one of the options is selected. For quantitative analysis, the probability of the selected option is passed to the ChoiceResolver. ChoiceResolver::Choose is not accessible for the modelers directly. Instead, Component provides a Choose method which is similar to the one presented in the simplified language in Sect.4 both for convenience and to prevent misuse.

Other sources of nondeterminism, such as race conditions of threads, are not captured by S#; S# does not analyze the

code it executes for illegal nondeterminism. There are two reasons why a state might not have any successors at all: Either Model::Step does not terminate or its execution is aborted abnormally due to an unhandled exception, for example. Both cases indicate bugs in the model or in the S# runtime that are readily discoverable.

5.2 Qualitative model checking of S# models

LTSmin allows S# to execute a Model using Model::Exec during model checking. In order to enable the integration of various modeling languages into LTSmin, the so-called PINS interface written in C [21] is provided that S# makes use of S#'s integration of LTSmin takes about 250 lines of C++/CLI code, a Microsoft-specific variant of C++ that integrates into the .NET framework, allowing for easy interoperability between C/C++ and C#. The PinsAdapter class maps LTSmin's C-based PINS interface to the C# interface of the Model class: PinsAdapter::Init initializes and sets up LtsMin, which in turn repeatedly calls PinsAdapter::Next to compute all successors of a serialized state using Model::Exec. PinsAdapter::Eval prompts S# to evaluate a Formula instance identified by its index for some serialized state by calling Model::Eval. It is also the job of the PinsAdapter to map between LabeledState and LtsState. An LtsState contains both the labeling and the state. For a S# model, a Kripke structure is generated on-the-fly using this interface. If there are no bugs that cause the S# model to get stuck in an infinite loop or to throw an unhandled exception, exploration of the Kripke structure terminates as soon as all reachable states are encountered. S# models always generate Kripke structures without any deadlock states; consequently, all paths through the Kripke structure are of infinite length.

It is the responsibility of LTSmin to do the actual model checking, that is, to check whether an LTL formula or an invariant is satisfied by an induced Kripke structure. In the case study, for example, the LTL formula checking whether there either is no tunnel closure or no collisions occur before the tunnel is closed is specified as

```
G(!model.TunnelClosed()) ||
U(model.Vehicles.All(v =>
    !v.IsCollided()),
    model.TunnelClosed())
```

in S#; the formula obviously does not hold as faults are indeed able to cause situations in which collisions occur before any tunnel closures. The two operands of the LTL until operator used above are two C# expressions that are represented by two Formula instances and evaluated during model checking; similarly, the operand of the globally operator is also such a Formula instance. Both LTL operators and their disjunction are also converted into Formula instances that are subsequently transformed for interpretation by LTSmin which in turn invokes the contained C# expressions at the appropriate times. If LTSmin detects a violation of the Formula instance that is checked, it generates a CounterExample that consists of a sequence of State instances, which are trivial to deserialize back into sequences of Component states using Model::Deser. For later replay, a CounterExample also captures the nondeterministic choices that the ChoiceResolver made during the generation of the CounterExample, which also include fault activations.

5.3 Quantitative model checking of S# models

We focused on two classes of probabilities for the quantitative analysis of safety-critical systems: the probability that a certain state formula is satisfied at some time in the future (*finally*) and the probability that a certain state formula is satisfied within a fixed number of system steps (*bounded finally*). To calculate these probabilities, it is not sufficient to stop the first time after a labeled state has been found which satisfies a formula. There might exist more such labeled states which contribute to the total probability as shown in Sect.4. Thus, to calculate the correct probability, the corresponding Markov chain of a model must be created.

We reimplemented LTSmin's state exploration algorithm [25]. This algorithm exploits the multi-core capabilities of modern processors and calculates the successors of each state exactly once. Due to the latter property, we could easily amend the algorithm to record the distributions of each state into a labeled Markov chain during model traversal.

For a S# model $\mathsf{m} : \mathsf{Model}$, a $2^{\mathsf{m.StateFormulas}}$-labeled Markov chain $\mathcal{M}$ is generated such that $\mathcal{M} = (S, R^{\mathcal{M}}, \mathsf{m.Init}())$ with $R^{\mathcal{M}}(s) = \mathsf{m.Exec}(s)$. Not having to represent a program counter or local variables reduces the state space enormously. This makes probabilistic model checking feasible for large models using an explicit approach.

The created labeled Markov chain can be exported into the `.tra` format of MRMC. Subsequently, MRMC may be used to check certain formulas denoted in the temporal logic PCTL. We also reimplemented algorithms for finally and bounded finally. Therefore, it is often not necessary to save the models first (those files easily grow to a size of several gigabytes) and for MRMC to load this file which costs time. These algorithms are described in [31].

5.4 Simulating S# models

Simulations of S# models work similar to model checking except that only a single path of the induced Kripke structure or Markov chain is explored. Simulator instances are either guided or unguided: Unguided simulations do not follow a predetermined path, whereas guided ones are used to replay the CounterExample instance passed to Simulator::Replay by forcing the nondeterministic choices

made by the model checker upon the simulator. Consequently, counter examples cannot only be stepped through state by state, but also allow debugging each transition, giving insights into why and how some possibly undesired successor labeled state is reached from some source state. A simulation stores all computed labeled states, allowing it to be fast forwarded or rewound by some number of steps using Simulator::FastForward and Simulator::Rewind. In contrast to Algorithm 2, Simulator::Next computes only one successor of the current state using the sequence of method calls Model::Deser, Model::Step, and Model::Ser based on a set of predetermined choices. Both simulation-based model tests as shown in Listing 3 as well as visualizations can be implemented on top of the Simulator class. In the interactive visualization of the case study, for instance, the user can spawn high and overheight vehicles and change their speed and lanes using the mouse or touch; visual replays of counter examples help to understand the situations in which hazards occur.

For the quantitative analysis, the options of each choice are selected by using the random number generator of .NET during an unguided simulation. Modelers using this feature for safety analysis must be aware that faults are in most cases very unlikely. Thus, bad behaving traces are chosen with an extreme low probability. The unguided simulation could be used as foundation for the approximation of the hazard probability using Monte Carlo experiments which requires an enormous number of simulation runs due to the low fault probabilities. Thus, using model checking to calculate the exact probability is often the easier way.

```
model = new Model(new PreControlOriginal(), new
MainControlOriginal(), new
    EndControlOriginal());
SuppressAllFaultActivations(model); new
Simulator(model).FastForward(steps: 20);
    foreach (var vehicle in
model.Vehicles)
    Assert.IsFalse(vehicle.IsCollided());
```

Listing 3 A model test based on a S# simulation of the case study's original design with all faults suppressed by the helper method `SuppressAllFaultActivations` (not shown), i.e., dormant the entire time. The test asserts that after the first 20 simulated steps, no vehicles collide with the tunnel, as all vehicles drive on the right lane without any fault activations.

6 Evaluation

For evaluating the S# approach to the analysis of safety-critical systems, we discuss both qualitative and quantitative results for our height control case study and compare these results to previous, traditional approaches.

6.1 Qualitative analysis of the height control case study

S# automatically conducts DCCAs to compute all minimal critical fault sets for a hazard H given as a Formula instance, i.e., an arbitrary C# expression that is interpreted as a propositional logic formula over the induced Kripke structure K: For faults F contained in K, S# individually checks all combinations of faults $\Gamma \subseteq F$, determining whether Γ does or does not have the potential to cause an occurrence of H [13]. Γ is a critical fault set for H if and only if there is the possibility that H occurs and before that, at most the faults in Γ have occurred. More formally, using LTL: $\Gamma \subseteq F$ is safe for H if and only if $K \models \neg(only_F(\Gamma)\mathbf{U}H)$, where $only_F(\Gamma) :\Leftrightarrow \bigwedge_{f \in F \setminus \Gamma} \neg f$. A fault set is critical if and only if it is not safe. A critical fault set Γ is minimal if no proper subset $\Gamma' \subsetneq \Gamma$ is critical; a complete DCCA computes all such minimal critical fault sets. For any critical fault set Γ, any superset $\Gamma' \supseteq \Gamma$ is also critical because, in general, additional fault activations cannot be expected to improve safety. The criticality property's monotonicity with respect to set inclusion [13] trivially holds regardless of the actual model as the LTL formula above does not require any critical faults $f \in \Gamma$ to be activated; instead, it only suppresses the activations of all other faults $f \in F \setminus \Gamma$. In practice, monotonicity 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. As seen in Listing 4, S# automatically takes advantage of monotonicity, significantly reducing the amount of fault sets to be checked for criticality; in particular for the hazard of false alarms, only 3% of all possible sets have to be analyzed for criticality. In the worst case, however, DCCA does indeed have exponential complexity.

```
Results: Collisions (1 second) Minimal
    Critical Sets (35 fault sets
had to be checked; <1
   (1) { leftOHV, slowTraffic }
   (2) { leftOHV, misdetectionLB2 }
   (3) { leftOHV, misdetectionLB1 }
   (4) { leftOHV, misdetectionODF }
   (5) { leftOHV, falseDetectionLB2 }

DCCA Results: false alarms (3.9\,s)
Minimal Critical Sets (43 fault sets had to be
    checked; 1
   (1) { leftHV }
   (2) { falseDetectionODF }
   (3) { falseDetectionODL }
   (4) { falseDetectionLB2 }
   (5) { misdetectionODR }
```

Listing 4 Overview of the DCCA results for both hazards using the case study's original design. S# automatically analyzes a total of 13 faults; exploiting the monotonicity of the criticality property is especially effective for the hazard of false alarms, significantly reducing analysis times.

Listing 4 shows the DCCA results for the original case study design; for collisions, specified in S# as `m.Vehicles. Any(v => v.IsCollided())` for a Model instance `m`, at least one overheight vehicle must drive on the left lane, hence the `leftOHV` fault of Listing 2 is contained in all minimal critical fault sets. For instance, the fault set { `leftOHV`, `slowTraffic` } is critical for collisions because two overheight vehicles can

pass the pre control's light barrier simultaneously, while one is faster than the other: The faster vehicle deactivates the main control, and the slower vehicle can pass the end control when it is already deactivated again. `{ leftHV }` is minimal critical for false alarms, specified as `m.HeightControl.IsTunnelClosed() && m.Vehicles.All(v => v.DrivesRight())`, because of a high vehicle passing the main control's left overhead detector when an overheight vehicle passes the main control's light barrier at the same time. Safety analysis times for some design variants are significantly higher than the ones in Listing 4 due to the larger number of faults that have to be checked.

The results of our qualitative analyses for the four design variants Original, PreImproved, NoCounterT, and NoCounter are summarized in the first four rows of Table 2. All DCCAs have been conducted automatically in between 1 and 13 s. In most cases, a system variant with a fewer number of minimal critical sets should be preferred over one with more when the cardinality of each minimal critical set is equal. For the prevention of the hazard Collision, both NoCounter and NoCounterT have the same four minimal critical sets each consisting of two elements. The minimal critical sets of both Original and PreImproved are proper supersets of the former minimal critical sets. Thus, NoCounterT and NoCounter should be preferred according to the DCCA result of hazard Collision. Applying the same reasoning on the hazard False Alarm leads to the result that NoCounterT should be chosen. To sum it up, NoCounterT seems to be the best variant to prevent both hazards according to the qualitative analysis.

6.2 Quantitative analysis of the height control system with S#

Including the probability of each fault into the model is essential for the quantitative analyses. Table 1 shows different sets of fault probabilities used in our analyses. One dimension of our analyses is the impact of the quality of different sensors and the probability of drivers complying with the traffic rules. The other dimension is the selected design variant of the height control.

For our quantitative analyses, we analyzed the probabilities of each hazard to occur within 50 time steps for the four design variants. Using a step limitation for the safety analysis is essential for the calculation of a hazard because in most systems the probability that a hazard will eventually (infinite time horizon) occur is almost always close to 1. The results are summarized starting with row 5 in Table 2. The analyses have been conducted automatically in between 3.5 and 17.0 minutes. The number of states and transitions had a major impact on the model checking time. In either case, the qualitative analysis is faster than the quantitative analysis by orders of magnitude. Still, these analyses show a more differentiated view than the qualitative analyses. The analyses confirm the common assumption that the quality of the senors has a major impact on the safety of the system. Besides from that, we had several more interesting findings for the height control by comparing the probabilities: (1) The better the quality of the sensors, the closer the hazard probabilities in different variants. (2) For low-quality sensors, the selection of a better variant can almost halve the probability of a false alarm. (3) Better drivers only have a minor impact on the probability of a false alarm. (4) Even when the minimal cut sets are equal, NoCounterT and NoCounter have different hazard probabilities. NoCounter should be preferred over NoCounterT when the probability of a collision should be reduced. But this has the price of a major increase of false alarms.

The quality of the sensors plays the most crucial role for the safety of the system. Thus, we further analyzed the impact of the quality of the light barriers by varying the probability of a false detection in a range between 1.0×10^{-6} and 1.0×10^{-2}. The results are shown in the graphs of Fig. 9.

Table 1 We analyzed four different sets of fault probabilities

Fault	StandardQuality	BetterLightBarrier	BetterSensors	BetterDrivers
False detection of light barrier	5×10^{-3}	5×10^{-6}	5×10^{-6}	5×10^{-3}
Misdetection of light barrier	1×10^{-4}	1×10^{-4}	1×10^{-6}	1×10^{-4}
False detection of overhead detector	5×10^{-3}	5×10^{-3}	5×10^{-6}	5×10^{-3}
Misdetection of overhead detector	1×10^{-4}	1×10^{-4}	1×10^{-6}	1×10^{-4}
High vehicle changes to left lane	1×10^{-2}	1×10^{-2}	1×10^{-2}	1×10^{-4}
Overheight vehicle changes to left lane	1×10^{-3}	1×10^{-3}	1×10^{-3}	1×10^{-5}
Slow traffic	1×10^{-1}	1×10^{-1}	1×10^{-1}	1×10^{-3}

The first set *StandardQuality* contains the probabilities the system design starts with. *BetterLightBarrier* deviates from this basis by decreasing the false detection probability of the light barriers by three orders of magnitude

The third set *Better sensors* decreases the fault probabilities of *all* sensors notably

In the last test set *BetterDrivers*, only the fault probabilities of *external* factors are decreased, i.e., faults the designers of the height control cannot influence

Table 2 This table summarizes the qualitative and quantitative analysis results of our the four variants of the height control system, namely Original, PreImproved, NoCounterT, and NoCounter, based on the two hazards collision and false alarm (see Sect. 2)

Hazard	Hazard collision				Hazard false alarm			
Variant	Original	PreImproved	NoCounterT	NoCounter	Original	PreImproved	NoCounterT	NoCounter
Faults	13	17	13	13	13	17	13	13
Time in sec – DCCA	1	4	1	2	4	8	13	3
MCS #	5	6	4	4	5	5	4	5
MCS ∅	2	2.3	2	2	1	1	1.5	1
Time in sec – Pr	554	1021	207	252	508	930	191	236
States	2,002,728	2,002,728	847,308	847,308	1,899,456	1,899,456	803,616	803,616
Transitions	616,840,849	1,273,222,777	247,131,751	355,869,721	635,826,673	1,312,411,465	254,738,251	366,823,081
Pr(StandardQuality)	2.00×10^{-7}	1.99×10^{-7}	1.56×10^{-7}	1.50×10^{-7}	5.45×10^{-2}	5.46×10^{-2}	3.23×10^{-2}	6.11×10^{-2}
Pr(BetterLightBarrier)	1.95×10^{-7}	1.94×10^{-7}	1.58×10^{-7}	1.54×10^{-7}	4.20×10^{-2}	4.20×10^{-2}	3.24×10^{-2}	4.23×10^{-2}
Pr(BetterSensors)	4.32×10^{-8}	4.31×10^{-8}	1.17×10^{-8}	1.16×10^{-8}	3.79×10^{-3}	3.79×10^{-3}	3.60×10^{-3}	3.85×10^{-3}
Pr(BetterDrivers)	1.57×10^{-9}	1.57×10^{-9}	1.47×10^{-9}	1.45×10^{-9}	5.01×10^{-2}	5.02×10^{-2}	2.93×10^{-2}	5.75×10^{-2}

The number of faults for the variant PreImproved increased compared to Original because of its two additional sensors
The variants NoCounterT and NoCounter have the same number of faults as Original because only the controller software was changed
The qualitative analyses using the DCCA were executed in between 1 and 13 s (row "Time in sec – DCCA")
The results of the DCCAs are condensed into the two metrics "number of minimal critical sets" (MCS #) and "average cardinality of the minimal critical sets" (MCS ∅)
For the quantitative analyses, we used the four different fault probability sets from Table 1 for the quantitative analysis
Within a variant and hazard, the different parameter sets do not change the number of states, transitions, or the quantitative analysis times (row "Time in sec – Pr"), but only the resulting probability. The resulting probabilities are shown in the row Pr(—) with the name of the parameter being the name of the probability set respectively

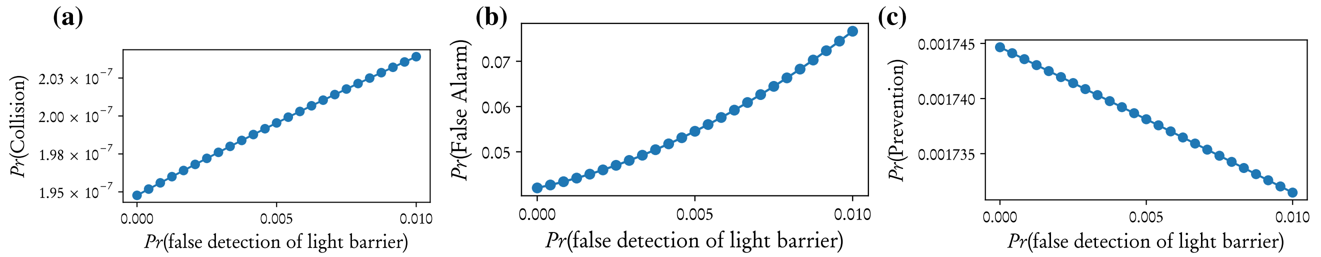

Fig. 9 These graphs illustrate the impact of the quality of the light barrier. The probability of the false detection of the light barriers is varied in a range between 1.0×10^{-6} and 1.0×10^{-2}, while all other fault probabilities keep their value (see StandardQuality in Table 1). Twenty-five linear sample points were used. We analyzed the effect on the hazards Collision and False Alarm and we also computed the probability that the height control system prevented a collision (Prevention). **a** The quality of the light barriers has a minor impact on the probability of a collision. When the false detection gets more probable, the collision gets more probable as well. Nevertheless, the leftmost and rightmost sample values only differ by 9.1×10^{-9}. It seems that the graph is almost linear with its inflection point being somewhere in the middle of the graph. **b** The quality of the light barriers has a clear impact on the probability of a false alarm. When the false detection gets more probable, the false alarm gets more probable as well. This graph shows a clear curve with its inflection point being somewhere in the middle of the graph. Comparing the leftmost and rightmost sample values, the probability of the hazard almost doubled. But increasing the quality of the sensors even more only has a minor effect. **c** The probability of a collision and of the prevention of a collision are obviously negatively correlated when the behavior of the drivers is the same. This is also shown clearly in this graph. The worse the light barriers, the worse the chance to prevent a collision

6.3 Traditional quantitative analysis of the height control system

In traditional fault tree-based methodologies, the hazard probability is deduced from the resulting minimal cut sets of the qualitative analysis [33]. Minimal cut sets are approximately the same as minimal critical sets of the DCCA. Faults are usually assumed to be stochastically independent. *Rare event approximation* can be applied when the probabilities of each fault are very small. The rare event approximation states that the probability of a hazard H can be approximated by summing up the probabilities of each minimal cut set leading to H. This is an upper bound for the probability of the hazard. The probability of one minimal cut set is approximated by multiplying the probabilities of the faults inside that minimal cut set. More formally,

$$Pr_{\text{rea}}(H) = \sum_{\Gamma \in MCS(H)} \prod_{f \in \Gamma} Pr(f),$$

where $Pr(f)$ denotes the probability of fault f. This delivers an explanation why a lower number of minimal critical sets should be preferred when the cardinality of each minimal critical set is the same.

The probabilities given in Table 1 are the probabilities for each fault to occur in 1 time step. For the traditional analysis, we need to know the probability to fail at least once in 50 time steps. These probabilities can be derived easily from the given probabilities. According to the geometric distribution in statistics, given a probability p of success in one trial and k independent trials, the probability that we have at least one success within k trials is $1-(1-p)^k$ [9]. In our case, we want to determine how probable it is to fail within 50 time steps. Ironically, we treat the occurrence of a fault as "success."

Note that component suppliers often publish the mean time to failure (MTTF) of their components. The definition of MTTF is based on the exponential distribution. Due to the discrete nature of S#, the geometric distribution is used, which is the discrete analogue of the exponential distribution. The probability $Pr_1(f) = 5 \times 10^{-3}$ of a fault f denotes the chance that the corresponding fault occurs within one time step. Thus, the accumulated probability to fail within 50 time steps at least once is $Pr_{50}(f) = 1 - (1 - Pr_1(f))^{50}$ ≈ 0.22.

We assume that we can ignore the fact that some faults are only relevant in some system states. We also assume that a fault of a minimal critical set is present when it occurs in any of these 50 time steps. This is clearly an over-approximation. Applying the formula for $Pr_{\text{rea}}(H)$ on the minimal cut sets calculated earlier gives us these probability:

$$\begin{aligned} &Pr_{\text{rea}}(\text{Collision}) = \\ &\quad Pr_{50}(\text{leftOHV}) \cdot Pr_{50}(\text{slowTraffic}) + \cdots \approx \\ &\quad 5 \times 10^{-3}\,, \\ &Pr_{\text{rea}}(\text{False Alarm}) = Pr_{50}(\text{leftHV}) + \cdots \approx 1.07\,. \end{aligned}$$

The calculated probability for false alarm is greater than 1 which is an invalid probability. The reason for that is that the assumption of the rare event approximation is not satisfied because the probability $Pr_{50}(\text{leftHV})$ makes leftHV certainly not a rare event.

Next, we assume that each of these transient faults is only relevant in exactly one of these 50 time steps. Applying the formula for $Pr_{\text{rea}}(H)$ on the minimal cut sets calculated earlier gives us these probability:

$$Pr_{\text{rea}}(\text{Collision}) = Pr_1(\text{leftOHV}) \cdot Pr_1(\text{slowTraffic}) + \cdots \approx 1.05 \times 10^{-4} ,$$
$$Pr_{\text{rea}}(\text{False Alarm}) = Pr_1(\text{leftHV}) + \cdots \approx 2.51 \times 10^{-2} .$$

The resulting collision probability is still quite high. The reason is that the ordering of faults is also important and that some faults are only relevant in certain steps. On the other hand, the probability of a false alarm is estimated too low. Assuming that each fault occurs only in one time step is too indulgent. Including details about fault activation ordering and better estimates of how often each fault must be activated into a formula would improve the resulting probabilities further. But this is far from trivial. This clearly shows that it is difficult to apply traditional methods on systems which have a complex logic and have a dynamically behaving environment. S# does not derive the probability from the minimal critical sets. Instead, S# derives the probability from the traces directly using the semantics shown in the previous section. This produces more accurate estimates and is especially useful when some of the faults are transient and only relevant in certain system states, e.g., the fault of a sensor in the main control is only relevant, when the main control is active. Also, when a controller detects a fault in a component and decides to switch to a spare component or to go into a degraded mode, this has a direct effect on the traces.

As already shown in Table 2, S# deduces the following results from the model:

$$Pr_{\text{S\#}}(\text{Collision}) \approx 2.00 \times 10^{-7} ,$$
$$Pr_{\text{S\#}}(\text{False Alarm}) \approx 5.45 \times 10^{-2} .$$

It is possible to make a better approximation with traditional methods, but these also require a deep knowledge of the exact order of fault activations which lead to a hazard. Fault probabilities cannot simply be multiplied with each other. For systems with a highly complex behavior, these approaches might be not feasible. As a consequence of looking into the traces directly, S# is not affected by this problem. Another source for the difference is that S# does not adhere to the "no miracles rule" of the fault tree analysis [33]. Simply put, the "no miracles rule" states that if one fault would prevent a more severe situation, then assume that this fault does not occur. Thus, when the first light barrier is defect, and its defect would prevent that the light barrier would not activate the height control system at all, which leads to no false alarm, then assume that the first light barrier works correctly in these cases. S# does not adhere to this rule, because it looks at the traces directly.

The precise quantitative analysis does not make the qualitative analysis redundant, because the minimal cut sets can help explain the results of the quantitative analysis and reinforce the validity of the results. The quantitative analysis of S# is not intended to replace traditional analysis, but to serve as an additional means for engineers. Its strength lies in the analysis of systems early in the development process. In early phases, a preliminary analysis of different design variants or different algorithms can be conducted with S#. In this phase, a traditional analysis would be too time-consuming and expensive to be applied on every variant because for each variant a separate fault tree needs to be created manually. The early rigorous analysis of design variants with S# can give differences in hazard probabilities even when traditional analysis would see no difference or expensive prototypes would be needed. Thus, S#'s quantitative analysis can make life easier for engineers. With S#, engineers do not need to know the interrelationship of different faults in these early phases. They just need to know the exact impact of each fault and S# derives the rest automatically.

6.4 Evaluation of S# analysis efficiency

Many safety analysis tools such as VECS, the Compass toolset, or AltaRica [5,27,28] rely on the standard approach of model transformations to use model checkers like SPIN, NuSMV, PRISM, or MRMC [8,17,22,24]. The Safety Analysis Modeling Language (SAML) is an extension of the PRISM input language to facilitate its application for the analysis of safety critical systems [27]. The safety analysis tool VECS transforms a SAML model directly into the NuSMV input language for qualitative analysis and into the PRISM input language for quantitative analysis. The qualitative analysis is based on the DCCA formula. The Compass toolset transforms SLIM models into the NuSMV input language [6]. Afterward, qualitative analyses are conducted by a fork of NuSMV in which symbolic algorithms specialized for minimal cut set generation are integrated. For the quantitative analyses, the procedure is even more sophisticated: The model is transformed to NuSMV first; afterward, the state space is exported to an intermediate file on which bisimulation is applied and probabilities are annotated by the tool SigRef. Finally, the resulting Markov chain is model checked by MRMC.

By contrast, S# unifies simulations, visualizations, and fully exhaustive model checking by executing the C# models with consistent semantics regardless of whether a simulation is run or some formula is model checked with LTSmin. Consequently, no model transformations are necessary, avoiding significant implementation complexity while retaining competitive model checking efficiency.

S# only has to execute C# code instead of understanding and transforming it, supporting most C# language features without any additional work; transformations, by contrast, would require large parts of the .NET virtual machine to be

encoded for model checking or to forgo many higher level C# features such as virtual dispatch or lambda functions.

The two main challenges of S#'s LTSmin integration were efficient state serialization and efficient handling of nondeterminism. The algorithm that allows ChoiceResolver to handle and track all combinations of nondeterministic choices, however, turned out to require only around 90 lines of C# code. To enable quantitative analyses based on Markov chains, only additional 30 lines of code had to be appended to the ChoiceResolver. Generating low overhead serialization methods, by contrast, is more involved, taking about 700 lines of C# code to generate the appropriate serialization methods at runtime. For the case study, serialization causes only around 5% of overhead during the entire model checking. The serialized states are smaller than the state vectors of a hand-optimized SPIN model of the height control, taking only 12 instead of 24 bytes per state. Another S# case study, the hemodialysis machine, has 71 variables but only requires 40 bytes per state.

In the worst case of valid formulas, S# and LTSmin have to enumerate the model's entire state space. For an earlier version of our model, S# required 68.8 s to enumerate 950,249 states and 40,197,857 transitions. 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. One reason for S#'s superior performance are automatic symmetry reductions [14] that allow S# to ignore irrelevant fault activations more efficiently than SPIN. These symmetry reductions enable S# to provide smaller models to LTSmin for model checking, increasing model checking efficiency noticeably; however, they can only be partially encoded into SPIN models, for full support, changes to SPIN would be required. For the case study, symbolic analysis with NuSMV, on the other hand, is faster than using S#: For a hand-written, very low-level and non-modular NuSMV model that is approximately equivalent to the S# model, the entire state space is generated almost instantly. However, some other S# case studies are 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.

S# models have a much higher level of expressiveness than either SPIN or NuSMV models, allowing variant modeling and analysis in a way that is not supported by either model checker directly. Additionally, S#'s explicit support for fault modeling guarantees conservative extension [13], i.e., faults only add or suppress system behavior when they are activated but cannot do so while they are dormant, which is important for adequate modeling and safety analyses. SPIN, NuSMV, PRISM, MRMC, or Zing, by contrast, cannot give this guarantee at a language level. While S# is more efficient than SPIN due to its fault optimizations, the increase in analysis time compared to NuSMV seems acceptable given the step-up in modeling flexibility, expressiveness, and fault modeling adequacy.

Analysis times are dominated by computing successor states, so we implemented an optimization that in many cases reduces the number of transitions by an exponential factor: Fault occurrences can be ignored for as long as they have no observable effects on the system state, cutting off large parts of the state space that subsequently do not have to be enumerated and analyzed [14]. Parts of these optimizations can also be applied for the quantitative analysis when the optimization has no impact on the probability measure.

Our findings, however, are solely based on our own experience with other modeling languages and other case studies analyzed with S#. For example, the railroad crossing case study available online in the S# repository is faster to check with S# than with NuSMV or VECS. 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 qualitative safety analysis that is equivalent to DCCA in 21 minutes using NuSMV instead of the 1.9 s it takes S# to do the same. We did not conduct a quantitative analysis of the railroad crossing with Compass. Of course, the comparison is unfair as forcing Compass semantics onto S# might likewise slow down analyses.

7 Conclusion and future work

S# provides an expressive C#-based modeling language for safety-critical systems and conducts fully automated DCCAs over these models to determine the minimal critical fault sets for all hazards. Also, S# can deduce the probability of hazards or other properties from the model using probabilistic model checking techniques. S#'s model execution approach not only has competitive analysis efficiency but also unifies model simulation and model checking to guarantee semantic consistency.

S# has a competitive edge over other approaches for safety modeling and analysis like Compass or VECS [27,28] by tightly integrating the development, debugging, and simulation of models with their formal analysis. Compared to other safety analysis tools [5,27,28], no model transformations are required by S# as LTSmin-based algorithms allow for executing S# models while model checking. To conduct quantitative analyses, Markov chains are generated on-the-fly by executing the model. Due to the step semantics of S#, the state space can be reduced notably, by being able to remove local

variables and a program counter from the state vector. This reduction makes the quantitative analysis of complex executable models feasible for explicit techniques. Competing approaches which transform models into the intermediate language of a model checker first cannot remove local variables in the intermediate models.

The qualitative safety analysis results of the case study match those from previous analyses [30]. These results could be augmented by quantitative results. Furthermore, the quality of the model could be improved thanks to S#'s modular modeling language and flexible model composition capabilities based on C# and .NET; manual work is no longer required for composing multiple modeled 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.

The quantitative analysis capability of S# opens up new possibilities. Currently, we are investigating more deeply how the exact probabilities of hazards correlate with the minimal cut sets in our models. We also analyze the effect of component quality, especially the case when the probability of one fault cannot be decreased without increasing the probability of another fault. Two faults of one component type might even be antagonistic, e.g., the probability of misdetections of a light barrier cannot be decreased without increasing the probability of false detections. The analyses in this paper are only a first step. Also, we are adding means to analyze models quantitatively, even when only some but not all fault probabilities are known. For this, we integrate Markov Decision Processes into S#. Then, S# can at least estimate the minimal and maximal probability of the hazard's occurrence. Furthermore, current work on our toolchain includes the integration of the Lustre language [16]. Lustre is a dataflow language, which is used for creating software for safety critical systems. This integration is a step toward offering modelers a means to create their models visually.

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.) Proceedings of the 16th International Conference Computer Aided Verification (CAV'04). Lecture Notes in Computer Science, vol. 3114, pp. 484–487. Springer (2004)
2. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. Dependable Secur. Comput. **1**(1), 11–33 (2004)
3. Baier, C., Ciesinski, F., Größer, M.: PROBMELA: a modeling language for communicating probabilistic systems. In: Proceedings of the 2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2004). pp. 57–66. IEEE (2004)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. 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)
6. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V., Noll, T., Roveri, M.: The COMPASS Approach: correctness, modelling and performability of aerospace systems. In: Computer Safety, Reliability, and Security, pp. 173–186. Springer (2009)
7. Butcher, J.: The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods, 2nd edn. Wiley, Chichester, UK (2003)
8. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an open source tool for symbolic model checking. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002)
9. Diez, D.M., Barr, C.D., Çetinkaya Rundel, M.: OpenIntro Statistics. OpenIntro, Inc., (2015)
10. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. **73**, 110–132 (2014)
11. Habermaier, A.: Design Time and Run Time Formal Safety Analysis using Executable Models. Ph.D. thesis, University of Augsburg (2017)
12. Habermaier, A., Eberhardinger, B., Seebach, H., Leupolz, J., Reif, W.: Runtime model-based safety analysis of self-organizing systems with S#. In: 2015 Self-Adaptive and Self-Organizing Systems Wsh.s. pp. 128–133. IEEE (2015)
13. 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)
14. Habermaier, A., Knapp, A., Leupolz, J., Reif, W.: Fault-aware modeling and specification for efficient formal safety analysis. In: ter Beek et al.[32] , pp. 97–114
15. Habermaier, A., Leupolz, J., Reif, W.: Unified Simulation, Visualization, and Formal Analysis of Safety-Critical Systems with S#. In: ter Beek et al. [32], pp. 150–167
16. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)
17. Holzmann, G.: The SPIN Model Checker. Addison-Wesley, Boston (2004)
18. ISO: ISO/IEC 23270: Information technology – Programming languages –C#(2006)
19. ISO: ISO 24765: Systems and software engineering – Vocabulary (2010)
20. ISO: ISO/IEC 23271: Information technology – Common Language Infrastructure (2012)
21. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer (2015)
22. Katoen, J.P., Zapreev, I., Hahn, E., Hermanns, H., Jansen, D.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. **68**(2), 90–104 (2011)
23. Kirsch, C., Sengupta, R.: The evolution of real-time programming. In: Handbook of Real-Time and Embedded Systems, chap. CRC Press, (2007)
24. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification, Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011)

25. Laarman, A., van de Pol, J., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Formal Methods in Computer Aided Design, pp. 247–255 (2010)
26. Leveson, N.: Engineering a Safer World. MIT Press, Cambridge (2011)
27. 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)
28. Noll, T.: Safety, dependability and performance analysis of aerospace systems. In: Formal Techniques for Safety-Critical Systems, CCIS, vol. 476, pp. 17–31. Springer (2015)
29. Object Management Group: OMG Systems Modeling Language, Version 1.4 (2015)
30. Ortmeier, F., Schellhorn, G., Thums, A., Reif, W., Hering, B., Trappschuh, H.: Safety analysis of the height control system for the Elbtunnel. In: Computer Safety, Reliability and Security, Lecture Notes in Computer Science, vol. 2434, pp. 296–308. Springer (2002)
31. Parker, D.: Implementation of symbolic model checking for probabilistic systems. Ph.D. thesis, University of Birmingham (2002)
32. ter Beek, M.H., Gnesi, S., Knapp, A. (eds.): Proceedings of the Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems (FMICS-AVoCS'16), Lecture Notes in Computer Science, vol. 9933, Springer (2016)
33. Vesely, W., Dugan, J., Fragola, J., Minarick, J., Railsback, J.: Fault Tree Handbook with Aerospace Applications. Tech. rep, NASA (2002)
34. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)