











# Towards Safe Autonomous Driving: Model Checking a Behavior Planner during Development

Lukas König<sup>1</sup><sup>(✉)</sup>, Christian Heinzemann<sup>1</sup>, Alberto Griggio<sup>2</sup>,  
Michaela Klauck<sup>1</sup>, Alessandro Cimatti<sup>2</sup>, Franziska Henze<sup>3</sup>,  
Stefano Tonetta<sup>2</sup>, Stefan Küperkoch<sup>1</sup>, Dennis Fassbender<sup>3</sup>, and  
Michael Hanselmann<sup>1</sup>

<sup>1</sup> Robert Bosch GmbH, 70465 Stuttgart, Germany  
`firstname.lastname@de.bosch.com`

<sup>2</sup> Fondazione Bruno Kessler, 38122 Trento, Italy  
`griggio,tonettas,cimatti@fbk.eu`

<sup>3</sup> CARIAD SE, 38440 Wolfsburg, Germany  
`firstname.lastname@cariad.technology`

**Abstract.** Automated driving functions are among the most critical software components to develop. Before deployment in series vehicles, it has to be shown that the functions drive safely and in compliance with traffic rules. Despite the coverage that can be reached with very large amounts of test drives, corner cases remain possible. Furthermore, the development is subject to time-to-delivery constraints due to the highly competitive market, and potential logical errors must be found as early as possible. We describe an approach to improve the development of an actual industrial behavior planner for the *Automated Driving Alliance* between Bosch and Cariad. The original process landscape for verification and validation is extended with model checking techniques. The idea is to integrate automated extraction mechanisms that, starting from the C++ code of the planner, generate a higher-level model of the underlying logic. This model, composed in closed loop with expressive environment descriptions, can be exhaustively analyzed with model checking. This results, in case of violations, in traces that can be re-executed in system simulators to guide the search for errors. The approach was exemplarily deployed in series development, and successfully found relevant issues in intermediate versions of the planner at development time.

**Keywords:** Autonom. Driving · Model Checking · Industry Application

## 1 Introduction

*Automated Driving (AD)* is an ever-growing research field with the potential to make traffic safer and more efficient. Recently, ISO 21448 on Road Vehicles – *Safety of the Intended Functionality (SOTIF)* specified that the number of unsafe, both known and unknown, scenarios should be minimized [30] and the political goal “Vision Zero” aims to practically eliminate traffic fatalities by 2050 [58]. This demonstrates that there is a high interest in and pressure on research to make automated vehicles (AV) safe. AD in the sense of a (future) product has to comply with a vast variety of safety requirements originating from

domains as diverse as physics, law and ethics [4, 21, 37, 40, 58]. Improving safety up to human driving level is already a challenging problem [51]. However, it is assumed that AD needs to vastly exceed the “human” safety benchmark since even very few fatalities caused by automated vehicles are hardly acceptable to the public [2, 49, 61]. The de facto standard today is that tremendous amounts of test drives are supposed to account for the safety of AVs. However, statistical considerations show that deriving confidence in the safety of an AV solely via test drives might indeed require ludicrous amounts of driving [34, 44, 62].

In this paper, we describe an approach to improve the development of AD software, adopted within the *Automated Driving Alliance (Alliance)*<sup>1</sup> between Bosch and Cariad. Specifically, we consider a *behavior planner (BP; also, tactical BP)*, that controls the high-level actions of an AV (e. g., accelerating, braking, lane changes) based on the perceived state of the environment (e. g., flow of the surrounding traffic). It is part of a system called *Highway Pilot* which realizes automated driving on highways or highway-like roads. The BP is implemented in C++, and is under active development, undergoing repeated updates, with the addition of new features and improvements. Due to time-to-delivery constraints, the *verification and validation (V&V)* activities are supposed to proceed in parallel to the development, preferably in an “observe only” manner.

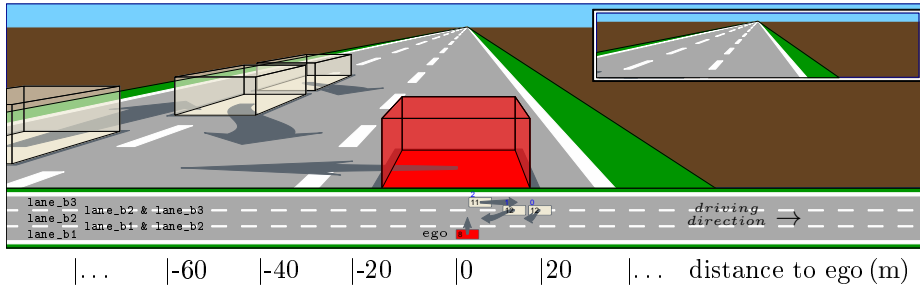
We enhance the V&V process by integrating automated formal verification techniques, in particular *model checking (MC)* of infinite-state transition systems [10], within the original development environment. The primary purpose is not (yet) to provide arguments for absolute correctness, but to increase the coverage of known unsafe scenarios, currently provided by test drives and simulation. By exhaustively analyzing a huge range of scenarios, MC is largely insusceptible to human bias and can discover corner cases that may be overlooked otherwise.

We face the challenge that MC must become part of the *continuous integration (CI)* process, hence it must be directly connected to the consecutive versions of the BP. This means that manual modeling is to be avoided, in order to enable a seamless connection between development and V&V. Therefore, we integrate suitable mechanisms for the extraction of the BP logic directly into the development environment. Starting from the C++ code of a BP, we automatically derive a model of the underlying logic including the interface to the surrounding software stack. This model is converted into K2, a low-level imperative-style language of the Kratos2 software model checker [23], which is, in turn, converted into SMV, the input language of the symbolic model checker nuXmv [8].

The scenarios for validation are obtained by composing the model of the BP in closed-loop with an *environment model (EM)*. The EM contains rules about the succession of a highway-like traffic scene in terms of the *possible* behavior of a set of free cars which drive in scope of a distinguished BP-controlled car (*ego*), cf. Fig. 1<sup>2</sup>. EM and BP are integrated such that the BP receives perception

<sup>1</sup> The Alliance has set out to “[build] a state-of-the-art ADAS software platform for use in all Volkswagen Group brand vehicles – and therefore in *one of the world’s biggest vehicle fleets*”. More information on BOSCH and CARIAD websites.

<sup>2</sup> The graphic was auto-generated by our explainability toolchain, cf. Sec. 3.1.



**Figure 1: Highway scene illustrating the base scenario.** We model three straight lanes unrestricted in longitudinal direction, non-ego positions are calculated relative to ego. Laterally, cars are either on a single lane or between two lanes (during a lane change). Blue numbers: car IDs, black numbers: velocities in  $m/s$ . Grey arrows: movement from current to next EM iteration.

inputs from the EM and returns an actuation output, which is translated into the movement of the ego. The EM is also written in SMV and can be parametrized to further specify the scenario (e. g., number of non-ego cars, physical features of the cars, driving behavior). The composed model of EM/BP is exhaustively analyzed with nuXmv, which results in a *counterexample (CEX)* in case of the violation of a property, e. g., a collision. The CEX can be re-executed in simulators to guide the search for errors. The approach was exemplarily deployed in series development, resulting in a fully automatic toolchain usable, e. g., in a CI system. The deployment was very successful in that actually relevant issues could be found in intermediate versions of the BP at development time.

Since the full BP code is restricted from publication, we use a mock BP in this publication to illustrate details about the process. Nonetheless, all presented results have been originally obtained with the actual BP used in the Alliance by tracking its development with CI techniques. Though being much simpler, the mock BP is designed to resemble the actual BP in some essential aspects, such that two major bugs found in the actual BP can be reproduced with it. Runtime and performance analyses are performed on data from the actual BP. Using the mock BP, we provide possible fixes to the found bugs in this simpler setup, and show that the model checker then efficiently supplies proof for the now correct functioning of the model of the mock BP, within the simulation by the EM.

Our contributions are:  $\langle 1 \rangle$  a self-consistent toolchain, involving automatic processing of the C++ code of a BP, integration with plausible physics and surrounding traffic behavior, MC with nuXmv, as well as, extracting traffic scenes from CEXs for debugging;  $\langle 2 \rangle$  presentation and discussion of two safety-relevant issues found in an industrial BP;  $\langle 3 \rangle$  by means of re-simulation ensuring that the model soundly captures the real-world system;  $\langle 4 \rangle$  analysis of the feasibility of the approach for an actual industrial context, including efficiency.

The remainder of the paper is structured as follows. Sec. 2 provides the basic AD context, as well as the theoretical background required for MC. Sec. 3

describes the methodology underlying the experimental setup. Sec. 4 describes and discusses the experimental results. Sec. 5 lists and assesses literature related to our approach. Sec. 6 provides a conclusion and an outlook to future work.

Additionally, we provide an appendix which is available in an extended version of the paper published as supplementary material. It is intended to facilitate reproducibility, but is not required to follow the presented results. The supplementary material also contains the mock BP and two versions of the EM, as used for the experiments. In an artifact associated to this paper, we provide the full functioning toolchain, i. e., all code and software (except for the Alliance BP) in a state as used for the experiments.

## 2 Background

The task of driving automatically can typically be segmented according to the classic “sense – plan – act” paradigm [7, 38]. The *sense* part comprises perception of the environment using sensors like camera, lidar or radar, and fusing their measurements into a *model of the environment* (e. g., [41]). This model contains all available information of the AV’s surroundings, e. g., lanes to drive on, the state of other traffic participants (e. g., position, velocity, acceleration) and predictions of their motion. This is the basis for planning the motion of the controlled AV. The *plan* part can be divided into three steps [3]: First, a *strategic planner* decides about the global navigation, i. e., the route to follow. Then, the *tactical BP* decides between available maneuvers, e. g., lane following or lane change. Finally, the *trajectory planner* calculates a desired trajectory which eventually results in a sequence of desired accelerations and curvatures. In the *act* part, these signals are forwarded to the actuators, thus accomplishing the actual driving on the road. We focus on the tactical BP which, for MC, is decoupled from the other software modules in the sense – plan – act pipeline.

**Model Checking of Infinite-State Symbolic Transition Systems.** The system under analysis is derived in several steps from the BP and the EM code, cf. Sec. 3, and finally represented as a symbolic transition system expressed using quantifier-free formulæ in first-order logic modulo theories (for further reading, refer to, e. g., [5]). We work in the setting of many-sorted first-order logic, and we assume the usual first-order notions of interpretation, satisfiability, validity, logical consequence, and theory, as given, e. g., in [17]. Unless otherwise stated, when we talk about a logical formula  $\varphi(X)$ , we mean that  $\varphi$  is a quantifier-free first-order formula whose free variables are included in the set  $X$ , and using symbols from the theory of (linear) arithmetic (with their usual interpretation). For example,  $\varphi(\{x_1, x_2\}) := (3x_1 > 0) \wedge (x_2 + x_1 < -5)$ . A symbolic transition system  $S = \langle X, I, T \rangle$  is a tuple, where  $X$  is a set of (state) variables,  $I(X)$  is a formula representing the initial states, and  $T(X, X')$  is a formula representing the transitions, where  $X'$  is the set of variables representing the next state of the system. A state  $s$  of a transition system  $S$  is an assignment to the state variables  $X$ ; a path (trace)  $\pi$  of  $S$  is a possibly infinite sequence  $\pi := (s_0, s_1, \dots, s_i, \dots)$

of states  $s_i$  such that  $I(X)$  is true under the assignment  $s_0$ , and  $T(X, X')$  is true under the assignment  $s_{i-1}, s'_i$  for all  $i > 0$  in  $\pi$ , where  $s'$  is the assignment obtained by replacing each  $x \in X$  with the corresponding  $x' \in X'$ . We say that a state  $s$  is reachable in  $S$  if and only if there exists a path  $\pi$  of  $S$  such that  $s \in \pi$ . Given a formula  $P(X)$  over the variables  $X$ , the invariant verification problem for  $S$  and  $P$  is the problem of checking if all the reachable states of  $S$  satisfy the formula  $P$ . In that case, we say that  $S$  satisfies  $P$ , written as  $S \models P$ .

**Tools.** For solving the core MC problem we use nuXmv [8], a state-of-the-art symbolic model checker for both finite- and infinite-state transition systems. It supports the verification of invariant and LTL properties using a combination of efficient algorithms based on Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT) [11, 13, 24]. Each verification engine can be used unbounded or bounded (in which case only disproving is possible). For details see the documentation [8]. The transition system for nuXmv is defined using an extension of the standard SMV language for finite-state systems (simply SMV in the following). An example of the syntax is reported in Alg. 2 of the appendix.

To automatically derive a nuXmv transition system out of imperative C++ code, Kratos2 can be used as an intermediate step. Kratos2 [23] is a tool for the automatic verification of imperative programs, using nuXmv as its main verification engine. The native language of Kratos2 is a verification language called K2 (similar to Boogie and Why3 [43]) that provides a well-defined and unambiguous formal semantics suitable for verification. An example program in K2 is shown in Alg. 3 of the appendix. The resulting K2 program is then transformed by Kratos2 into SMV and verified with nuXmv.

Parsing C++ code and creating the K2 model out of it is done by a prototypical software called vfm which was developed within the Alliance (cf. artifact).

### 3 Methodology

This section describes the proposed workflow and justifies how the respective components were chosen to establish an industry-ready setup.

#### 3.1 Overview

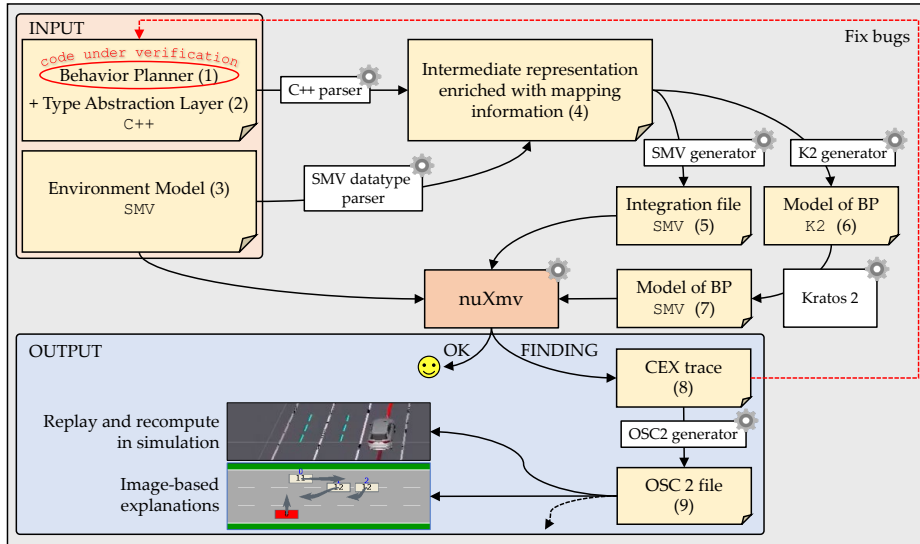
Our toolchain consists of several components centered around a BP under analysis, cf. Fig. 2. We impose MC on top of a functioning software and in scope of an established development process. Therefore, the BP’s source code, (1) in the figure, is considered largely *immutable*, meaning that we cannot change it to our likings, but developers *can* change it at any time outside of our control.

The BP cannot be checked on its own, since its behavior *within a traffic situation* is of major interest. The entity responsible for providing an initial traffic scene, and keeping track of how it evolves according to the BP’s actions, is the EM (3). It is given in SMV language, i. e., we can directly identify it with a transition system  $S_E := \langle X_E, I_E, T_E \rangle$ . In a first processing step, the BP and

the EM are parsed separately to create an intermediate representation (4) which contains an internal description of the BP logic including its interface towards the EM. The respective mapping between EM and BP is provided by the *type abstraction layer (TAL)* (2), which is embedded into the C++ code of the BP via comments (see below). The intermediate representation is translated into a K2 version of the BP logic (6) which, in turn, is translated by Kratos2 into an SMV representation (7; cf. discussion in Sec. 4.3 about making this detour instead of directly translating from C++ to SMV); we identify the SMV representation with the transition system  $S_P := \langle X_P, I_P, T_P \rangle$ ; additionally, the interface information is used to generate the full integrated transition system, called  $S_{E \sqcup P}$ . On SMV level, it is constructed to include and connect  $S_E$  and  $S_P$  as two separate modules within a main module (5); it is also the place to add specifications to check. This file is then handed to the nuXmv model checker for the actual MC task. Depending on the result, we either terminate the process after MC, if the specifications are fulfilled (“OK”), or otherwise (“FINDING”) trigger the creation of a CEX (8), which is a witness of the specification being violated. It provides a trace within the checked transition system  $S_{E \sqcup P}$  in the course of which the specification does not hold. In our case, the sequence of variable values along this trace provides information about  $\langle 1 \rangle$  the traffic situation in which the violation occurred and  $\langle 2 \rangle$  the BP actions in this situation.  $\langle 1 \rangle$  is used to generate visualizations of the traffic scene in question, and to further process the resulting scenarios of interest, by using the *Open Scenario 2 (OSC2)* format [1] as underlying representation (9). Figs. 1, 4 and 5 were created using this functionality.  $\langle 2 \rangle$  *could*, in principle, be used for a deep analysis of the error, such as tracking back which lines in the original C++ code are involved in the violation. This is planned to be done in future, but is not possible yet.

The retrieval of the transition system to check needs to be flexible enough to adapt to future changes of the BP, which are expected to happen frequently during development. Therefore, the EM should not be customized towards a specific BP, but rather provide a generic interface which works with a variety of BP instances. The toolchain then runs fully automatically by attaching flexibly to different BPs, as long as the EM is “suitable” for them, i. e., all required data is available in the EM’s generic interface, agnostic of naming. For example, the BP might require as input its own velocity in a variable called `agent.v` which the EM provides as `ego.v`. The TAL connection can be established by adding an “aka” tag as comment to the BP variable `agent.v` (for details cf. the appendix).

Therefore, the remaining manual effort consists of adding or adapting the TAL information when the BP changes. This is expected to be mostly a one-time effort, since once the connection to EM variables is established, it only needs to be changed if new data is added to the interface (*not* on mere re-usage or renaming). This happens occasionally, but is not very common in practice. Only if the EM becomes unsuitable for a developed BP, i. e., when a non-existent signal is requested, this implicates the fairly high effort of adapting the EM.



**Figure 2: Overview of the presented toolchain.** The BP to check (1) is the main input. The EM is given as transition system in native SMV language (3). nuXmv (center) is run on an integration of BP/EM created in several steps (3/5/7). Visualizations and simulations are derived from CEXs (9).

### 3.2 Environment Model

Since we cannot model-check the full software stack the BP is part of, with perception on one side and actuation on the other, the BP needs to be directly fed with mock data from a simulated environment, and its output needs to be propagated back into this environment in a closed-loop manner. This is accomplished by the EM, which creates an initial state of the environment, and then progresses by supplying the current state as perception input to the BP and deriving the next state based on its output. Therefore, the EM maintains physical states of all agents, and applies laws of physics and possible driving behavior to them.

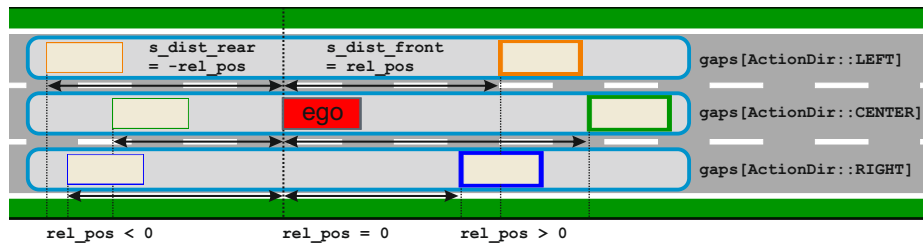
We assume a *perfect environment*, i. e., perfect knowledge without any sensor uncertainties and perfect behavior without actuator imprecision. We use a highway-like road model with 3 lanes, allowing traffic only in one direction, as illustrated in Fig. 1. We support an arbitrary number of non-ego vehicles<sup>3</sup>, each of which has physical properties such as relative position to ego, velocity and lane association. A vehicle is associated to either a single lane or, during a lane change, to both its source and its target lane (cf. `veh[1]` and `veh[2]` in Fig. 1).

In each verification step, corresponding to 1s, all non-ego vehicles may choose a new acceleration  $a \in \{-8, -7, \dots, 6\}$  m/s<sup>2</sup>, which is used to update the velocity and relative position simultaneously. In addition, each non-ego vehicle may

<sup>3</sup> A generator is used to create an EM with an adjustable number of non-ego cars for each MC run (cf. artifact).

choose to perform a lane change. A lane change is executed in two stages. In the first stage, the non-ego vehicle signals the lane change via turn indicators but is still located on its source lane. The second stage is the transition from source to target lane, i. e., it starts when the lane marking is initially touched until the non-ego vehicle is contained entirely in the target lane. During both stages, the non-ego vehicle may non-deterministically decide to abort the lane change. If the decision to abort is made while being in the second stage, the non-ego vehicle moves back to the source lane. The durations of both stages of the lane change and of the return to the source lane when aborting are chosen non-deterministically from intervals of possible values (cf. section marked “**Begin of lc parameterization**” in the published version of the EM). Thereby, we enable to verify a high variety of lane changes in the MC process, including much tougher ones than expected to usually happen on real roads, with the goal to over-estimate (rather than under-estimate) violations by the planner in the sense discussed in Sec. 4.3. On the other hand, we do prohibit excessively malicious behavior by ensuring that ego is able to prevent a collision by staying in its lane and reacting instantaneously with a deceleration of up to  $-8\text{m/s}^2$ .

The ego vehicle tracks objects in its proximity via so-called *gaps*, defined by a front and a rear vehicle on its own and its neighboring lanes, cf. Fig. 3. This data needs to be provided by the EM, as well. Depending on lane availability, there can be up to three gaps, one is always in the ego lane, the other two can be in the left or right lane next to ego, respectively, if available. For each car in the gaps, information such as relative distance to ego, velocity and acceleration are stored which can be used by the BP for decision making. If one of the positions is not filled, e. g., if the next car is out of perception range, this is indicated by a special value. The full SMV code of the EM, as well as a more in-depth description are published as supplementary material.



**Figure 3: Illustration of gaps tracked for ego.** The *gap data structure* provides information about the free space which ego drives in on its own lane or could dive into when performing a lane change, in terms of the two cars limiting this space to the front and rear. A gap contains the IDs, distances, velocities, accelerations etc. of the closest cars to the front and rear on the respective lane.



### 3.3 The Original and Mock BPs

We consider two BPs,  $\langle 1 \rangle$  the actual BP currently under development for the Alliance project, and  $\langle 2 \rangle$  the mock BP which is a simplified version of  $\langle 1 \rangle$ . The code considered for  $\langle 1 \rangle$  is an excerpt of the project BP containing the logic for a *lane change decision towards the “fast” lane (LCfast)* (left assumed; as opposed to *LCslow*). It consists of more than 1000 lines of C++ code. The mock planner  $\langle 2 \rangle$  is much smaller (about 100 lines) and can be inspected in its entirety as supplementary material. It contains a simple version of the logic for LCfast, logic for LCslow and a simple longitudinal control.

The interface towards the EM is equal for  $\langle 1 \rangle$  and  $\langle 2 \rangle$  regarding LCfast. It uses the *gap structure* as input and returns as output the decision whether or not to initiate a lane change towards the fast lane. The mock BP requires additional data for the longitudinal control, as well as some signals used to fix the found issues; we mostly disregard these differences in the following, for simplicity. The exact interfaces are given in the appendix.

## 4 Experiments

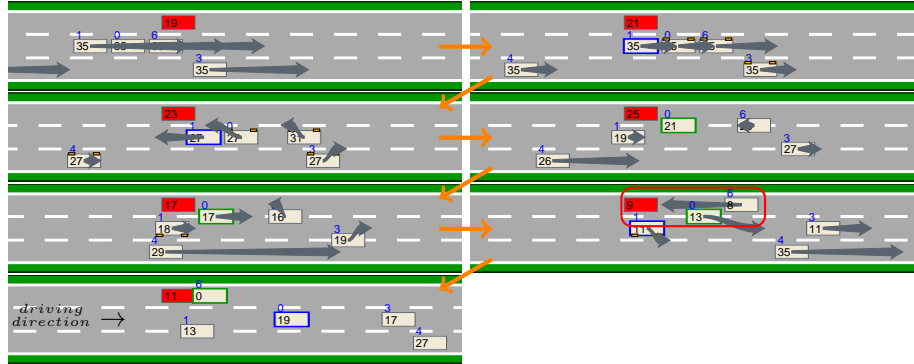
Our goal is to demonstrate that the proposed approach can practically guide development processes of a BP in industry. Therefore, we need to show that  $\langle 1 \rangle$  our setup derives valuable insights for  $\langle a \rangle$  development and/or  $\langle b \rangle$  release;  $\langle 2 \rangle$  it does this in acceptable runtime,  $\langle 3 \rangle$  it does not overly disturb every-day development, and  $\langle 4 \rangle$  its results are self-explanatory to developers and V&V experts. In Sec. 4.1 we analyze two major issues found early during development in the Alliance BP  $\langle 1a \rangle$ . We also comment on the implications of proofs of error-freeness on the model level, which may be of high relevance for possible future release argumentation  $\langle 1b \rangle$ . Efficiency of bounded and unbounded MC  $\langle 2 \rangle$  is analyzed in Sec. 4.2. Sec. 4.3 discusses the results and elaborates qualitatively on the topics  $\langle 3 \rangle$  and  $\langle 4 \rangle$ .

### 4.1 Disproven Specifications with Counterexamples

The two issues we describe were found by checking for the invariant property `!blamable_crash`. It is true if ego’s bounding box never overlaps with the bounding box of any car in front of ego. nuXmv showed in both cases that it does not hold true, which led to the generation of CEXs. They reveal violations of the BP in typical highway-traffic, which we named *Lead Vehicle Occlusion* and *Double Merge*<sup>4</sup>. The CEXs have been further processed to show the succession of the traffic scenes that lead to the violations (Figs. 4 and 5), and to confirm the issues in simulation by using the full underlying software stack (cf. Sec. 4.3).

---

<sup>4</sup> Note that these issues certainly could have been detected with regular V&V techniques, as well, however, with considerably greater effort.



**Figure 4: Full CEX trace of crash caused by ego overlooking hard brake ahead due to another car “cutting out”.** The highlighted portion of the second-to-last image shows how the car in the lead position (0) pulls away (“cuts out”) while another car ahead (6) brakes hard, unnoticeable by ego due to the way lead vehicles are tracked. Instead of braking, ego even accelerates, since car 0 also accelerates and departs.

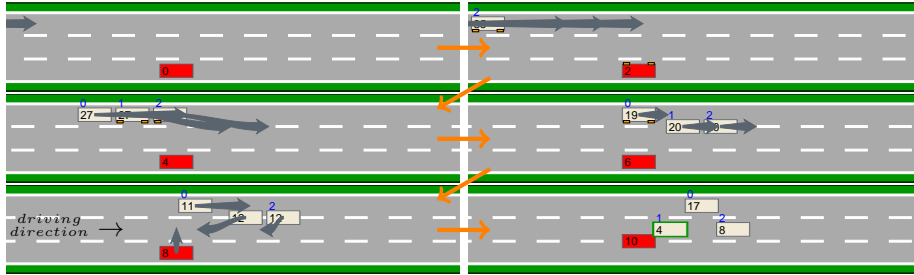
**Lead Vehicle Occlusion.** Since ego bases all decisions on the *gap structure*, it cannot “see” possible cars in front of the one it finds as front car in one of the gaps. On the other hand, the surrounding traffic is allowed to behave arbitrarily rudely, as long as ego has, upon immediate action, a chance to avoid a crash (cf. Sec. 3.2). Considering this, nuXmv reported the situation illustrated in Fig. 4 as a CEX<sup>5</sup>. The seven snapshots correspond to the full path given in the CEX (i. e., it takes at least 7 steps from an initial to a violating state). The actual crash occurs in the last step, but the underlying wrong decision is made already in the sixth step. Here, car 0 is partially on ego’s lane, and is, therefore, considered the lead car in the CENTER gap (indicated by a green frame). On the other hand, car 6 is farther ahead, and, at this point invisible to ego (in a logical, *not* a perception sense). The problem arises from car 6 performing a hard brake, while car 0 budges towards the middle lane (“cuts out”). Considering car 0 as lead car, which actually departs to the front, ego misinterprets the scene to clearing up, and accelerates rather than braking, which leads to the crash. The CEX points to a fundamental problem caused by considering only a single lead vehicle per gap. (Indeed any constant number  $n$  of lead vehicles won’t fix the underlying problem, assuming unrestricted velocities and/or restricted braking capabilities. Imagine a row of  $n + 1$  cars in front of ego, with the furthest one standing still while the other  $n$  cut out. In practice, safety distance needs to be adjusted such that braking in time is possible even in the worst case scenarios.)

**Double Merge.** This issue is caused by ego observing only cars on neighboring lanes, which can lead to conflicts if cars from two lanes apart change towards

<sup>5</sup> The CEX can be produced with any number of non-ego cars  $\geq 2$ . We choose to show non-minimal examples here to give intuitive insight into the functioning of the EM.

a neighboring lane while ego itself is changing towards this lane. This issue is related to the gap structure, too, but points to the lateral, rather than longitudinal limit of the structure. Fig. 5 shows how the violation unfolds in the course of 6 steps. The actual problematic decision already occurs in step 2 where ego decides to change the lane towards the middle, as displayed by the indicators turned on<sup>6</sup>. At this point, ego waits for two steps before actually starting the lane change. During this whole period, the middle lane appears to be free (although all three non-egos indicate at some point to be starting a lane change, which is actually performed by cars 1 and 2, but aborted by car 0). When ego actually does the lane change, car 1 happens to finish its own and ends up colliding with ego. While it could be argued that the other car could have avoided this situation, too, ego is at least partially to blame for the crash.

As opposed to the Lead Vehicle Occlusion issue, which was more fundamental in nature, this issue is strongly connected to the exact way the BP performs a lane change. Having a more flexible cancellation mechanism, or including cars *between* a neighboring lane and the one next to that into the gaps, could, for example, easily avoid this type of issues.



**Figure 5: Full CEX trace of crash caused by ego not noticing a car merging towards the middle lane while itself merges there.** The issue is caused by ego not looking further than one lane to the left when deciding to change a lane, and delaying the actual lane change after the decision for 2 steps.

**Comment on Specifications Proven to Hold.** We used unbounded MC for validating the EM by proving many of its desired properties to hold; for example, “ego is never ‘forced’ into a collision by a non-ego vehicle”, and “the vehicles in the gaps are always the ones closest to ego on the respective lane”. Also, when fixing the two issues described above, the mock BP can be shown to fulfill the `!blamable_crash` specification, cf. the artifact published with this paper. However, it is currently unclear what such proofs of *absence* of errors on the model level mean for the real system, see discussion in Sec. 4.3.

<sup>6</sup> The actual Alliance BP, whose behavior is shown here, supports lane changes upon driver request, while the mock planner performs lane changes towards the fast lane only to overtake a slower vehicle. Thus, the issue can, for the Alliance BP, be produced with at least one, for the mock BP with at least two non-ego vehicles.

## 4.2 Runtime Analysis

For the runtime analysis we are particularly interested if the model checker terminates within a time limit “acceptable” for typical aspects of the V&V workflow utilized in the Alliance (and probably similarly in other AD projects). We agreed that for the following standard situations the respective approximate runtimes would be acceptable. For runs performed on each *pull request (PR)*, e. g., as gatekeepers:  $\approx 2$  min. For runs performed during each *nightly job (NI)*:  $\approx 4$  h. For runs performed for a *release (RE)*: 5 d and more, may be acceptable.

Assuming an ever-changing development setting, it seems misguided to compare rigorously clocked runs of specific software versions to find the best EM/BP setup. To present a broader picture, we rather list the average runtimes of all runs made during a fairly mature phase of the study with somewhat changing BP and EM versions. This mature phase is defined as starting at the point from which no major fixes to the EM occurred anymore. We do not analyze the differences in implementations, but deliberately provide by this means a general impression of practicability, in a setting close to how it is expected to emerge in real industrial contexts.

We focus on the checked property `!blamable_crash`, using the Alliance BP (i. e., the expected result is a CEX for all runs; its creation time is included in the listed times) and do not restrict the behavior in any way. Tab. 1 summarizes the runtimes obtained for up to 10 non-ego vehicles with bounded or unbounded MC. We estimate many of the numbers in the table for the sake of a comprehensive, rather than overly exact, image, as they reflect our experience very well. Note, however, that the results only include runs that terminated at all (as opposed to runs that needed to be aborted, usually due to excessively long runtime). This particularly distorts the rows with three and four non-ego cars and unbounded MC, which – especially with earlier versions of the EM – frequently ran for days and weeks without finishing. Considering future changes to the BP, these rows need to be taken cautiously, although we also expect further improvements to the EM, which, in the past, greatly reduced runtime (cf. Sec. 4.3).

The general takeaway is that the overall setup is sufficiently efficient to be used in an industrial context, largely even at PR level. For future release argumentation, using unbounded MC seems to be possible with up to 4 non-ego cars. Note that adding non-ego cars beyond two did not gain any new insight into issues of the BP in all runs so far. Discussions are ongoing which number of non-ego cars is sufficient to reflect all relevant situations on a straight 3-lane highway, in terms of BP logic, with 4–5 being among the highest estimates.

Considering further improvements, the setup seems to be extendable towards more complex road topologies. However, we experienced the runtimes to be fairly volatile. It may well happen that marginal changes to the BP and/or the EM yield a factor of 2 – 3 in runtime (cf., e. g., the high variance for the “2-car unbounded” case). While this is still acceptable for our context in many regards, it needs to be considered for each individual case, cf. also discussion in Sec. 4.3.

**Table 1: Runtimes for MC runs clustered along number of non-ego cars and bounded vs. unbounded MC.** The “bounded” rows comprise runs that were not all tracked and analyzed exactly, but overall yield a fairly clear picture (runtimes estimated here; as well as for the 2-car unbounded row). The “Suitable for” row is a best guess based on the given numbers.

# Non-egos	MC type	# Finished Runs	Average runtime	Suitable for
1	bounded	> 50	≈ 1 min	PR/NI/RE
2	bounded	> 30	≈ 1 min	PR/NI/RE
3	bounded	≈ 10	≈ 2 min	PR/NI/RE
4	bounded	≈ 10	≈ 2 min	PR/NI/RE
5	bounded	≈ 10	≈ 5 min	(PR)/NI/RE
10	bounded	2	19.6 min	NI/RE
1	unbounded	6	30 s	PR/NI/RE
2	unbounded	> 50	≈ 1–7 h	(NI)/RE
3	unbounded	2	16.6 h	RE
4	unbounded	1	5.7 d	(RE)

### 4.3 Discussion

The presented results indicate that MC can be used in a real industrial context to guide the iterative development of a BP. The approach is overall suitable to be used on top of the regular development processes for AD without considerably interfering with them, and to guide this development by detecting safety-relevant issues earlier than with plain testing, thus reducing futile testing time. It is important to recognize the complete lack of human bias in this process. Particularly, the type of scenario is not provided in any way, but all scenarios producible by the respective EM/BP combination are inspected. The toolchain works mostly self-governed by automatically extracting the BP logic from C++, and creating self-explanatory visualizations and test cases out of CEXs.

**Current Limitations.** *Relevance for the real world:* MC provides either ⟨1⟩ a proof that the (extracted version of the) BP logic complies with a given specification (in relation to the EM), or ⟨2⟩ a proof that this is not the case, which is complemented with a CEX. These proofs *relate* to statements about the correctness of the BP behavior, i. e., the BP is supposed to be “correct” ⟨1⟩ or “incorrect” ⟨2⟩ w. r. t. to the checked specification when steering an actual car on the road. However, full confidence in these statements requires not only to prove that the EM itself as well as the extraction of the BP logic are correct, but also that the EM reflects real-world traffic, physics and perception/actuation “adequately”. Assuming any deviation from the real behavior of the system means that the EM would need to over-estimate violations for ⟨1⟩ and under-estimate them for ⟨2⟩. Therefore, it is not possible to provide an EM which simplifies the actual system behavior in any way, and evidently entails only true statements about both correctness and violations of the BP. It may be possible, though,

to strengthen one direction up to a level where some notion of confidence in its validity can be derived. Then, it appears reasonable to let the EM over-estimate, rather than under-estimate violations such that error-freeness  $\langle 1 \rangle$  is reflected accurately. This is due to the additional information provided by CEXs for  $\langle 2 \rangle$ . While the proof of error-freeness is a somewhat final statement, false positives of violations *can* be ruled out by re-simulating the CEXs with the actual software stack. This additional check was performed for all the CEXs presented in this paper. The EM is built with the *intent* to rather over-estimate than under-estimate violations, cf. Sec. 3.2. However, at present we do not investigate more profoundly to what extent this is actually accomplished and what exactly, consequently, a proof of correctness implies on BP level.

*Number of non-ego cars:* According to the results presented in Sec. 4, a natural measure of performance of the approach is the maximum number of non-ego cars that can be processed in a time acceptable for one of the presented use cases. For unbounded MC, this number is currently limited to about 4 non-ego cars, if considering the “release” use case. Using bounded MC, up to 10 non-ego cars can be processed quite efficiently, i. e., easily suitable for a regular “nightly job”. These limitations are relativized by  $\langle 1 \rangle$  the potential to further improve the EM, and  $\langle 2 \rangle$  by arguing that on a straight highway most critical traffic situations involve only few directly involved participants.

*Runtime volatility:* Another limitation is the high sensitivity of nuXmv to changes in the checked transitions systems, cf. Sec. 4.2. Small differences can result in a factor of 2 – 3 in runtime. It needs to be considered for each individual case if this is acceptable. A related issue, which can be problematic in an industrial context, is not knowing the remaining time of a run (“will it finish in a minute or run for another two weeks?”). For bounded MC, this question intensifies to whether the run will finish at all, which it never can if the specification is fulfilled. In productive use, long-enduring runs probably need to be aborted after some time. There is currently no general solution for this problem.

*Force to fix bugs right away:* The characteristic of nuXmv, to always produce essentially the same CEX as long as an issue is not fixed, is somewhat problematic for practical application. In theory, a discovered issue is supposed to be fixed immediately, before going on discovering and fixing others. In practice, however, it may well happen that an issue is not easily fixable in a solid and process-abiding way while development still must go on in other directions. An “ignore” option is desirable which lets users “skip” a CEX and trigger the generation of “profoundly” new ones. In fact, we were able to present two differing CEXs with the same version of the Alliance BP in Sec. 4.1 only due to the fact that in one case lane changes were prohibited for ego. These sort of tricks can help to work around this issue.

*Range of applicable BP types:* Technically, we only assume that the BP is written in an imperative language, i. e., C++ for now. Our approach is not limited to deterministic BPs; the presented EM is already non-deterministic, e. g., in modeling the behavior of other cars. However, the presented toolchain is not directly applicable to most AI-based approaches. This is not a structural limita-

tion (a C++ implementation of a deep neural network could, in principle, be fed into our toolchain), but seems infeasible, at today’s state of research, due to well-known issues regarding runtime and numerical instability of these approaches in combination with non-probabilistic MC. An extension to probabilistic MC or probability-based models like POMDPs is conceivable, but has so far not been part of our investigations.

**Lessons Learned.** *A hybrid solution is required:* As pointed out before and in Sec. 5, out-of-the-box solutions (e. g., C model checkers, as well as a pure combination of Kratos2 + nuXmv) can deliver some aspects of the presented toolchain. However, important features like the closed-loop integration of an EM, and the CEX explainability functionality need to be customized for the problem at hand. Particularly, this made it necessary to use a specific C++ parser for the creation of the interface between EM and BP. In future, such a functionality could become a native feature of a model checker.

*“Detour” over Kratos2 and SMV is beneficial:* As shown in Fig. 2 (page 7), the BP code is first translated from C++ into K2, and from there into SMV, where it is connected to an EM in SMV. Here, two “shortcuts” are thinkable,  $\langle 1 \rangle$  the SMV representation of the BP could be generated directly from C++, and  $\langle 2 \rangle$  the explicit representation in SMV could be overall omitted by rather implementing the EM in C++, as well. However, the presented path makes full use of the imperative MC functionality provided by Kratos2 and the rich syntax of SMV. Both shortcuts have been tried out in the beginning and abandoned later, since the current setup outperformed them by far.

*Explainability can be simple:* We experienced our method of extracting traffic scenes from CEXs and further processing them towards visualization and testing as highly effective for  $\langle 1 \rangle$  quickly explaining bugs to both BP developers and, during early phases, EM designers,  $\langle 2 \rangle$  for re-simulation of the results with the actual full software stack to confirm the MC results, and  $\langle 3 \rangle$  for further processing the scenes, for example, for test case generation. Generally, there are a number of further explainability methods to explore, such as translation to natural language [9, 31–33], or, specific to this use case, a further investigation of which code lines of the BP lead to a violation.

*General scalability:* The approach scaled well across the different versions of the Alliance BP, as well as the mock BP (i. e., runtime differences were insignificant). However, none of the BPs’ logic so far contained loops or recursion, which might significantly increase complexity.

*Granularity of abstraction is an open question:* All presented results were produced with a time scale rasterized to one iteration per second in the EM. This shows that this granularity of abstraction can produce useful findings. Going towards an argumentation of “error-freeness”, it needs to be more profoundly inspected what granularity is actually required for which types of statements.

*Every-day development is not impaired:* In practice, for typical “every-day” changes to the code adapting the tags was simple enough to be correctly done

by non-MC-experienced developers. More complex changes were done by an MC expert, or split into work packages. The additional effort was overall tolerable.

## 5 Related Work

In this section, we discuss related work on formal methods and how our work complements them. In general, formal methods aim to prove safety properties of algorithms theoretically. Redfield et al. give a good overview of the challenges within this field [50]. Formal methods include temporal logic encodings [63], monitoring [45, 47], theorem proving [46, 52] (see also overview in [48]) and MC [5]. In the following, we will focus on approaches that could be used for the safety assessment of AV. Notably, only few works actually incorporate such methods into industrial production or for the verification of (parts of) products [16, 18, 29]. A more extensive body of research focuses on theoretically adopting formal methods to the problem landscape given in the automotive industry, cf. overview in [59, 64].

A typical practical issue, preventing broader adoption, is the interface between the problem to solve and the theoretical tooling. The input languages for MC are often quite low-level and lacking a lot of features of modern programming languages like object-oriented features, dynamic data structures, etc. [6, 8, 15, 27, 42]. Therefore, many approaches embed MC into modeling approaches based on domain specific languages and translate from such higher-level representations to the model checkers [14, 22, 26, 56]. In our setting, such approaches cannot be applied as the BP under verification is only available in source code. Other industrial MC applications involve a manual translation step of (part of) the code under verification into model checker language [20, 35]. However, during the development process in a fast-changing environment this is not feasible. Especially in early development stages, not only the code under analysis, but also interfaces and data structures change rapidly.

For handling cases where the system under verification is only available as source code, several MC approaches taking source code as an input have been developed. Existing MC approaches in this category mostly focus on (subsets) of C code [12, 55, 60]. Similar to our strategy, most of these approaches translate the model of the code into a suitable mathematical input language for the model checkers [28, 36]. Checking C++ code requires more sophisticated approaches, since object-orientation and other specific C++ concepts introduce additional layers of complexity. One MC method checking C++ code is DIVINE, which also includes, e. g., exceptions [54]. It could be used as an alternative backend for our approach. The Bogor framework [53] provides means for creating software model checkers for object-oriented languages, but is only available for Java.

Verifying the decisions of the BP in different traffic situations requires to represent these in the EM. While we manually implemented the EM in our approach, there exist approaches using ontologies for representing features of an EM [19]. They represent abstract scenes of driving scenarios and use them



in logic-based reasoning systems. Such approaches could be combined with our approach in future works for automatically configuring the EM.

Another approach is to directly include all safety constraints into the planning itself [25]. A popular example is to use reinforcement learning and integrate safety constraints into the learned policy [39, 57]. However, reinforcement learning approaches suffer from a strong dependency on the specific environment the agents interact with. Thus, one can never ensure safety in all possible corner cases.

## 6 Conclusion

In this paper we described the application of automated verification to improve the development of an actual industrial *behavior planner (BP)*. Complementing the regular validation process based on simulation and test drives, we developed a mechanism to automatically extract from C++ code the model of the underlying BP logic. This model can be integrated with a model of the environment (features of the road and the other vehicles), in a closed-loop manner. This allows to deal in a seamless way with multiple versions of the BP, as they occur during development, and to exhaustively analyze a huge variety of scenarios. In case of violations, the model checker is able to produce traces that can be re-executed in simulators of the original system to guide the search for errors. The approach was exemplarily deployed in series development, and successfully detected multiple relevant issues of intermediate versions of the BP at development time.

There are several directions for future activity. First, we will broaden the scope of the environment modeling to more general scenarios. Second, we will investigate the gray area between the exhaustive exploration of a set of scenarios and a general guarantee of correctness in the real world. Finally, we aim at the application of the methodology to other software components. In fact, it is often the case that the development and the validation teams proceed in parallel. In this respect, the context of application of automated model extraction described in this paper can be considered paradigmatic.

**Acknowledgements.** A. Cimatti, A. Griggio and S. Tonetta acknowledge the support of the project “AI@TN” funded by the Autonomous Province of Trento and of the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU.

## References

1. Amid, G.: ASAM OpenSCENARIO V2.0.0. Tech. rep., Association for Standardization of Automation and Measuring Systems (2022)
2. Aptiv, Audi, Baid, BMW, Continental, Daimler, Fca, Here, Infineon, Intel, Volkswagen: Safety first for automated driving. Tech. rep. (2019), <https://www.aptiv.com/docs/default-source/white-papers/safety-first-for-automated-driving-aptiv-white-paper.pdf>, accessed: 25.09.2023

3. Artuñedo, A., Godoy, J., Villagra, J.: A decision-making architecture for automated driving without detailed prior maps. In: 2019 IEEE Intelligent Vehicles Symposium (IV). pp. 1645–1652. Paris, France (2019)
4. Audi AG, Audi Kommunikation: Audi SocAlty Study (2022), [https://www.audi.com/content/dam/gbp2/company/research/audi-beyond/2021/AUDI\\_SocAlTy\\_Study\\_dgtl\\_1201\\_English\\_small.pdf](https://www.audi.com/content/dam/gbp2/company/research/audi-beyond/2021/AUDI_SocAlTy_Study_dgtl_1201_English_small.pdf), accessed: 25.09.2023
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge, MA, USA (2008)
6. Behrmann, G., David, A., Larsen, K.G., Petterson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems. pp. 125–126. QEST 2006, IEEE Computer Society, Los Alamitos, CA, USA (Sep 2006). <https://doi.org/10.1109/QEST.2006.59>
7. Brooks, R.A.: A robust layered control system for a mobile robot. IEEE Journal on Robotics and Automation **2**(1), 14–23 (1986)
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: Computer Aided Verification. CAV 2014 (2014)
9. Cherukuri, H., Ferrari, A., Spoletini, P.: Towards Explainable Formal Methods: From LTL to Natural Language with Neural Machine Translation. In: Gervasi, V., Vogelsang, A. (eds.) Requirements Engineering: Foundation for Software Quality. pp. 79–86. Springer International Publishing, Cham (2022)
10. Cimatti, A., Griggio, A., Mover, S., Roveri, M., Tonetta, S.: Verification modulo theories. Formal Methods in System Design (2023). <https://doi.org/10.1007/s10703-023-00434-x>
11. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. Formal Methods in System Design **49**(3), 190–218 (2016)
12. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2004. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer, Berlin, Heidelberg (2004)
13. Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., Mover, S.: Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. CAV 2016. Lecture Notes in Computer Science, vol. 9779, pp. 271–291. Springer International Publishing, Cham (2016). [https://doi.org/10.1007/978-3-319-41528-4\\_15](https://doi.org/10.1007/978-3-319-41528-4_15)
14. Daw, Z., Cleaveland, R., Vetter, M.: Integrating model checking and uml based model-driven development for embedded systems. In: Automated Verification of Critical Systems 2013. Electronic Communications of the EASST, vol. 66 (2013). <https://doi.org/10.14279/tuj.eceasst.66.888>
15. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kunčák, V. (eds.) Computer Aided Verification. pp. 592–600. Springer International Publishing, Cham (2017)
16. Eberhart, C., Dubut, J., Haydon, J., Hasuo, I.: Formal verification of safety architectures for automated driving. In: 2023 IEEE Intelligent Vehicles Symposium (IV). pp. 1–8 (2023). <https://doi.org/10.1109/IV55152.2023.10186763>
17. Enderton, H.B.: "A Mathematical Introduction to Logic". Academic Press, Boston, MA, USA, 2. edn. (2001)
18. Farrell, M., Bradbury, M., Fisher, M., Dennis, L.A., Dixon, C., Yuan, H., Maple, C.: Using threat analysis techniques to guide formal verification: A case study

- of cooperative awareness messages. In: Ölveczky, P.C., Salaün, G. (eds.) *Software Engineering and Formal Methods*. pp. 471–490. Springer International Publishing, Cham (2019)
19. Fuchs, S., Rass, S., Lamprecht, B., Kyamakya, K.: A Model for Ontology-Based Scene Description for Context-Aware Driver Assistance Systems. In: 1st International ICST Conference on Ambient Media and Systems. Phoenix, AZ, USA (2010). <https://doi.org/10.4108/ICST.AMBISYS2008.2869>
  20. Gardner, R.W., Genin, D., McDowell, R., Rouff, C., Saksena, A., Schmidt, A.: Probabilistic model checking of the next-generation airborne collision avoidance system. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). pp. 1–10 (2016). <https://doi.org/10.1109/DASC.2016.7777963>
  21. Geisslinger, M., Poszler, F., Betz, J., Lütge, C., Lienkamp, M.: Autonomous Driving Ethics: from Trolley Problem to Ethics of Risk. *Philosophy & Technology* **34**(4), 1033–1055 (2021)
  22. Gerking, C., Dziwok, S., Heinzemann, C., Schäfer, W.: Domain-specific model checking for cyber-physical systems. In: 12th Workshop on Model-Driven Engineering, Verification and Validation. pp. 18–27. MoDeVva 2015, CEUR-WS.org Vol-1514, Ottawa (Sep 2015)
  23. Griggio, A., Jonáš, M.: Kratos2: an SMT-Based Model Checker for Imperative Programs. In: Enea, C., Lal, A. (eds.) *Computer Aided Verification*. pp. 423–436. Springer Nature Switzerland, Cham (2023)
  24. Griggio, A., Roveri, M.: Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits Systems* **35**(6), 1026–1039 (2016). <https://doi.org/10.1109/TCAD.2015.2481869>
  25. Halder, P., Althoff, M.: Minimum-Violation Velocity Planning with Temporal Logic Constraints. In: 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC). p. 2520–2527. IEEE Press, Macau, China (2022). <https://doi.org/10.1109/ITSC55140.2022.9922114>
  26. Heinzemann, C., Lange, R.: vTSL – a formally verifiable dsl for specifying robot tasks. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 8308–8314. IROS'18, IEEE Computer Society, Madrid, Spain (2018). <https://doi.org/10.1109/IROS.2018.8593559>
  27. Holzmann, G.J.: The model checker spin. *Software Engineering, IEEE Transactions on* **23**(5), 279–295 (may 1997). <https://doi.org/10.1109/32.588521>
  28. Holzmann, G.J., H. Smith, M.: Software model checking: extracting verification models from source code†. *Software Testing, Verification and Reliability* **11**(2), 65–79 (2001). <https://doi.org/10.1002/stvr.228>
  29. Ishigooka, T., Saissi, H., Piper, T., Winter, S., Suri, N.: Practical use of formal verification for safety critical cyber-physical systems: A case study. In: 2014 IEEE International Conference on Cyber-Physical Systems, Networks, and Applications. pp. 7–12 (2014). <https://doi.org/10.1109/CPSNA.2014.20>
  30. ISO/TC 22/SC 32 Electrical and electronic components and general system aspects: ISO 21448:2022 Road vehicles – Safety of the intended functionality (2022), <https://www.iso.org/standard/77490.html>, accessed: 25.09.2023
  31. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A user-study protocol for evaluation of formal verification results and their explanation. *arXiv abs/2108.06376* (2021)
  32. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A systematic literature review on counterexample explanation. *Information and Software Technology* **145**, 1–20 (2022). <https://doi.org/10.1016/j.infsof.2021.106800>

33. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A user study for evaluation of formal verification results and their explanation at bosch. *Empirical Software Engineering* **28**(5) (2023)
34. Kalra, N., Paddock, S.M.: Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice* **94**, 182–193 (2016)
35. Keating, D., McInnes, A., Hayes, M.: An industrial application of model checking to a vessel control system. In: 2011 Sixth IEEE International Symposium on Electronic Design, Test and Application. pp. 83–88 (2011). <https://doi.org/10.1109/DELTA.2011.24>
36. Keller, C.W., Saha, D., Basu, S., Smolka, S.A.: FocusCheck: A Tool for Model Checking and Debugging Sequential C Programs. In: Halbwachs, N., Zuck, L.D. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 563–569. Springer, Berlin, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-31980-1\\_39](https://doi.org/10.1007/978-3-540-31980-1_39)
37. Kerner, B.S.: Physics of automated driving in framework of three-phase traffic theory. *Physical Review E* **97**(4) (2018). <https://doi.org/10.1103/PhysRevE.97.042303>
38. Kortenkamp, D., Simmons, R.: *Robotic Systems Architectures and Programming*. In: Siciliano, B., Khatib, O. (eds.) *Springer Handbook of Robotics*. pp. 187–206. Springer, Berlin, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-30301-5\\_9](https://doi.org/10.1007/978-3-540-30301-5_9)
39. Krasowski, H., Zhang, Y., Althoff, M.: Safe Reinforcement Learning for Urban Driving using Invariably Safe Braking Sets. In: 2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC). pp. 2407–2414. Macau, China (2022)
40. Kriebitz, A., Max, R., Lütge, C.: The German Act on Autonomous Driving: Why Ethics Still Matters. *Philosophy & Technology* **35**(2), 29 (2022). <https://doi.org/10.1007/s13347-022-00526-2>
41. Krämer, S., Stiller, C., Bouzouraa, M.E.: LiDAR-Based Object Tracking and Shape Estimation Using Polylines and Free-Space Information. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 4515–4522. Madrid, Spanien (2018). <https://doi.org/10.1109/IROS.2018.8593385>
42. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. LNCS, vol. 6806, pp. 585–591. Springer (2011)
43. Leino, K., M., R.: Program Proving Using Intermediate Verification Languages (IVLs) like Boogie and Why3. In: *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*. pp. 25–26. Association for Computing Machinery (2012). <https://doi.org/10.1145/2402676.2402689>
44. Majzik, I., Semeráth, O., Hajdu, C., Marussy, K., Szatmári, Z., Micskei, Z., Vörös, A., Babikian, A.A., Varró, D.: Towards System-Level Testing with Coverage Guarantees for Autonomous Vehicles. In: Kessentini, M., Yue, T., Pretschner, A., Voss, S., Burgueño, L. (eds.) *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019*. pp. 89–94. IEEE, Munich, Germany (2019). <https://doi.org/10.1109/MODELS.2019.00-12>
45. Mehdipour, N., Althoff, M., Tebbens, R.D., Belta, C.: Formal methods to comply with rules of the road in autonomous driving: State of the art and grand challenges. *Automatica* **152** (2023). <https://doi.org/10.1016/j.automatica.2022.110692>

46. Mitsch, S., Ghorbal, K., Vogelbacher, D., Platzer, A.: Formal verification of obstacle avoidance and navigation of ground robots. *The International Journal of Robotics Research* **36**(12), 1312–1340 (2017). <https://doi.org/10.1177/0278364917733549>
47. Mitsch, S., Platzer, A.: ModelPlex: verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design* **49**, 33–74 (2016). <https://doi.org/10.1007/s10703-016-0241-z>
48. Nawaz, M.S., Malik, M., Li, Y., Sun, M., Lali, M.I.U.: A survey on theorem provers in formal methods (2019)
49. Nees, M.A.: Safer than the average human driver (who is less safe than me)? examining a popular safety benchmark for self-driving cars. *Journal of Safety Research* **69**, 61–68 (2019)
50. Redfield, S.A., Seto, M.L.: Verification challenges for autonomous systems. In: Lawless, W., Mittu, R., Sofge, D., Russell, S. (eds.) *Autonomy and Artificial Intelligence: A Threat or Savior?*, pp. 103–127. Springer International Publishing, Cham (2017). [https://doi.org/10.1007/978-3-319-59719-5\\_5](https://doi.org/10.1007/978-3-319-59719-5_5)
51. Reid, T., Houts, S., Cammarata, R., Mills, G., Agarwal, S., Vora, A., Pandey, G.: Localization requirements for autonomous vehicles. *SAE International Journal of Computer Aided Verification* **2**(3), 173–190 (2019). <https://doi.org/10.4271/12-02-03-0012>
52. Rizaldi, A., Keinholtz, J., Huber, M., Feldle, J., Immler, F., Althoff, M., Hilgendorf, E., Nipkow, T.: Formalising and Monitoring Traffic Rules for Autonomous Vehicles in Isabelle/HOL. In: Polikarpova, N., Schneider, S. (eds.) *Integrated Formal Methods: 13th International Conference, IFM 2017, Turin, Italy*, pp. 50–66. No. 10510 in *Lecture Notes in Computer Science*, Springer International Publishing, Cham (2017). [https://doi.org/10.1007/978-3-319-66845-1\\_4](https://doi.org/10.1007/978-3-319-66845-1_4)
53. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: A flexible framework for creating software model checkers. In: *Proceedings of Testing: Academic and Industrial Conference - Practice And Research Techniques*. pp. 3 –22. TAIC PART 2006 (aug 2006). <https://doi.org/10.1109/taic-part.2006.5>
54. Ročkai, P., Barnat, J., Brim, L.: Model checking C++ programs with exceptions. *Science of Computer Programming* **128**, 68–85 (2016). <https://doi.org/10.1016/j.scico.2016.05.007>
55. Schlich, B., Kowalewski, S.: Model checking c source code for embedded systems. *International Journal on Software Tools for Technology Transfer* **11**(3), 187–202 (2009). <https://doi.org/10.1007/s10009-009-0106-5>
56. Schmidt, Á., Varró, D.: Checkuml: A tool for model checking visual modeling languages. In: Stevens, P., Whittle, J., Booch, G. (eds.) *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, *Lecture Notes in Computer Science*, vol. 2863, pp. 92–95. Springer Berlin Heidelberg (Oct 2003). [https://doi.org/10.1007/978-3-540-45221-8\\_8](https://doi.org/10.1007/978-3-540-45221-8_8)
57. Schmidt, L.M., Kontes, G., Plinge, A., Mutschler, C.: Can You Trust Your Autonomous Car? Interpretable and Verifiably Safe Reinforcement Learning. In: *2021 IEEE Intelligent Vehicles Symposium (IV)*. pp. 171–178. Nagoya, Japan (2021). <https://doi.org/10.1109/IV48863.2021.9575328>
58. Schreurs, M., Steuer, S.: Autonomous Driving - Political, Legal, Social, and Sustainability Dimensions. *Autonomes Fahren: Technische, rechtliche und gesellschaftliche Aspekte* pp. 151–173 (2015)
59. Selvaraj, Y., Ahrendt, W., Fabian, M.: Verification of decision making software in an autonomous vehicle: An industrial case study. In: Larsen, K.G., Willemse, W.P.M. (eds.) *Formal Methods in System Design*. pp. 1–15. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-52221-2\\_1](https://doi.org/10.1007/978-3-030-52221-2_1)

- T. (eds.) *Formal Methods for Industrial Critical Systems*. pp. 143–159. Springer International Publishing, Cham (2019)
60. Shankar, S., Pajela, G.: A tool integrating model checking into a c verification toolset. In: Bošnački, D., Wijs, A. (eds.) *Model Checking Software, Lecture Notes in Computer Science*, vol. 9641, pp. 214–224. Springer International Publishing (2016). [https://doi.org/10.1007/978-3-319-32582-8\\_15](https://doi.org/10.1007/978-3-319-32582-8_15)
  61. Shariff, A., Bonnefon, J.F., Rahwan, I.: How safe is safe enough? Psychological mechanisms underlying extreme safety demands for self-driving cars. *Transportation Research Part C: Emerging Technologies* **126**, 1–12 (2021). <https://doi.org/10.1016/j.trc.2021.103069>
  62. Wachenfeld, W., Winner, H.: *The Release of Autonomous Vehicles*, pp. 425–450. Springer, Berlin, Heidelberg (2016). [https://doi.org/10.1007/978-3-662-48847-8\\_21](https://doi.org/10.1007/978-3-662-48847-8_21)
  63. Zhao, T., Yurtsever, E., Paulson, J.A., Rizzoni, G.: Formal Certification Methods for Automated Vehicle Safety Assessment. *IEEE Transactions on Intelligent Vehicles* **8**(1), 232–249 (2022). <https://doi.org/10.1109/TIV.2022.3170517>
  64. Zhao, T., Yurtsever, E., Paulson, J.A., Rizzoni, G.: Formal certification methods for automated vehicle safety assessment. *IEEE Transactions on Intelligent Vehicles* **8**(1), 232–249 (2023). <https://doi.org/10.1109/TIV.2022.3170517>