A Systematic Framework to Identify Violations of Scenario-dependent Driving Rules in Autonomous Vehicle Software

QINGZHAO ZHANG, University of Michigan, USA
DAVID KE HONG, University of Michigan, USA
ZE ZHANG, University of Michigan, USA
QI ALFRED CHEN, University of California, Irvine, USA
SCOTT MAHLKE, University of Michigan, USA
Z. MORLEY MAO, University of Michigan, USA

Safety compliance is paramount to the safe deployment of autonomous vehicle (AV) technologies in real-world transportation systems. As AVs will share road infrastructures with human drivers and pedestrians, it is an important requirement for AVs to obey standard driving rules. Existing AV software testing methods, including simulation and road testing, only check fundamental safety rules such as collision avoidance and safety distance. Scenario-dependent driving rules, including crosswalk and intersection rules, are more complicated because the expected driving behavior heavily depends on the surrounding circumstances. However, a testing framework is missing for checking scenario-dependent driving rules on various AV software.

In this paper, we design and implement a systematic framework *AVChecker* for identifying violations of scenario-dependent driving rules in AV software using formal methods. *AVChecker* represents both the code logic of AV software and driving rules in proposed formal specifications and leverages satisfiability modulo theory (SMT) solvers to identify driving rule violations. To improve the automation of systematic rule-based checking, *AVChecker* provides a powerful user interface for writing driving rule specifications and applies static code analysis to extract rule-related code logic from the AV software codebase. Evaluations on two open-source AV software platforms, *Baidu Apollo* and *Autoware*, uncover 19 true violations out of 28 real-world driving rules covering crosswalks, traffic lights, stop signs, and intersections. Seven of the violations can lead to severe risks of a collision with pedestrians or blocking traffic.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Computer systems organization** → Embedded and cyber-physical systems.

Additional Key Words and Phrases: Autonomous vehicle; Software system; Formal methods

ACM Reference Format:
Qingzhao Zhang, David Ke Hong, Ze Zhang, Qi Alfred Chen, Scott Mahlke, and Z. Morley Mao. 2021. A Systematic Framework to Identify Violations of Scenario-dependent Driving Rules in Autonomous Vehicle Software. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 2, Article 15 (June 2021), 25 pages. https://doi.org/10.1145/3460082

Authors' addresses: Qingzhao Zhang, University of Michigan, USA, qzzhang@umich.edu; David Ke Hong, University of Michigan, USA, kehong@umich.edu; Ze Zhang, University of Michigan, USA, zezhang@umich.edu; Qi Alfred Chen, University of California, Irvine, USA, alfchen@uci.edu; Scott Mahlke, University of Michigan, USA, mahlke@umich.edu; Z. Morley Mao, University of Michigan, USA, zmao@umich.edu.

2476-1249/2021/6-ART15 $15.00
https://doi.org/10.1145/3460082

1 INTRODUCTION

Emerging autonomous vehicles (AVs) hold great promise in transforming today's transportation systems and mobility services while driving safety is the most important design requirement before their real-world deployment. The AV is a complicated cyber-physical system where a central AV software interacts with digital sensors and physical devices to automatically drive the vehicle. Since the AV software controls the AV's driving behavior, the testing of AV software is of great importance for ensuring driving safety. Unanimously agreed by AV software vendors and government authorities, the AV developers should validate the AV software's compliance with essential safety standards (e.g., traffic laws, voluntary safety standards from NTHSA [1], Responsibility-Sensitive Safety [54]).

Existing AV software safety testing approaches (e.g., simulation, track, or road testing) [2, 4, 13] mainly use the black-box dynamic testing. They collect driving traces from the real-world test drives or simulations and then verify their safety properties. However, these methods only target a limited set of driving rules related to basic safety requirements such as collision avoidance and safe distance on urban roads or highways [29]. Such fundamental rules should always hold during the drive and only consider relation among vehicles. More complicated driving rules including on-road crosswalks, four-way stop signs, and intersections with traffic lights are ignored by general AV testing techniques. We name the above rules **scenario-dependent driving rules** because the expected behavior of an AV heavily depends on its location as well as surrounding circumstance. For instance, to decide to drive into an intersection, the AV needs to consider the whole circumstance including its lane and direction, the traffic light, pedestrians on the crosswalk ahead, and the behaviors of all other vehicles. To check these complicated rules, road/track testing or simulation requires test cases with high quantity and diversity to enumerate possible map layouts and different behaviors of all pedestrians and vehicles, which is difficult due to regulation barriers, high cost/risk of real-world testing, and the complexity of the driving scenarios even in simulators. In addition, a general framework is still missing for testing driving rule compliance on AV software. Existing AV testing methods are mostly designed for one specific AV software or one specific driving rule. For example, different companies launch road testing for their own AV systems [2, 4, 13] and *AV-FUZZER* [41] (simulation-based) only checks collision avoidance rules. A general framework is in need to support various AV software and a wide range of driving rules simultaneously.

To address this limitation, we propose *AVChecker*, a framework for AV developers to systematically find violations of complex scenario-dependent driving rules in AV software using formal methods. Instead of enumerating test cases in dynamic testing, *AVChecker* proposes a formal representation to model complex driving scenarios considering both the map layouts and behaviors of moving objects. Then we can apply formal verification techniques to identify driving rule violations by analyzing the formal representations. To start with, *AVChecker* takes driving rule specifications and source code of AV motion planning (i.e., the module implementing driving rules) as inputs. The driving rule specifications are formatted in the proposed formal representation which is encoded in Satisfiability Modulo Theory (SMT), and provided by issuers of traffic rules. Meanwhile, a pipeline of static code analysis extracts AV's driving behaviors from the AV source code with the assistance of code annotations provided by AV developers. The driving behaviors are encoded into a finite state machine based specification, called behavior specification. Next, we apply formal methods (i.e., SMT theorem proving) to reveal the violation between the driving rule specification and AV's behavior specification. The violation means that at one specific moment, the AV's driving behavior is different from the expected behavior defined in the driving rule. As long as an violation exists between the two specifications, the SMT solver can generate a counterexample which helps in reproducing the identified flaw. *AVChecker* will post-process the generated violation case to construct

a test case of simulation, which can help to rule out false positives or debug the software. Though *AVChecker* does not handle perception or machinery failures and cannot guarantee end-to-end driving safety, it is the best effort for validating driving rules implementation which is one of the most critical modules of AV driving.

To identify rule violations through formal methods, our work addresses two key challenges. First, performing a direct comparison between driving rules expressed in natural language and AV software code is impractical because of the semantic gap. To bridge this gap, we propose a domain-specific formal representation modeling traffic scenes. Both rule specifications and AV's behavior specifications are based on the same formal representation so that they can be analyzed in the same domain. In addition, *AVChecker* provides user interfaces with various levels of abstractions to minimize manual effort for generating specifications. Second, characterizing continuous driving scenarios requires modeling real-world physical dynamics in continuous time and space domains, which is challenging to realize using the theory of SMT because of the complexity. Thus, the model of driving scenarios should abstract the real world as much as possible but maintain necessary expressiveness for rule compliance checking. We notice that the driving rule handling in AV software is executed in cycles and the driving behavior is decided in each cycle according to the circumstance of the current moment. *AVChecker* abstracts the continuous driving behaviors by splitting the time sequence into moments and checks AV's driving decisions on each moment. To achieve the abstraction, *AVChecker* constructs one moment of the driving scenario using SMT symbolic variables, called symbolic **traffic snapshot**. *AVChecker* detects violations when AV's behaviors break the driving rule at any possible traffic snapshot in this scenario without the need of considering over-complicated details of physical dynamics during a time sequence, making our SMT-based approach scalable on complicated driving scenarios.

We prototype *AVChecker* using LLVM [38] and *Z3* SMT solver [25], and evaluate it on two open-source AV software platforms, *Baidu Apollo* [3] and *Autoware* [6] which have 108K LOC and 42K LOC in their planning modules respectively. Our prototype is able to detect 19 true violations with driving rules for crosswalks, traffic lights, stop signs, and intersection scenes that are defined in driving manuals from the Department of Motor Vehicles (DMV) or common safe driving practices on both platforms. The simulation-based validation further confirms that the violations are all true positives and 7 of them may lead to severe safety consequences, including the risk of hitting a sprinting pedestrian on a crosswalk or blocking traffic at intersections. The violations are caused by the incomplete implementation, ignorance of corner cases, or bugs. The specification API of *AVChecker* is demonstrated to be effective for reducing AV developer's specification efforts, requiring less than 10 lines of code for specifying different complex scenes for the evaluated driving rules and reducing manual specification efforts by 15x. Moreover, the snapshot abstraction significantly reduces the state space so that *AVChecker* can complete the violation identification within 13 seconds for each targeted rule.

The contributions of this paper are as follows:

1) We propose an AV domain-specific abstraction, which is a formal representation amenable to SMT solving, to bridge the semantic gap between software code and safety specifications. This new abstraction simplifies the representation of infinite-state space in the physical world and addresses the scalability challenge in SMT-based driving rule compliance checking.

2) We design and implement *AVChecker*, to the best of our knowledge, the first general framework for checking scenario-dependent driving rules in AV software in a systematic manner. Using the user interface of *AVChecker*, AV developers can reveal rule violations in the code logic against predefined driving rule specifications with the minimum manual effort.

3) We evaluate *AVChecker* on *Baidu Apollo* [3] and *Autoware* [6] to check rule compliance with 28 DMV's driving rules. *AVChecker* uncovers 13 violations in *Apollo* and 6 violations in *Autoware*, which are all validated by simulation.

2 BACKGROUND & MOTIVATION

In this section, we introduce the background of AV's driving rule enforcement (§2.1), limitations of previous AV testing work (§2.2), and our design goals (§2.3) to address the limitations.

2.1 Driving rule enforcement in AV software

Handling driving rules is one of the critical components of AV software systems. The AV system is composed of a processing pipeline with several key modules to perform self-driving functionalities, including localization, perception, prediction, routing, motion planning, and control. In particular, motion planning aims to generate collision-free motions for moving the vehicle from point A to B by handling on-road traffic safely and legally. It takes the static map reference, moving obstacles, and all road blockages as input. It outputs controller directives including lateral driving bias, the leading vehicle to follow, the aggressiveness of distance keeping, and maximum speed [60].

There exists complex rule-based logic in motion planning. We studied driving rule enforcement in *Baidu Apollo*, an open-source AV software platform that supports most representative driving rules. The typical traffic rule enforcement logic handles common traffic objects, e.g., traffic lights, stop signs, crosswalks, and key driving actions, e.g., stop, overtake. At the run time of motion planning, the rule enforcement is performed for each execution cycle (100ms) using the latest perceived traffic scene.

Violation detection of AV's driving rule enforcement is indispensable. As suggested by the *Baidu Apollo* developers, autonomous driving should follow traffic regulations at all times [28]. In particular, we survey driving rules in driving manuals published by Department of Motor Vehicles (DMV) of US states [5, 8, 9, 11, 12] and check these rules on *Apollo* and *Autoware*.

Table 1. Comparison of AV testing or validation approaches.

Core technique		Unit	Fuzz	Simulation			SMC		Model		SMT
Tool		[3]	[7]	[41]	[47]	[15]	[48]	[18]	[51]	[49]	*AVChecker*
Objective of testing	Low-level bugs	√	√	×	×	×	×	×	×	×	×
	Fundamental safety rules	×	×	√	√	√	√	√	√	√	×
	Scenario-dependent rules	×	×	×	×	×	×	×	×	×	√
Bug-free assurance	Code-level	×	×	×	×	×	×	×	×	×	×
	Spec-level	×	×	×	×	×	×	×	√	√	√
Generality	AV systems	×	×	√	√	√	√	√	√	√	√
	Driving rules	-	-	×	×	×	×	×	×	×	√

The first row: Unit test [3], Fuzzing [7], Simulation [15, 41, 47], Statistical Model Checking [18, 48], model-based formal checkers [49, 51], SMT (*AVChecker*).

2.2 Limitations of previous AV testing

Table 1 lists existing software testing methods on AVs, including unit tests [3], fuzz [7], simulation case generation [15, 41, 47], Statistical Model Checking [18, 48] and model-based formal checkers [49, 51]. Track and road testing are not listed in the table.

Existing methods have the following limitations on identifying driving rule violations. First, none of the existing tools checks complicated scenario-dependent driving rules on AV software code. Conventional software testing such as unit tests and fuzzing focuses on low-level bugs (e.g., program crashes) in the source code. Existing simulation-based AV testing focuses on fundamental safety violations such as collision avoidance. Existing formal methods for AV testing verify whether a driving rule can avoid collisions in the real world, which does not consider code implementation. *AVChecker* is the first study to check general complicated driving rules on AV software. Second, neither unit/fuzzing tests or simulation can ensure the testing completeness since they cannot

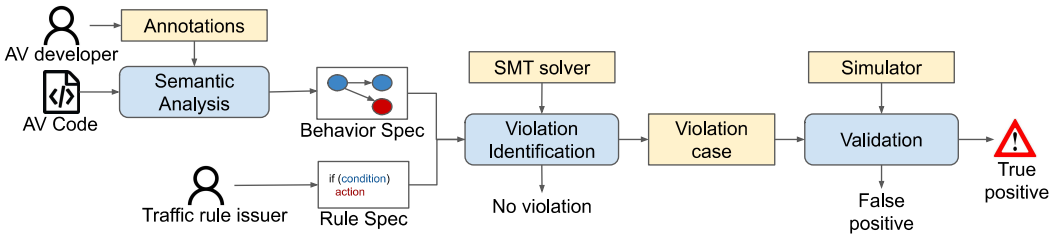


Fig. 1. Overall workflow of *AVChecker* framework.

enumerate sufficient test cases to cover all corner cases. By applying formal methods, if *AVChecker* finds no violations and all specifications are assumed to be sound, the specifications are violation-free. Although *AVChecker* cannot provide an end-to-end safety guarantee because of inaccuracies in specification generation, it is the best effort service to achieve a low false-negative rate in violation detection. Third, existing tools focus on specific rules, resulting in poor generality for tool design. *AVChecker*, as a framework, supports various AV software and driving rules simultaneously.

2.3 Design goals

AVChecker should achieve the following design goals to address the limitations of previous work.

First, *AVChecker* should be able to identify real violations of scenario-dependent rules. The violation identification should be correct, i.e., found violations are validated to be realistic. After generating violation cases using formal models, *AVChecker* applies simulators (§3.6) to confirm whether the violation can happen in the real world, under the assumption that the simulator is correct. In addition, the violation identification should be complete at the specification level. In other words, there is no missed violation among the checked specifications assuming all specifications correctly represent AV's behavior and driving rules, as discussed in §3.5. Note that the generation of specifications involves code analysis and manual work so that inaccuracies exist (§6). *AVChecker* requires a manual validation of the generated specifications to ensure their correctness.

Second, *AVChecker* should be general for various AV software systems and driving rules. Especially, *AVChecker* should support AV software which makes driving decisions in cycles, which is the common design of existing AV systems [3, 6], and general scenario-dependent driving rules including crosswalks, intersections, etc. In addition, *AVChecker* as a framework promotes generality by isolating the interfaces for AV software and driving rules. Changing either AV software or driving rules does not affect the other side.

Third, *AVChecker* should be easy to use. Despite the complexity of the low-level source code and rule details, developers should not be required to handle a large amount of tedious manual work. Though we require manual annotations on software code from developers, we design *AVChecker*'s code analysis and user interface to minimize the manual effort.

3 SYSTEM DESIGN

AVChecker aims to validate whether the driving behaviors implemented in AV software meet the requirements of real-world scenario-dependent driving rules. The core of *AVChecker* is an *AV traffic model* (§3.2) that formally represents traffic scene snapshots using SMT symbolic variables. One symbolic snapshot refers to one moment of a traffic scene. It presents the status of traffic objects, including moving objects such as vehicles and pedestrians as well as stationary landmarks such as crosswalks and intersections. For example, the position of a vehicle is represented by two symbolic variables: a two-dimensional coordinate point. All specifications in *AVChecker* are encoded into SMT syntax using the AV traffic model (§3.3).

Given the AV traffic model, *AVChecker* achieves the systematic violation identification through three modules, as illustrated in Figure 1. *Semantic analysis* (§3.4) analyzes the input AV software and automatically extracts the behavior specification, which represents the code logic of handling driving rules in AV software. The extraction process is achieved by a pipeline of static code analysis, which identifies rule-related code, and encodes the code logic into the behavior specification. In addition, annotations from developers are required to bridge source code and AV traffic model but *AVChecker* provides powerful APIs for writing annotations to minimize this manual effort. *Violation identification* (§3.5) then reveals the inconsistency between the behavior specification and the rule specification. The rule specification is the formal representation of a driving rule, provided by traffic rule issuers. *AVChecker* uses *z3* SMT solver to reason the existence of violations between the rule specification and the behavior specification. If *z3* produces one counterexample that identifies a violation between the two specifications, the counterexamples can be used as test cases for further debugging. *Validation* (§3.6) subsequently runs in an AV simulator to validate the discovered violation and thus rules out false positives. The validation module automatically translates the counterexample produced by the verification module into a test case for the simulator, and the simulator will demonstrate the violation in a realistic setting. In addition, *AVChecker* provides framework support for various customization. Developers can extend the user interfaces in the AV traffic model to further reduce manual effort, customize code analysis to better fit a specific codebase, or change the simulator for validation.

We claim the accuracy and generality of *AVChecker* as follows. If the violation identification module cannot produce violations, it guarantees the non-existence of violations among specifications under the assumption that rule and behavior specifications are correctly formulated. However, the generation of specifications is not guaranteed to be accurate because of the limitations of code analysis as well as the untrusted manual annotations. *AVChecker* needs the inspection and refinement from AV developers to ensure the correctness of specifications. Validation module rules out false positives of violations caused by the gap between the abstracted model and the real world, assuming the correctness of simulators. In terms of generality, we assume the AV software code satisfies the formulation in §3.3.2 and can be compiled by LLVM, which is satisfied by all open-source AV software as far as we know. Also, we assume the driving rules satisfy, or can be transformed to, the "condition and action" structure defined in §3.3.1.

3.1 Example: crosswalk rule in Baidu Apollo

To demonstrate *AVChecker*'s methodology, we check one crosswalk driving rule on *Baidu Apollo* software as one example.

The driving rule says that *stop for pedestrians on your side of an oncoming crosswalk* (rule 2 in Table3). This rule illustrates a common structure of driving rules: the driver should take a specific action when a condition is met. In this case, the action is “stopping at the crosswalk” and the condition is a set of sub-conditions connected by ANDs, including "pedestrian on crosswalk", "crosswalk is in front of AV" and "pedestrian is on the current road".

AV software such as *Baidu Apollo* uses a different way to enforce crosswalk rules. The related code segment is shown in Figure 2. AV's planning module is executed in cycles and maintains a software state indicating the current status of the driving task, as mentioned in § 2.1. In each execution cycle, the software accepts the latest perceived traffic scene, updates the software state, and optionally takes specific driving actions. The update of software states and action-taking is under some conditions, which are determined by the branches and control flows in the program.

Although general driving rules and AV software code semantics have different structures, the action and the condition are the common components. Actions refer to a finite set of possible driving actions during handling driving rules. Conditions are constraints on traffic scenes, which

means the condition can be true or false given one moment of the traffic scene. AV traffic model (§ 3.2) defines both actions and conditions so that we can build formal specifications (§3.3) to represent driving rules as well as AV's behavior. In the following sections, we will demonstrate how *AVChecker* processes this example case to identify rule violations.

3.2 AV traffic model

AV traffic model is a toolkit for representing actions and conditions that appeared in the driving rules and AV software. The representation of actions is straightforward. The model defines a finite set of possible driving actions including "stop_at_crosswalk", "stop_at_stop_sign", "park", etc. For representing conditions, the model first formally representing a traffic scene snapshot using symbolic variables, which contains the status of on-road objects in one moment, including shape, position, velocity, and other properties of objects such as vehicles and crosswalks. Then we uses constraints on the symbolic traffic snapshot to represent the conditions. §3.2.1 will introduce the representation of symbolic traffic snapshots and the construction of snapshot constraints.

3.2.1 Snapshot representation. AV traffic model provides a user interface with four layers including *base variables*, *geometric interface*, *traffic objects* and *high-level interface*, from low-level to high-level abstraction. In addition, the interfaces can be extended to support specialized traffic scenarios.

Base variables are SMT symbolic variables, which together represent a symbolic snapshot. By assigning concrete values to base variables, a symbolic snapshot becomes one concrete one. However, base variables are not comprehensible for users so that we have higher-level interfaces.

Geometric interface. Decision making in AV mainly depends on the geometric information of the surrounding environment in a two-dimensional space. The geometric interface uses AV-centric *S-L coordinate* [28, 54], as the two-dimensional coordination system, due to its simplicity. Given the current lane of the AV, an S-L coordinate point represents the specific position where S denotes the distance along AV's trajectory, and L denotes the lateral distance from the AV. The S-L coordinate is commonly used by robotic systems and can simplify our representation since it is AV-centric: origin as AV's position and S axis as AV's trajectory.

Based on the S-L coordinate system, we define geometric objects: *point*, *line* and *area*. (1) *Point.* A point in the two-dimensional coordinate system is a tuple of two symbolic values representing coordinates. For example, a point $p \equiv (l, s)$ where l and s present L and S coordinate, respectively. (2) *Line.* A line is a function taking a point as input and returning symbolic binary output indicating whether the point is on the line. Formally, the line l is a function $l : P \rightarrow \{0, 1\}$ where P is the set of points. (3) *Area.* Similar to lines, an area a are also represented as a function $a : P \rightarrow \{0, 1\}$.

In addition to geometric objects, we provide geometric APIs to describe geometric relationships. Formally, each geometric API accepts geometric objects as arguments and returns a boolean expression. We implement the following APIs which are necessary for describing traffic scenes. (1) *Point in Line/Area.* Given a point p and a line/Area m, return $m(p)$. (2) *Line/Area cross Line/Area.* Given two lines or areas m and n, return $\exists P, m(p) \wedge n(p)$. (3) *Line/Area in Line/Area.* Given one line/area m and area n, return $\forall p, m(p) \rightarrow n(p)$.

Though the formula of geometric relations contains quantifiers, we apply quantifier elimination [21] in our implementation to improve the scalability of SMT proving.

Traffic objects. Both road geometry and on-road traffic are inputs to the motion planning of common autonomous driving systems [39, 44, 54, 58, 60]. Based on base variables and geometric interface, we are able to define these traffic objects including vehicles, pedestrians, intersections, crosswalks, stop signs, traffic lights, lanes, roads, etc. We adopt an object-oriented approach that the traffic objects have their own properties.

We divide traffic objects into static ones and mobile ones. Static objects are immovable landmarks including traffic lights, stop signs, intersections, crosswalks, lanes, etc. Static objects have properties such as position (a point), size (a point), and boundary (an area). Mobile objects (i.e., vehicles, pedestrians) have all properties of static objects plus some extra properties such as velocity (a point) and trajectory (a line). The traffic objects can have customizable extra properties to extend the expressivity of the model. We list some properties implemented in our prototype as follows. (1) Traffic lights have an integer property "color" representing the signal color. (2) One lane object and one road object are defined to represent AV's lane and road respectively. They both have "left distance" and "right distance" properties representing the distance from AV to the road/lane's boundaries. (3) Intersections have integer properties defining how many roads and lanes are connected by themselves and label all roads or lanes with IDs. (4) Vehicles have an integer property "turn direction" indicating it is going straight or turning left/right, a "lane id" representing the lane it is driving on, and a real number property "waited time" indicating how long it has been stopped.

High-level interface. The high-level interface provides more flexible and powerful APIs to further improve the usability of the AV traffic model. In our implementation, we summarize a few frequently used logic in AV software as high-level APIs. For instance, API *TrajectoryCross* determines whether two mobile objects will collide with each other based on their trajectories and API *Wait* determines whether a vehicle is waiting behind a stop line based on its position and velocity. Note that developers can freely define customized APIs in the high-level interface.

Snapshot constraints. Given the four layers, a symbolic snapshot is constructed by traffic objects at a high level and base variables at a low level. Besides, there is always one constraint restricting the snapshot. In AV traffic model, we define a default constraint restricting the snapshot to comply with real-world settings. For example, all traffic objects have a limited size and velocity, object properties have their own definition domains, the crosswalks or stop signs are always on the roads, and so on. We call the default constraint *realistic constraint*.

The symbolic snapshot can also represent specialized traffic scenes by adding other constraints other than the realistic constraint. For instance, to represent the crosswalk rule condition in § 3.1, we first consider a one-pedestrian scenario and initialize one AV, one road, one crosswalk, and one pedestrian in the AV traffic model. Then the rule's constraint is as follows.

```
And(Approach(ego, crosswalk), In(pedestrian, crosswalk), In(pedestrian, road))
```

In the constraint, `ego` means the AV (a special instance of vehicles), `And` is the built-in connective of SMT formula, `In` is the geometric API *point in area* and `Approach` is a high-level API representing that the crosswalk is ahead of the AV.

In summary, besides a set of actions, the AV traffic model defines a set of traffic objects, a realistic constraint, and a four-layer user interface.

3.2.2 Model expressivity and abstraction. Similar to existing safety models on autonomous driving [49, 51, 54], the AV traffic model is abstracted from the real world and inevitably cannot include all details in real-world scenarios. The AV traffic model adopts abstractions to alleviate the modeling burden while maintaining sufficient expressivity. Since all traffic snapshot constraints are constructed by a set of atomic constraints (e.g., rule condition in §3.1 has three atomic sub-conditions), the AV traffic model has sufficient expressivity if it can represent all the atomic constraints.

After studied collected driving rules and AV software, we classify the driving rule related atomic constraints into two types. First, geometric information, i.e., the relation among objects' position, size, bounding box, velocity, and trajectory. The rule condition in § 3.1 is one example. Since the model defines a complete set of APIs for representing the relations among points, lines, and areas in a two-dimensional space, it can completely represent the geometric information. Second, history information. Some constraints contain the states of traffic objects in the past and we use the

properties of traffic objects to represent the history information as long as the history states can be represented by finite variables. For example, whether the vehicle has been stopped for enough time is one atomic constraint mentioned in stop sign rules. In this case, the model assigns vehicle object a non-negative real number property called "waited time" indicating how long the vehicle has been stopped and adds an constraint of its range (i.e., zero when the vehicle is not waiting) to the realism constraint. In our prototype, we defined a minimum set of object properties to satisfy the requirement of checking our test cases in the experiment (§ 4.2) but developers can freely add more object properties to address other historical information. Due to the extendability of the model, the model has sufficient expressivity on the history information.

While assuring the sufficient expressivity of the model, model abstractions are necessary because of the limitation of SMT-based theorem proving. SMT can efficiently solve linear constraint satisfaction problems but has scalability problems when solving nonlinear problems. As a result, we reduce nonlinearity of the AV traffic model through following abstractions.

First, bounding box representation. Complicated shapes of the objects' boundaries cost many symbolic variables and constraints to represent, which harms the efficiency of the SMT solving. However, driving rules and AV software both focus on relations among bounding boxes rather than their shapes. As a result, the shape of bounding boxes is a redundant detail for checking driving rules. In our model, we set the default shape of areas as a rectangle whose edges are parallel to coordination axes, thereby making area representation light-weight. A rectangle only requires 4 symbolic values (i.e., 2-dimensional position and size) and a simple constraint (i.e., the value of size is positive). The abstracted bounding boxes can still represent all geometric relations (i.e., line/area cross area, point/line/area in area) so that the model expressivity is preserved.

Second, trajectory representation. It is common for AV software to check whether the trajectories of two objects are overlapped. In the real world, the trajectories are usually curves along the lanes. However, it costs too much for a formal model to represent the arbitrary lane layout and curves. To solve the problem, we regard the ray starting from the position with a symbolic direction as the trajectory. The ray representation can represent all geometric relation about trajectories mentioned in driving rules (i.e., trajectory cross an area or another trajectory) except one limitation: unlike curves, two rays can have at most one intersection point. However, even in the complicated intersection scenarios, two vehicles' trajectories can have at most one intersection point so that the limitation does not affect driving rule checking.

3.3 Specification

Based on the AV traffic model, we define rule specifications representing the real-world driving rules and behavior specifications representing code logic implemented in AV software. In the following sections, we notate the set of actions as A and the AV traffic model as M.

3.3.1 Rule specification. Driving rule specification is a formal representation of a real-world driving rule which defines that a specific driving action should be take when the current traffic snapshot satisfies a constraint. The specification of the crosswalk example in § 3.1 is defined as follows:

```
action = stop_at_crosswalk
constraint = And(Approach(ego, crosswalk), In(pedestrian, crosswalk), In(pedestrian,road))
type = 1
```

Formally, the rule specification is defined as a tuple $Spec_r \equiv (M, a, C_r, \mu)$ where M is the AV traffic model, $a \in A$ refers to one action, and $C_r \subset \Sigma$ is a constraint on traffic snapshots, called **rule constraint**. All mentioned objects (i.e., ego, crosswalk, pedestrian, and road) are defined by AV traffic model. Especially, we support two types of rule specifications. If the rule type identifier $\mu = 1$, the vehicle **must** take the action when the rule constraint is met. Otherwise, if $\mu = 0$, the vehicle

must not take the action when the constraint is satisfied. The above example rule specification is a "must" type rule and its constraint is explained in §3.2.1.

3.3.2 Behavior specification. In this section, we present the specifications we used to describe the AV's driving behavior related to driving rules.

Formulation of AV motion planning. As mentioned in §2.1 and §3.1, AV motion planning executes its planning algorithm once per execution cycle. AV motion planning manages a software state, which is a set of state variables recording necessary information about the current traffic scenario. In each execution cycle, the AV planning module accepts the current traffic snapshot (i.e., on-road traffic and map information represented in a three-dimensional coordination) and the current software state as input. Then it generates driving actions and updates the software state according to the above inputs.

Correspondingly, we use a finite state machine scheme to formulate AV driving rule enforcement, which is part of AV motion planning. In the state machine, the inputs are traffic snapshots and the states are various software internal states. The state-transition function refers to the code logic of updating state variables, which accepts two states and outputs the constraint of snapshots when transiting one state to the other. In addition, there are action constraints representing the code logic of making driving decisions, which accepts one state plus one action and outputs the constraint of snapshots when taking the specific action on the state. Formally, given the action space A, the state machine maintains infinite software states S, state transitions δ (i.e., state s_1 is updated to s_2 iff the snapshot constraint $\delta(s_1, s_2)$ is satisfied) and action constraints α (i.e., action a is taken on state s_i iff the snapshot constraint $\alpha(s_i, a)$ is satisfied).

Specification design. The behavior specification adopts the state machine formulation. The specification first lists the state variables for identifying the state space, and then presents state transitions as well as action constraints. An example specification about the crosswalk example (§3.1) is as follows:

```
state_vars = [Bool finished]
states = {S0: finished==False, S1: finished==True}
transitions = {S0->S1: Not(Approach(ego, crosswalk))}
actions = {S0: {"stop_at_crosswalk": And(In(pedestrian, crosswalk), Or(pedestrian.position.L<=4, ...), ...)}}
```

Formally, the behavior specification is a tuple $Spec_c = (M, S, \delta, \alpha)$, where M is the AV traffic model, S is the state space, δ is the **transition constraints** and α is the **action constraints**. Especially, we identify the state space S by extracting state variables from the software. State variables are special in terms of liveness and mutability because they are globally alive across different cycles and get updated in each cycle. We deploy code analysis to automatically identify state variables as mentioned in §3.4.2.

As for the example, there is one state variable *finished* and one transition assigning *finished* from false to true, thus there are two states where *finished* is true or false. The *stop_at_crosswalk* action is only possible to be taken when *finished* is false.

3.4 Semantic analysis

The semantic analysis is a code analysis pipeline for extracting the behavior specification from AV software and is built on LLVM [38], a popular compiler infrastructure for building code analysis frameworks [53]. The analysis accepts a set of annotations on the source code from AV developers (§3.4.1), compiles the source code to LLVM Intermediate Representation (IR), and then applies a domain-specific static code analysis to generate the behavior specification (§3.4.2). An example of the semantic analysis in shown in Figure 2 on top of *Baidu Apollo* (§3.1). Since *Baidu Apollo* is a C++ project, the source code samples in this section are written in C++. However, our code analysis

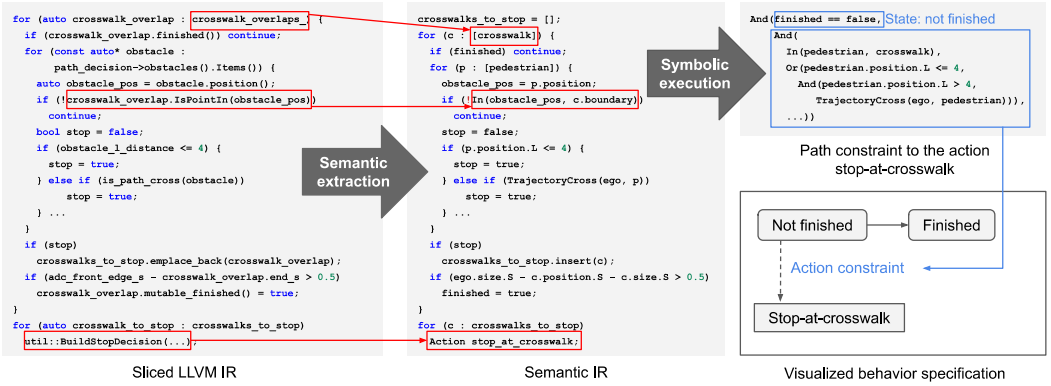


Fig. 2. An example of extracting the behavior specification from AV software code. Developer-provided annotations are notated by red boxes and arrows. We show sliced LLVM IR and semantic IR in C++ source code style for demonstration though *AVChecker* processes IR in the instruction level.

pipeline is also compatible with other programming languages because of LLVM's compatibility with various frontends.

3.4.1 Annotations. The annotation provides semantic information of the AV software by linking elements in the source code with their corresponding expressions in the AV traffic model. Its format is written as an equivalence function where the left side is one statement, variable, or function in the source code and the right side is one action, expression, or API defined in the AV traffic model. Based on different types of expressions used in the equivalence function, annotations are classified into action, variable, and function annotations.

Action annotations identify which statements in the source code make the driving division. While writing the equivalence function, its left side is a pattern of the statements and its right side is the identifier of one specific action. As an example in Figure 2, we link *Apollo*'s function *BuildStopDecision* with action *stop_at_crosswalk* with the following annotation:

```
{"statement": "function_call", "function": "util::BuildStopDecision(...)",
 "parent_function": "apollo::planning::Crosswalk::MakeDecisions(...)"} = stop_at_crosswalk
```

In this case, we identify the function call statement using the signature of the callee function and the parent function. Developers can also use other ways such as file name or line number.

Variable annotations maps variables that affect the driving decision in code (e.g., traffic objects and their size, position, velocity or trajectory) to expressions in the AV traffic model. The left side is a pattern of source code variables while the right side is an expression defined in the AV traffic model, and both sides should have the same semantics. For instance, in order to provide semantic information of the variable *crosswalk_overlaps_* in Figure 2, we need to define the annotation:

```
{"variable": "class_property", "class": "apollo::planning::Crosswalk", "name": "crosswalk_overlaps_"}
  = [crosswalk]
```

The variable pattern identifies the class property "crosswalk_overlaps_" which records the observed crosswalks. The right side is the crosswalk object defined in the AV traffic model.

Function annotations links functions in the source code, mainly including functions calculating geometric relations, to APIs defined in the AV traffic model. Although these functions are called in multiple places in the source code with different parameters, their semantics remain unchanged. In this case, we use one function annotation with parameters to annotate all related function calls at one time to avoid repeated annotations and save manual effort. In Figure 2, function *IsPointIn* in *Baidu Apollo* has an annotation as follows:

```
{"function": "apollo::common::math::Polygon2d::IsPointIn(Vec2d)", "arguments": ["area", "point"]}
  = {"function": In, "arguments": ["point", "area"]}
```

In this case, the function *IsPointIn* from the source code and the model's API *In* (geometric relation "point in area") share the same semantics and both have two arguments. The annotation states the function signature of *IsPointIn* and the name of API *In*. Moreover, there is a list of arguments on both sides of the equivalence function, which have the same elements but in different orders. The two argument lists indicate the mapping of arguments. Based on these information, we can construct the in-model expression which has the same semantics as the return value of *crosswalk_overlap.IsPointIn(obstacle_pos)* in Figure 2. First, function *IsPointIn* is mapped to API *In*. Second, the arguments of *IsPointIn*, *crosswalk_overlap* and *obstacle_pos* are translated to in-model expressions *crosswalk.boundary* and *pedestrian.position* respectively according to other annotations. Third, we fill translated arguments into API *In* in the right order according to the argument lists in the function annotation. Finally, *crosswalk_overlap.IsPointIn(obstacle_pos)* is equal to *in(pedestrian.position, crosswalk.boundary)* in semantics.

3.4.2 Pipeline of code analysis. Semantic analysis builds the behavior specification through a series of code analysis techniques including *program slicing*, *semantic extraction* and *symbolic execution*. Before code analysis, *AVChecker* first compiles the AV software to LLVM IR and identifies entries and exits of analysis. Entries are the starting execution point of the motion planning module while exits are the annotated actions.

Program slicing picks the code of driving rule handling from the original large codebase to minimize the code size to analyze. First, *AVChecker* constructs a customized call graph. A function should be included in the call graph if and only if it could be invoked on at least one possible execution path from entries to exists. Second, *AVChecker* traverses this call graph to perform instruction-level slicing. The slicing in each function is a backward process starting from a set of *sink points*. The sink points include function calls to other functions in the call graph, exits, and return instructions if exist. Then, *AVChecker* selects a subset of instructions that has control or data dependencies on these sink points. The selected instructions construct the final sliced LLVM IR program which captures the minimal code for understanding AV's driving rule handling logic.

After slicing, we pinpoint state variables (§3.3.2) in the code through pattern matching. State variables have finite definition domain (i.e., in the type of boolean values or integers), are read or written in the execution cycle, have dependencies on the actions, and are not annotated (i.e., no relation with AV traffic model). By identifying the state variables and all updates on them in the sliced code, *AVChecker* can now construct a state machine where states represent different values of state variables and transitions refer to one update on the state variables. However, compared with the definition of the behavior specification (§3.3.2), transition or action conditions are missing. The next two steps, semantic extraction, and symbolic execution are going to extract the constraints and complete the behavior specification.

Semantic extraction transforms the sliced LLVM IR to another intermediate form where all variables are expressions defined in the AV traffic model and all operators are compatible with SMT formula. We call this intermediate form semantic IR.

The translation from sliced LLVM IR to semantic IR is done instruction by instruction based on two lookup tables: variable lookup table and operator lookup table. The variable lookup table contains all variable and function annotations (§3.4.1) and maps variables in sliced LLVM IR to expressions in the AV traffic model. The operator lookup table maps operators in LLVM/C++ to operations that are compatible with SMT. Most LLVM's arithmetic and comparison operators have corresponding SMT operators. For example, from LLVM's *add/fadd*, *sub/fsub*, *mul/fmul*, *udiv/sdiv/fdiv* to SMT's "+,-,*,/", from LLVM's *icmp/fcmp gt/lt/eq* to SMT's ">,<,=". Typecasting

among different bit widths in LLVM is all mapped to equivalence in SMT formula because SMT implements real numbers, integers, and boolean values without considering bit width. For memory operations such as *load* and *getelementptr*, *AVChecker* applies static pointer analysis (e.g., *SVF* [56]) to identify which variable the address is stored to and maps the memory access to one simple assignment. For example, considering the load instruction *a = load addr*. If we know *addr* stores variable *b*, then the *load* instruction can be mapped to the assignment *a = b*. Note that a pointer can reference different variables depending on which path the program is executed. In this case, the semantic IR records all the possible variables and their corresponding parent basic blocks, which is similar to a *phi* instruction in LLVM [38]. Semantic IR *a = phi[b bb1, c bb2]* states that, when executing this instruction, variable *a* is assigned by *b* if the last basic block of the execution is *bb1* and is assigned by *c* if the last basic block is *bb2*. In addition to LLVM instructions, the operator lookup table also interprets some higher-level standard library calls. For instance, the C++'s vector operations are mapped to abstracted array operations.

Given predefined lookup tables, *AVChecker* traverses instructions in LLVM IR and translates them from LLVM IR to semantic IR while preserving control flows. We first translate the arguments to corresponding SMT-compatible variables, which are either defined by earlier semantic IR instructions or defined by annotations in the variable lookup table (if the pattern of the argument matches the left side of annotations, as mentioned in §3.4.1). Then, according to the operator lookup table, the LLVM operation with original arguments is converted to the SMT-compatible operation with translated arguments. Finally, a new variable is initialized as the left value in semantic IR. Note that the *AVChecker* will abort and throw errors to notify AV developers to refine their annotations if there is an undefined variable/operator or a mismatch on the data types of function arguments with the function signatures. As shown in Figure 2, variables in the semantic IR are all replaced by in-model expressions but program control flows do not change.

To sum up, semantic IR maintains the control flows and the code logic in LLVM IR but replaces the program with an SMT-compatible format. During this process, a few code details which are irrelevant with driving rules are dropped, including bit-width of data types, pointers, and exceptions (because we do not annotate them). The semantic IR cannot handle complicated class type casting and function pointers but we do not observe them in driving rule handling of *Apollo* and *Autoware*. It can neither handle complex mathematical computations that exceed the limit of SMT's first-order logic but those are usually lower-level details of annotated functions so that they do not block the translation. More discussions on the limitations of semantic IR are presented in §4.4.

Symbolic execution extracts action or transition constraints, which are part of the behavior specification (§3.3), from the semantic IR. Using our proposed program slicing and pointer elimination in semantic analysis, we can significantly reduce the code size and avoid complicated memory modeling, making *AVChecker*'s symbolic execution scalable on AV software. The symbolic execution engine used in this work is the same as a standard symbolic execution algorithm [23, 30]. It processes instructions one-by-one, forks or merges when having branches, and maintains program states throughout the program execution. Then, *AVChecker* traverses the execution paths from program entries to the points of updating state variables or taking actions and produces path constraints, which are conjunctions of branch conditions along the execution path. By definition, if the inputs of the program satisfy the path constraint, the program execution will reach the specific program point. Given a specific program point, the corresponding constraint is the union of path constraints from all entries to the exit. Finally, we can fill the path constraints as transition or action constraints in the behavior specification to make it complete. As shown in Figure 2, the path constraint to the action *stop_at_crosswalk* is the action constraint in the final specification.

3.5 Violation identification

The violation identification aims to identify violations between the behavior specification and the rule specifications. *Violation* is a specific traffic scene snapshot in which the behavior of AV conflicts with the driving rule. We generate an SMT constraint representing the existence of the violation and use an SMT solver to solve a solution that identifies the violation.

Definition 3.1 (State constraint)**.** Given behavior specification (M, S, δ, α) where $\delta(s_m, s_n)$ is the constraint of transition from state $s_m \in S$ to $s_n \in S$, the state constraint for state $s_i \in S$ is $C_{si} = \neg(\bigvee_{s_j \in S, i \neq j} \delta(s_i, s_j))$.

Given the behavior specification, we first extract the implicit relationship between software states and snapshots. We calculate a constraint of possible snapshots for each state in the behavior specification, called *state constraint*. As shown in Definition 3.1, the state constraint is the complement of outgoing transition constraints, because snapshots triggering outgoing transitions always make the current state unstable and are impossible to coexist with the current state. For instance, in the sample behavior specification presented in Section 3.3.2, once the AV passes the crosswalk, the state will shift from "finished=false" to "finished=true". Hence the state constraint of the state "finished=false" is "AV has not passed crosswalk", which correctly captures the state's semantics.

For one traffic snapshot, state constraint C_{si}, $s_i \in S$ is one necessary condition for the software state to be s_i. This is because if $\neg C_{si}$ is satisfied, at least one transition from s_i takes effect so that the current state cannot be s_i.

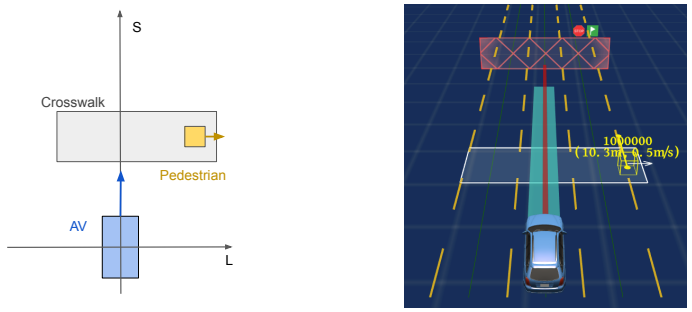
Definition 3.2 (Violation constraint)**.** Given the action space A, realistic constraint C_m, behavior specification (M, S, δ, α), rule specification (M, a, μ, C_r) and state constraints $C_{si}, s_i \in S$, the violation constraint C_v is: $C_m \wedge C_r \wedge \bigvee_{s_i \in S}(C_{si} \wedge ((\neg\alpha(s_i, a) \wedge \mu = 1) \vee (\alpha(s_i, a) \wedge \mu = 0)))$.

Next, we reason the existence of violations as shown in Definition 3.2. First of all, a violation must always satisfy the realistic constraint to be a valid snapshot, and the rule constraint to be within the scope of rule specification. The violation may happen in any software state, but it must satisfy the corresponding state constraint and trigger actions differently from the rule specification. Concatenated all constraints together, we have a constraint of violations.

The violation constraint is one necessary condition for a snapshot to be a violation. Realistic constraint C_m and rule constraint C_r by definition are both necessary conditions of being a violation. Given the definition of violations, for one state $s_i \in S$, $(\neg\alpha(s_i, a) \wedge \mu = 1) \vee (\alpha(s_i, a) \wedge \mu = 0)$ is one necessary condition of there is one violation on s_i. By combining the constraint for each software state, the violation constraint is one necessary condition of having one violation on at least one software state.

The final step is to solve one violation case or prove the absence of violations, which is a satisfaction problem. We query SMT solver *Z3* for finding a solution to the violation constraint. If *Z3* finds the violation constraint to be unsatisfiable, the rule violation does not exist between the two specifications. Otherwise, *Z3* will produce a solution that represents a violation case.

Correctness. The correctness is under the assumption that the rule specification and the behavior specification correctly model driving rule logic and AV's behavior respectively. Since the violation constraint is one necessary condition for a snapshot to be a violation, all possible violations satisfy the violation constraint. Assuming the SMT solver is correct and sound, if the violation constraint is unsatisfiable, it ensures no possible violation exists between the specifications. However, the produced violation case is not guaranteed to be a real violation so *AVChecker* deploys the validation module to rule out false positives.



(a) Visualized violation case. (b) Screenshot of simulator.

Fig. 3. Validation of the example rule in §3.1 on *Apollo*.

3.6 Validation

The validation module aims to examine found violations in the simulator, which simulates more realistic details than abstracted models so that it can rule out false positives caused by model abstractions. As discussed in §3.5, the violation constraint is only a necessary constraint of being a violation. This is because the realism constraint in AV traffic model may not include all constraints about the real-world driving. For instance, AV with a very high speed right after a stop sign is possible according to the specification but is impossible in the real world because the AV will stop first and the acceleration is limited. Meanwhile, inaccuracy of the semantic analysis may also cause false positives. Fortunately, we can simulate the violation case to quickly rule out these violations that are impossible to happen.

AVChecker automatically transforms the violation case produced by the SMT solver to a test case for AV simulators. The test case generation contains two steps. First, *AVChecker* generates a map for simulation according to the relative positions of static objects (e.g., crosswalks, stop signs, and intersections). Second, *AVChecker* puts moving objects (e.g., AV, other vehicles, and pedestrians) on the map and set their initial status. Then the developer can replay the case in the simulator and observes whether the violation really occurs. The validation module also supports a fuzzing mode, which uses the violation case as a seed to generate more simulation test cases to understand the violation better. The details of generating test cases depends on the AV software and simulation platform. For both *Apollo* and *Autoware*, we use HD maps to configure static objects and ROS packets [10] to manage moving objects.

One such generated test case of the example (§3.1) is shown in Figure 3, which automatically generates a straight road, a crosswalk, and a pedestrian in the simulator. The simulation confirms the violation that *Apollo* does not stop when the pedestrian is still on the crosswalk. The violation is because *Apollo* thinks the stop is unnecessary as long as the pedestrian is not very close and moving very slowly, which is a bit aggressive. Though the *Apollo*'s behavior does reduce the probability of collisions, pedestrians on the crosswalks can still be frightened by a full-speed AV.

4 EVALUATION

We evaluate our prototype on two popular open-source AV software platforms, *Baidu Apollo 5.5* [3] and *Autoware 1.13* [6], by checking traffic rules defined in the DMV driving manual and intended driving behaviors inferred from commit logs in their software repository. We first go through the evaluation on *Baidu Apollo*, which has the most detailed driving rule handling to our best knowledge, and then present the checking on *Autoware* in §4.7. We aim to answer the following questions: 1) How effective can *AVChecker* detect driving rule violations; 2) How long does *AVChecker* take to check real-world rules on AV software; 3) How much savings of manual efforts can *AVChecker*

achieve; 4) How general can our framework apply to different AV software platforms and different driving rules. All experiments are executed on a server (Intel Xeon CPU E5-4620v2 2.60GHz, 128GB RAM). Our experiments show that *AVChecker* detects 13 violations out of 32 test cases for *Apollo* and 6 out of 8 test cases for *Autoware*. The semantic analysis for each software takes less than 25 minutes and the violation identification process finishes in 13 seconds for each rule. The simulation finds **no false positive** in our test cases.

4.1 Implementation

We introduce the implementation effort on three modules. **Semantic analysis** is implemented as an LLVM pass of 3,475 LOC C++. We leverage default LLVM analyses, including control flow graph (CFG) and call graph, and *SVF* [56] for pointer analysis. We implement semantic IR and symbolic execution from scratch. **Violation identification** (along with the AV traffic model) is implemented by 2,286 LOC Python. We use SMT solver *Z3*'s interfaces including symbolic variables, constraint solving, and quantifier elimination. We manually annotate in-code variables and write rule specifications using interfaces of AV traffic model (§3.2.1). **Validation** is implemented in 814 LOC Python scripts. Each test case contains an HD map and ROS packets [10]. We use simulator *SimControl* [3] and *Carla* [26] for *Baidu Apollo* [3] and *Autoware* [6], respectively.

4.2 Test cases

In this section, we introduce the driving rules for evaluation and how the test cases are generated.

DMV rules. We collected 45 DMV driving rules about various non-trivial scenarios, including crosswalks, traffic lights, stop signs and intersections, described in English [5, 8, 9, 11, 12]. From the driving rule dataset, we manually write 28 rule specifications as the test suite, 5 for crosswalks, 16 for intersections, 4 for traffic lights, and 3 for stop signs.

Some DMV rules have no related specifications because of several reasons. First, some rules do not address issues of decision making. Only the rules with specific actions such as stop and deceleration are considered in our study. Second, some decision-making process does not need to validate because *Apollo* does not implement such features, for instance, the flashing traffic lights. Third, some rules are about routing logic rather than motion planning, e.g., which lane should the vehicle select. Unfortunately, routing is a complicated algorithm in AV software and our code analysis does not handle it. We leave the coverage on routing as future work.

Commit logs. To demonstrate the benefits of *AVChecker* in AV development, we generate extra test cases from Apollo's bug fixing commits and evaluate whether our framework can detect these bugs. We found 4 bug fixing commits within the scope of driving rule logic in total from *Apollo*'s *Github* repository. For each commit, we execute the code analysis twice on the version of code before and after the commit, respectively. Then, we use the two versions of the code to check the same driving rule. If *AVChecker* produces a violation on code version before the commit but no violation after the commit, the implementation bug is correctly captured.

4.3 AV traffic model

We evaluate our AV traffic model in terms of coverage on various traffic scenarios and ease of use.

First, the model can represent a wide range of common traffic objects and scenarios. We implement 9 types of traffic objects in the AV traffic model: vehicle, pedestrian, bicycle, lane, road, crosswalk, traffic light, stop sign, and intersection. These objects are mentioned by both traffic rule implementation in AV software (i.e., *Baidu Apollo* and *Autoware*) and 28 traffic rule specifications we collect. Besides, the AV traffic model is extensible to define other traffic objects using provided APIs (§3.2.1). In our experiment, the whole AV traffic model defines one AV object and one object for each traffic object type though the model by design can define more objects. This configuration

Table 2. Selected APIs of AV traffic model.

API	Description	#op	Depth
On(Point a, Line b)	Point a on line b.	20	8
In(Point a, Area b)	Point a inside area b.	15	7
In(Area a, Area b)	Area a inside area b.	26	9
Cross(Line a, Area b)	Line a cross area b.	28	8
Cross(Area a, Area b)	Area a cross area b.	23	7
TrajectoryCross(Object a, Object b)	Trajectories of two mobile objects cross.	72	11
Wait(Vehicle v, Area a)	The vehicle is waiting for entering area.	11	6

can already support the rule-based validation on 32 test cases we collected because the collected rules contain no interaction among multiple traffic objects in the same type. In the scenario of multiple objects, these rules can be applied to each object independently so that the violation identification has the same result as the single-object scenarios. The final model has 92 base variables (i.e., symbolic variables) in total and the realistic constraint C_m has 342 SMT operators and an Abstract Syntax Tree (AST) depth of 8. Thanks to the abstractions presented in §3.2.2, the AV traffic model is lightweight enough to be easily handled by *Z3* SMT solver.

Second, the AV traffic model's APIs are easy to use. We list some frequently used APIs and the complexity of the corresponding backend SMT representation in Table 2. Developers can write one line of code calling one API to formally represent the corresponding meaning. If not using APIs, developers must manually construct the SMT expression using dozens of operators to represent the same meaning, which is a tedious and error-prone job. For instance, the most complicated API in the table, *TrajectoryCross*, has 72 operators after encoded in SMT. If we construct the meaning of *TrajectoryCross* using the lowest-level symbolic variables, at least 50 LOC Python code is required. As a result, using these APIs significantly reduces manual effort as mentioned in §4.6.

4.4 Semantic analysis

In this section, we discuss the semantic analysis in terms of efficiency and accuracy.

The semantic analysis overall covers the code handling crosswalks, intersections, stop signs, and traffic lights in *Baidu Apollo*. The generated behavior specification for *Baidu Apollo* contains 12 unique states, 27 transition constraints, and 6 action constraints. The most complicated constraint, the action constraint of stopping at the crosswalk, contains 4,566 SMT operators and has an AST with a depth of 18. *AVChecker* processes the compiled bit code which contains 307K LLVM instructions (excluded external libraries), slices the original code to 1,969 instructions, and exploits 671 execution paths in total.

As for efficiency, program slicing can shrink the code size to around 0.6% since 1,969 out of 307K instructions are identified as rule-related. Before code slicing, symbolic execution cannot complete due to the path explosion. After removing irrelevant code in *Baidu Apollo*, the symbolic execution can finish in 20 minutes and the whole semantic analysis takes less than 25 minutes.

The inaccuracy of semantic analysis mainly comes from pointer analysis and manual annotations. In our experiment, the pointer analysis succeeds in resolving each pointer to one unique points-to value for each execution path, which implies no ambiguity. Also, we manually examine the analysis result and observe no failed cases on the code of *Baidu Apollo* and *Autoware*. However, the pointer analysis is not promised to be perfect, which is a general limitation in related work [50]. Second, the accurate semantic analysis relies on the quality of annotations. For each rule-related functions or variables in the code, developers need to provide the corresponding in-model expressions whose semantic meaning is the closest. The semantic extraction (§3.4.2) aborts when there are undefined variables or unmatched data types, which also help to avoid careless mistakes.

Table 3. Test cases of driving rule verification on *Apollo 5.5*.

ID	Action	Rule constraint in English	Rule constraint in formal representation	#op	Dep	T/s	Val
1	Stop at crosswalk	A vehicle has stopped behind the crosswalk.	wait(vehicle,cross) ∧ Approach(ego,cross)	46	9	12.81	TP
2	Stop at crosswalk	Pedestrians on your side of an oncoming crosswalk (example rule in §3.1).	In(ped,cross) ∧ In(ped,road) ∧ Approach(ego,cross)	66	9	12.62	TP
3	Stop at intersection	A pedestrian has not finished crossing the side of the road onto which you are turning.	In(ped,cross) ∧ In(ped,road) ∧ In(cross,inter) ∧ Approach(ego,inter) ∧	88	9	12.62	TP
4	No park	Within 20 feet of a crosswalk.	Cross(cross,road) ∧ Dist(ego,cross) < 20	23	7	0.93	TP
5	No park	AV blocks a crosswalk.	Cross(ego,cross)	23	8	0.64	TP
6	No stop at crosswalk	Unable to decelerate for safely stopping at stop line, no pedestrian close.	Decelerate(cross.start_s) > max_a ∧ Dist(ego,ped) > safe_dist	57	9	12.87	N
6*	No stop at crosswalk	Rule #6 before commit *ae3082a3*.	The same as Rule #6	57	9	5.86	TP
7	No stop at intersection	Right turn on yellow; AV has passed stop line; traffic is clear.	ego.turn = right ∧ signal.color = yellow ∧ Passed(ego,inter.start_s) ∧ ¬TrajectoryCross(ego,vehicle)	158	13	3.01	N
7*	No stop at intersection	Rule #7 before commit *1c1e7e98*.	The same as Rule #7	158	13	4.01	TP
8	Stop at intersection	Approaching an intersection without signs or traffic lights.	Approach(ego,inter) ∧ signal.color = unknown	17	7	3.02	N
8*	Stop at intersection	Rule #8 before commit *e896fa9b*.	The same as Rule #8	19	7	2.85	TP
9	No park	Within 30 feet of a signal.	Cross(signal,road) ∧ Dist(ego,signal) < 30	20	7	0.33	TP
10	Stop at traffic light	Approaching red signal.	Approach(ego,signal) ∧ signal.color = red	10	7	0.41	TP
11	No park	Within 30 feet of a stop sign.	Cross(stop,road) ∧ Dist(ego,stop) < 30	11	6	0.27	TP
12	Stop at stop sign	Another vehicle stops at a stop sign intersection turning right or going straight; meanwhile AV is turning left, conflict exists.	vehicle.turn ≠ left ∧ Wait(vehicle,stop) ∧ ego.turn = left ∧ Wait(ego,stop) ∧ TrajectoryCross(ego,vehicle)	147	13	3.32	TP
13	Stop at stop sign	Another vehicle comes to a stop at an stop sign intersection earlier, conflict exists.	Wait(ego,stop) ∧ Wait(vehicle,stop) ∧ vehicle.arrival < ego.arrival ∧ TrajectoryCross(ego,vehicle)	128	13	3.45	N
13*	Stop at stop sign	Rule #13 before commit *cc8009a2*.	The same as Rule #13	128	13	3.20	TP
14	No stop at intersection	Left turn; traffic light is green; no oncoming traffic.	ego.turn = left ∧ signal.color = green ∧ ¬ TrajectoryCross(ego,vehicle) ...	123	14	2.94	N
15	Stop at intersection	Left turn; traffic light is green; oncoming traffic.	ego.turn = left ∧ signal.color = green ∧ TrajectoryCross(ego,vehicle) ...	190	15	2.83	N
16	Stop at intersection	Left turn on red from a one-way street onto another one-way street; have not stopped.	ego.turn = left ∧ signal.color = red ∧ Approach(ego,signal) ...	13	7	3.00	N
17	Stop at intersection	Left turn on red from a one-way street onto another one-way street; after a full stop; traffic approaching from the right exists.	ego.turn = left ∧ signal.color = red ∧ Wait(ego,signal) ∧ TrajectoryCross(ego,vehicle) ...	126	13	2.89	N
18	Stop at intersection	Left turn on red from a two-way street onto another one-way street; have not stopped.	ego.turn = left ∧ signal.color = red ∧ Approach(ego,signal) ...	13	7	2.69	N
19	Stop at intersection	Left turn on red from a two-way street onto another one-way street; after a full stop; traffic approaching from the right exists.	ego.turn = left ∧ signal.color = red ∧ Wait(ego,signal) ∧ TrajectoryCross(ego,vehicle) ...	151	13	2.90	N
20	Stop at intersection	Right turn on red; have not done the full stop.	ego.turn = right ∧ signal.color = red ∧ Approach(ego,signal)	13	7	3.45	N
21	Stop at intersection	Right turn on red; after a full stop; conflicts with oncoming traffic exist.	ego.turn = right ∧ signal.color = red ∧ Wait(ego,signal) ∧ TrajectoryCross(ego,vehicle)	148	13	3.56	N
22	Stop at intersection	Approaching intersection; yield the right-of-way to traffic that is in the intersection.	Approach(ego,inter) ∧ In(vehicle,inter) ∧ TrajectoryCross(ego,vehicle)	129	15	2.51	N
23	Stop at intersection	Approaching intersection; traffic is backed up on the other side and you can not get through.	Approach(ego,inter) ∧ In(vehicle,inter) ∧ Cross(ego.trajectory,vehicle)	64	9	2.44	N
24	Stop at intersection	Waiting at intersection; traffic light is green; should stop for crosswalk.	Wait(vehicle,inter) ∧ signal.color = green ∧ In(ped,cross) ∧ In(cross,inter)	88	9	3.09	N
25	No park	Within an intersection.	Cross(vehicle,inter)	20	7	0.56	N
26	Stop at intersection	A vehicle on the right arrives at the same time, yield to that vehicle.	Wait(ego,inter) ∧ Wait(vehicle,inter) ∧ vehicle.arrival<ego.arrival ∧ vehicle.turn=R	127	13	3.09	N
27	Stop at traffic light	Yellow signal; can safely decelerate to stop.	Approach(ego,signal) ∧ signal.color = yellow ∧ Decelerate(cross.start_s) ≤ max_a	19	7	2.16	N
28	No Stop at traffic light	Green signal; no conflict with other vehicles or pedestrians.	Approach(ego,signal) ∧ ¬TrajectoryCross(ego,vehicle) ∧ ¬TrajectoryCross(ego,ped)	220	15	3.01	N

The first row: #op → the count of SMT operators; Dep → depth of SMT expressions' AST; T/s → Time in seconds; Val → validation result.

4.5 Violation identification

We have 32 test cases for *Baidu Apollo* including 28 driving rules and 4 extra test cases from commit logs (Table 3). We present the number of SMT operators and the depth of SMT expression's AST to indicate the complexity of rule constraints. For each rule, we also show the consumed time of the violation identification process and the validation result, which is either true positive (TP), false positive (FP), or negative (N). 13 validated violations are produced, which reveal potential safety risks caused by ignorance of corner cases, improper checks, or implementation bugs (details in

§5). The other 19 test cases are proved to be violation-free in the specification level. Violations in 4 rules extracted from *Baidu Apollo*'s git commits are correctly detected, which proves the capability of *AVChecker* in capturing implementation bugs. In this section, we evaluate the efficiency and accuracy of violation identification.

As for efficiency, violations can be identified in 13 seconds. The time cost is closely related to the complexity of constraints and is much smaller than state-of-the-art simulation-based AV testing methods. For instance, AV-fuzzer [41], a fuzzer generating simulation cases that violate safety rules, discovers 13 violations in 195 hours.

We evaluate the accuracy using the validation result. All violations can be reproduced in the simulator, implying *no false positives* in our detected violations. Under the assumption that all specifications are sound and complete, the violation identification process will not miss any specification-level violations, as analyzed in §3.5. Compared with simulation-based methods (e.g., AV-fuzzer [41]), *AVChecker* theoretically has a comparable false positive rate because *AVChecker* also uses simulation for the final validation. In terms of false negatives, *AVChecker* has significant advantages since *AVChecker* is free of false negatives under the assumption of sound models and specifications but the simulation-based fuzzing approach can never have the assurance.

4.6 Manual efforts

Manual effort mainly comes from two aspects: annotating the source code, and writing rule specifications. Note that our AV traffic model isolates the code side and rule side, thus analyzing another AV software requires new annotations but there is no need to update rule specifications. On the other hand, supporting new driving rules only requires writing a new rule specification.

Annotations. To extract the behavior specification, developers need to provide three types of annotations as mentioned in §3.4.1. In our implementation, we manually do 5 action annotations, 55 variable annotations, and 9 API annotations written in 170 LOC Python for *Baidu Apollo*, which takes about 4 man-hours assuming the developer has knowledge of the codebase. Over 90% of the variable annotations are simple mappings from one in-code variable to one existing in-model variable. Existing SMT-based system verification work requires a comparable amount of annotation effort. For example, *Hyperkernel* [45] requires 53 annotations written in 250 LOC Python to map variables in source code to states in specifications.

Rule specifications. Given high-level APIs, users can define complex specifications with a couple of lines of code. The number of operators (including APIs of AV traffic model) used in each specification (Table 3) is only about 7% of the SMT operator count of SMT-encoded rule constraints, demonstrating a 15x savings of manual effort. In terms of lines of code, rule specifications can be written in less than 10 lines but 60 lines are needed on average without AV traffic model's APIs.

4.7 Generality to AV software

AVChecker is a general framework for driving rule compliance checking on AV software but we select *Apollo* for demonstration because *Apollo* is one of the best implemented open-source AV software. In this section, we carry out a case study of validating driving rules in *Autoware-1.13* [6] to show the deployability of *AVChecker* on various platforms.

Autoware uses state machines to manage the vehicle state, which perfectly satisfies the formulation in §3.3.2. Our extracted behavior specification contains the 21 states, 25 transitions as well as 8 action constraints, and requires 38 manual annotations. Rule specifications are not changed. However, *Autoware* does not have a well-defined driving rule implementation about intersections. As a result, from Table 3, we evaluate eight rules in total for *Autoware*: three crosswalk rules (i.e., rules 1,2,6), three traffic light rules (i.e., rules 10,27,28) and two parking rules (i.e., rules 9,11,25).

AVChecker finishes the violation identification process for each rule in 10 seconds. As for crosswalk rules, *AVChecker* finds a violation of rule 1 and proves that rules 2, 6 have no violation. *Autoware* does not implement rule 1 either and we validated the violation using simulator *Carla* [26]. As for traffic light rules, *AVChecker* finds violations on rules 27 and 28. *Autoware* only implemented three signal colors: red, green, and unknown without yellow signals, which caused the violation of rule 27. Also, *Autoware* does not consider the traffic inside the intersection when dealing with traffic lights, which violates rule 28. *AVChecker* produces violations on all three parking rules since *Autoware* does not explicitly enforce any no-parking areas. As a whole, *Autoware*'s implementation of driving rule handling does not cover a lot of corner cases, which results in violations.

In conclusion, *AVChecker* can also be applied on *Autoware* and by design all AV software that satisfies the state machine formulation in §3.3.2.

5 FINDINGS

5.1 Violation findings

The 32 test cases produce 13 validated violations in *Apollo* as listed in Table 3. Particularly, violations of rules 1,2,3 causes risk of collision with pedestrians while violations of rule 4,5,9,11 may block normal traffic. Through code reading on *Apollo* we summary three types of causes of violations: *not implemented*, *over restricted* and *implementation bugs*.

Violations of rules 1, 4, 5, 9, 11, 12 are caused by not implemented features. For 4 parking rules, *Apollo* defines "keep clear" areas to ban illegal parking actions but fails to restrict all no-parking landmarks, which may block normal traffic. Especially, rules 1, 12 reveal the ignorance of special cases in AV software and we do detailed case studies in §5.2.

Violations of rules 2, 3, 6*, 7*, 10 are because the code applies unsafe constraints on the action taking. For instance, rule 10 is caused by an extra check on the deceleration rate. *Apollo* may skip stop actions if stop distance is not sufficient for deceleration. However, skipping stop actions sometimes is not safe and there exist accidents of AV caused by the failure of deceleration [22].

Violations of rules 8*, 13* are caused by implementation bugs and these test cases are indeed constructed from *Apollo*'s bug-fixing commit logs. In these cases, some predicates are incorrect which causes abnormal decision making. *AVChecker* can detect such implementation bugs effectively.

5.2 Case studies

Besides rule 2 (§3.1), we study rule 1 and rule 12 in Table 3 by analyzing the safety consequences.

Crosswalk rule #1. In *Apollo*, the code about crosswalk handling considers the distance of pedestrians away from the AV, the position and trajectory of the pedestrian, etc. However, *Apollo* does not consider the behavior of other vehicles in the crosswalk scenario, which differs from what traffic rules state. This driving rule is designed for minimizing the safety risk caused by occlusion, which is also a problem for AV sensors. Figure 4 in Appendix shows the simulation of the generated violation scene with a sprinting pedestrian on the crosswalk. The simulation results in AV's hitting the pedestrian in the end as he was obstructed by the adjacent vehicle and AV's perception did not detect him until very close to him.

Stop sign rule #12. *Apollo*'s implementats of stop sign rules in a state machine whose states include *pre-stop*, *stop*, *creep*, and *cruise*. State *stop* takes the "stop st stop sign" action and state *creep* cancels the action. When turning left (i.e., transiting from *stop* to *creep*), the driving rule requires the AV to yield vehicles that turning right or going straight but *Apollo* does not consider the turning directions. This rule has no severe safety consequence but affects the efficiency of traffic. Human drivers have a common sense that vehicles that are going straight have higher priority than those

who are turning left. If AV does not follow the same principle but shares the same road with human drivers, unnecessary conflicts are likely to occur.

6 LIMITATIONS

We discuss the limitations of *AVChecker* in two aspects.

Scope of violation identification. *AVChecker* can find violations against driving rules in the code semantics of AV software's motion planning module. *AVChecker* checks whether the developers are adopting the right driving rules but cannot reveal of low-level bugs such as null pointers and buffer overflow. This is because the general driving rules, written in natural language and designed for human, only contains a high-level description of the driving scenarios without considering low-level data processing in software. AV developers can deploy *AVChecker* along with conventional software testing techniques such as unit tests and fuzzing so that both low-level bugs and high-level logic are covered.

Also, *AVChecker* validates driving rule implementation in the module of motion planning specifically. *AVChecker* does not cover other AV software modules such as perception through sensors and control over cyber-physical components. We require the driving rule specifications to consider possible failures from other modules (e.g., rule #1 in Table 3 considers perception failures because of occlusion). We leave the model-based correctness checking on other modules as future work.

Modeling capability. The expressing ability of AV traffic model has restrictions since SMT theorem solving based on first-order logic by design cannot accurately represent complicated non-linear algorithms. To alleviate the restriction, *AVChecker* relies on manual annotations to interpret the semantic meaning of complicated computation and allows developers to extend the model by adding APIs or properties of traffic objects. For example, though the software code computes whether two trajectories cross by iterating each trajectory point in one function, the AV developer can annotate the function that it is equal to "line cross line" geometric relation in the model, which captures the code logic but ignores lower-level details. Also, the model cannot support an infinite number of traffic objects due to the limitation of SMT's first-order logic.

7 RELATED WORK

Testing and validation of autonomous driving. This paper focuses on safety validation of autonomous vehicles. Autonomous driving is mission critical so it is important to validate an AV system is safe and secure before its real-world deployment [31, 32, 36, 37]. Simulation is the main approach to testing an AV system. Autonomous driving simulators [26, 52] enable efficient AV testing in realistic driving scenes and simulation-based safety validation [15, 17, 41, 43, 47, 62] tries to maximize the efficiency of testing. Besides, software testing [29, 57, 61] is applied on AV software, which efficiently generates critical test cases to search for safety vulnerabilities. Formal models [14, 16, 42, 54] are proposed to model the behavior of drivers or safety-related driving situations. However, existing formal models cannot check code implementation and are limited in checking specific driving rules. More related AV testing approaches are summarized in §2.2 and there is no work finding violations of scenario-dependent driving rules as *AVChecker* does.

SMT-based testing or verification. Satisfiability Modulo Theories (SMT), as an extended form of Boolean satisfiability (SAT), represents a constraint solving problem for logical formulas with respect to background theories expressed in classical first-order logic. SMT provides encoding of formal variables at a higher level expressiveness and becomes an important backend tool for formal verification or test case generation techniques [19, 20, 59]. SMT-based verification is applied for detecting vulnerabilities or bugs on various domains including operating system [45, 55], GPU program [40], side-channel defenses [27], neural networks [34] and so on. Generally, SMT-based verification first encodes the domain to SMT semantic and then applies SMT reasoning to prove

theorems. On the other hand, SMT-based test case generation leverages SMT solving to identify valuable test cases in a large search space [24, 33, 35, 46]. *AVChecker* is also in the category of test case generation which uses the generated driving scenarios to test AV software. But different from existing SMT-based test case generation for programs [24, 46], the driving scenario is much more complicated since it contains the spatio-temporal information in the real world. *AVChecker* designs sound modeling of traffic snapshots to enable the SMT-based test case generation for AV testing.

8 CONCLUSION

We propose a framework *AVChecker* for identifying violations of scenario-dependent driving rules in AV software. In the core of the framework, we leverage a domain-specific abstraction to bridge the semantic gap between software code and safety-related specifications. Also, automatic processes minimize the required human effort and make the framework deployable in the real world. Our evaluation on *Apollo* and *Autoware* reveal potential safety risks effectively.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is partially supported by National Science Foundation (NSF) under the Award Numbers CNS-1930041, CNS-1544678, CNS-1850533, and CNS-1929771, and the Office of Naval Research under the Award Number N00014-18-1-2020.

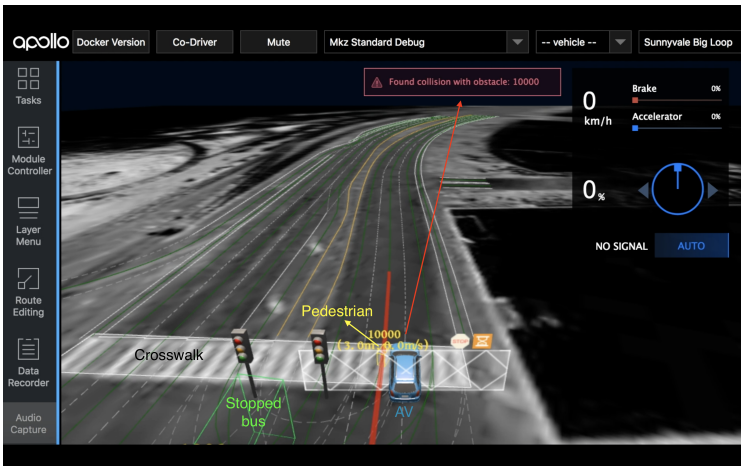


Fig. 4. Snapshot of simulation to validate detected violation with Crosswalk rule #1 *Stop at a crosswalk if a vehicle has stopped behind the crosswalk* in Baidu Apollo

A VIOLATION SIMULATION

Figure 4 is captured from the simulation of an AV running Baidu Apollo in the traffic scene generated by *AVChecker* with a sprinting pedestrian on the crosswalk. This rule violation in Baidu Apollo leads to AV's hitting a pedestrian on a crosswalk walking across the street that has been obstructed by a stopped bus until approaching very close to him.

REFERENCES

[1] 2017. Automated Driving Systems 2.0: A Vision for Safety. https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-ads2.0_090617_v9a_tag.pdf.

[2] 2019. A Matter of Trust Ford's Approach to Developing Self-driving Vehicles. https://media.ford.com/content/dam/fordmedia/pdf/Ford_AV_LLC_FINAL_HR_2.pdf.

[3] 2019. ApolloAuto: An open autonomous driving platform. https://github.com/ApolloAuto/apollo.

[4] 2019. General Motors 2018 Self-Driving Safety Report. https://www.gm.com/content/dam/company/docs/us/en/gmcom/gmsafetyreport.pdf.

[5] 2020. 2010 Georgia Code Title 40 - Motor Vehicles and Traffic. https://law.justia.com/codes/georgia/2010/title-40/chapter-6/article-5/40-6-91.

[6] 2020. Autoware: Open-source software for self-driving vehicles. https://gitlab.com/autowarefoundation/autoware.ai.

[7] 2020. Baidu Apollo's fuzzing support. https://github.com/ApolloAuto/apollo/commit/7aca63966211ceada44981d96b35a1252f9f1729.

[8] 2020. Louisiana DMV Handbook. https://driving-tests.org/wp-content/uploads/2018/03/LA_Guide-2017.pdf.

[9] 2020. New York State DMV: Driver's Manual and Practice Tests. https://dmv.ny.gov/driver-license/drivers-manual-practice-tests.

[10] 2020. ROS.org | Powering the world's robots. https://www.ros.org/.

[11] 2020. State of California DMV: California Driver Handbook. https://www.dmv.ca.gov/web/eng_pdf/dl600.pdf.

[12] 2020. State of Michigan DMV: What Every Driver Must Know. https://www.michigan.gov/documents/wedmk_16312_7.pdf.

[13] 2020. Waymo Safety Report. https://waymo.com/safety.

[14] Matthias Althoff and John M Dolan. 2011. Set-based computation of vehicle behaviors for the online verification of autonomous vehicles. In *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1162–1167.

[15] Matthias Althoff and Sebastian Lutz. 2018. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1326–1333.

[16] Matthias Althoff, Olaf Stursberg, and Martin Buss. 2009. Model-based probabilistic collision detection in autonomous driving. *IEEE Transactions on Intelligent Transportation Systems* 10, 2 (2009), 299–310.

[17] Alexander Amini, Igor Gilitschenski, Jacob Phillips, Julia Moseyko, Rohan Banerjee, Sertac Karaman, and Daniela Rus. 2020. Learning Robust Control Policies for End-to-End Autonomous Driving From Data-Driven Simulation. *IEEE Robotics and Automation Letters* 5, 2 (2020), 1143–1150.

[18] Mathieu Barbier, Alessandro Renzaglia, Jean Quilbeuf, Lukas Rummelhard, Anshul Paigwar, Christian Laugier, Axel Legay, Javier Ibañez-Guzmán, and Olivier Simonin. 2019. Validation of perception and decision-making systems for autonomous driving via statistical model checking. In *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 252–259.

[19] Dirk Beyer and Matthias Dangl. 2016. SMT-based software model checking: an experimental comparison of four algorithms. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 181–198.

[20] Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2018. A unifying view on SMT-based software verification. *Journal of automated reasoning* 60, 3 (2018), 299–335.

[21] Nikolaj Bjørner. 2010. Linear quantifier elimination as an abstract decision procedure. In *International Joint Conference on Automated Reasoning*. Springer, 316–330.

[22] NTS Board. 2018. Preliminary report, highway, hwy18mh010. *National Transpotation Safety Board, https://www. ntsb. gov/investigations/AccidentReports/Reports/HWY18MH010-prelim. pdf, accessed* (2018), 11–15.

[23] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. 209–224.

[24] Silviu S Craciunas and Ramon Serna Oliver. 2014. SMT-based task-and network-level static schedule generation for time-triggered networked systems. In *Proceedings of the 22nd international conference on real-time networks and systems*. 45–54.

[25] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[26] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. *arXiv preprint arXiv:1711.03938* (2017).

[27] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. SMT-based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 62–77.

[28] Haoyang Fan, Fan Zhu, Changchun Liu, Liangliang Zhang, Li Zhuang, Dong Li, Weicheng Zhu, Jiangtao Hu, Hongye Li, and Qi Kong. 2018. Baidu apollo em motion planner. *arXiv preprint arXiv:1807.08048* (2018).

[29] Daniel S Fowler, Jeremy Bryans, Siraj Ahmed Shaikh, and Paul Wooderson. 2018. Fuzz testing for automotive cybersecurity. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 239–246.

[30] David Ke Hong, Qi Alfred Chen, and Z. Morley Mao. 2017. An Initial Investigation of Protocol Customization. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*.

[31] David Ke Hong, John Kloosterman, Yuqi Jin, Yulong Cao, Qi Alfred Chen, Scott Mahlke, and Z Morley Mao. 2020. AVGuardian: Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 445–459.

[32] Ke Hong. 2019. *Performance, Security, and Safety Requirements Testing for Smart Systems Through Systematic Software Analysis*. Ph.D. Dissertation. University of Michigan, Ann Arbor, MI.

[33] Eunkyoung Jee, Donghwan Shin, Sungdeok Cha, Jang-Soo Lee, and Doo-Hwan Bae. 2014. Automated test case generation for FBD programs implementing reactor protection system software. *Software Testing, Verification and Reliability* 24, 8 (2014), 608–628.

[34] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 97–117.

[35] BaekGyu Kim, Akshay Jarandikar, Jonathan Shum, Shinichi Shiraishi, and Masahiro Yamaura. 2016. The SMT-based automatic road network generation in vehicle simulation environment. In *2016 International Conference on Embedded Software (EMSOFT)*. IEEE, 1–10.

[36] Philip Koopman and Michael Wagner. 2016. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety* (2016).

[37] Philip Koopman and Michael Wagner. 2017. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine* (2017).

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[39] John Leonard, Jonathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, Olivier Koch, Yoshiaki Kuwata, David Moore, Edwin Olson, Steve Peters, Justin Teo, Robert Truax, Matthew Walter, David Barrett, Alexander Epstein, Keoni Maheloni, Katy Moyer, Troy Jones, Ryan Buckley, Matthew Antone, Robert Galejs, Siddhartha Krishnamurthy, and Jonathan Williams. 2008. A Perception-Driven Autonomous Urban Vehicle. *Journal of Field Robotic* (2008).

[40] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 187–196.

[41] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Siva Hari, Michael Sullivan, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. AV-FUZZER: Finding safety violations in autonomous driving systems. *ISSRE'20, Proceedings of the IEEE International Conference on Software Reliability Engineering* (Nov 2020).

[42] Nan Li, Dave W Oyler, Mengxuan Zhang, Yildiray Yildiz, Ilya Kolmanovsky, and Anouck R Girard. 2017. Game theoretic modeling of driver and vehicle interactions for verification and validation of autonomous vehicle control systems. *IEEE Transactions on control systems technology* 26, 5 (2017), 1782–1797.

[43] Satoshi Masuda, Hiroaki Nakamura, and Kohichi Kajitani. 2018. Rule-based searching for collision test cases of autonomous vehicles simulation. *IET Intelligent Transport Systems* 12, 9 (2018), 1088–1095.

[44] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, D. Johnston, S. Klumpp, D. Langer, A. Levandowski, J. Levinson, J. Marcil, D. Orenstein, J. Paefgen, I. Penny, A. Petrovskaya, M. Pflueger, G. Stanek, D. Stavens, A. Vogt, and S. Thrun. 2008. Junior: The Stanford Entry in the Urban Challenge. *Journal of Field Robotics, Special Issue on the 2007 DARPA Urban Challenge, Part II* (2008).

[45] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 252–269.

[46] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated test case generation for P4 programs. In *Proceedings of the Symposium on SDN Research*. 1–7.

[47] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, Russ Tedrake, and John C Duchi. 2018. Scalable end-to-end autonomous vehicle testing via rare-event simulation. In *Advances in Neural Information Processing Systems*. 9827–9838.

[48] Anshul Paigwar, Eduard Baranov, Alessandro Renzaglia, Christian Laugier, and Axel Legay. 2020. Probabilistic Collision Risk Estimation for Autonomous Driving: Validation via Statistical Model Checking. In *31st IEEE Intelligent Vehicles Symposium*.

[49] Christian Pek, Peter Zahn, and Matthias Althoff. 2017. Verifying the safety of lane change maneuvers of self-driving vehicles based on formalized traffic rules. In *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1477–1483.

[50] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 925–935.

[51] Albert Rizaldi, Fabian Immler, and Matthias Althoff. 2016. A formally verified checker of the safe distance traffic rules for autonomous vehicles. In *NASA Formal Methods Symposium*. Springer, 175–190.

[52] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Martinš Mozeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. [n.d.]. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. ([n. d.]).

[53] Tao B Schardl, Tyler Denniston, Damon Doucet, Bradley C Kuszmaul, I-Ting Angelina Lee, and Charles E Leiserson. 2017. The CSI framework for compiler-inserted program instrumentation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 2 (2017), 1–25.

[54] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2017. On a Formal Model of Safe and Scalable Self-driving Cars. *CoRR* (2017).

[55] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[56] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[57] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[58] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William "Red" Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. 2008. Autonomous Driving in Urban Environments: Boss and the Urban Challenge. *Journal of Field Robotics* (2008).

[59] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–31.

[60] J. Wei, J. M. Snider, T. Gu, J. M. Dolan, and B. Litkouhi. 2014. A behavioral planning framework for autonomous driving. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*.

[61] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 132–142.

[62] Marc René Zofka, Sebastian Klemm, Florian Kuhnt, Thomas Schamm, and J Marius Zöllner. 2016. Testing and validating high level components for automated driving: simulation framework for traffic scenarios. In *2016 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 144–150.

Received February 2021; revised April 2021; accepted April 2021