Leveraging Contracts for Failure Monitoring and Identification in Automated Driving Systems*

Srajan Goyal[1,2][0009−0005−2189−7026] sgoyal@fbk.eu, Alberto Griggio[1][0000−0002−3311−0893] griggio@fbk.eu, and Stefano Tonetta[1][0000−0001−9091−7899] tonettas@fbk.eu

[1] Fondazione Bruno Kessler, Trento, Italy
[2] University of Trento, Trento, Italy

Abstract. As the deployment of AI agents in Automated Driving Systems (ADS) becomes increasingly prevalent, ensuring their safety and reliability is of paramount importance. This paper presents a novel approach to enhance the safety assurance of automated driving systems by employing formal contracts to specify and refine system-level properties. The proposed framework leverages formal methods to specify contracts that capture the expected behavior of perception and control components in ADS. These contracts serve as a basis for systematically validating system behavior against safety requirements during design and testing phases. We demonstrate the efficacy of our approach using the CARLA simulator and an off-the-shelf AI agent. By monitoring the components contracts on the ADS simulation, we could identify not only the cause of system failures, but also the situations which could lead to a system failure, facilitating the debugging and maintenance of the AI agents.

Keywords: Formal properties · Runtime verification · Test case generation · Autonomous systems · Assume-guarantee reasoning

1 Introduction

Automated Driving Systems (ADS) represent a technological innovation that is revolutionizing traditional transportation, potentially improving the traffic flow and increasing overall road safety. A central role in the success of ADS is played by AI agents, which may be responsible for various perception and control tasks. However, as ADS deployment accelerates, ensuring the safety and reliability of these AI agents becomes increasingly challenging.

Various formal verification techniques have been proposed to prove properties of AI-based autonomous systems. Much work in the literature [11,19,13,27,15] focuses on system-level simulation-based testing using formal methods to specify

* This work has been supported by: the "AI@TN" project funded by the Autonomous Province of Trento; the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU; and the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

formal properties and coverage criteria, and to generate automatically test cases. In this line of work, AI agents can be seen as black-boxes integrated into the simulator of the autonomous system, while formal properties can rely on the simulation data of the environment that is considered the ground truth and can be used to compare it with the agents' knowledge.

The VIVAS project [12], funded by ESA, developed a generic framework for system-level simulation-based Verification and Validation (V&V) of autonomous systems. This approach uses a simulation model, an abstract model to symbolically describe the system behavior, and formal methods to generate scenarios and verify simulation executions. It permits the specification of diverse coverage criteria, thereby guiding automated scenario creation and verification of formal properties. The framework has been evaluated on space and automotive applications [15]. For the latter, the simulation uses CARLA [10], a popular platform for simulating complex driving scenarios. Various AI-based solutions (e.g., [25,28,4]) for car perception and control components are validated with CARLA, which also hosts a competition to rank such solutions [2]. In [15], we demonstrated the integration of the VIVAS framework with CARLA to effectively generate interesting highway scenarios and evaluate AI-based agents in ADS.

However, one of the issues with these approaches is that the system-level verification focuses on system-level properties, mainly car crashes in the automotive case study. This has the limit of not being able, first, to identify the component that is failing in case of a crash and, second, to monitor component failures that may lead to a crash without an actual occurrence of a crash.

In this paper, we propose to integrate the simulation-based V&V framework with a contract-based design of the perception and control components of the autonomous system. The design-level verification ensures that the assumptions formalized on the environment model, used to generate the test cases, are sufficient to provide the system-level guarantees. Component properties are then monitored on the simulation traces. These two phases are iterated to tune the property specifications, as they may end up being either too strong, failing in the runtime monitoring, or too weak, failing in the contract refinement verification.

We evaluated the approach on the ADS case study using the CARLA simulator and the InterFuser AI agent [25], operating on a highway. The iterations led us to fine-tune challenging formal properties, for example, to formalize the safety conditions on the bounding boxes generated by the perception, and formalizing the safe states for AI-based planner. Through the evaluation, we were able to localize the faults in case of a crash, and monitor the component failures that could have led to a crash, but did not. Moreover, we could pinpoint interesting cases in which the ADS could have avoided the crash, if it acted in time.

2 Related work and Contributions

There has been much recent work on compositional verification approaches using contract reasoning to achieve design-time assurance guarantees for autonomous systems [24,26,18]. These approaches leverage the robustness properties of DNNs to verify closed-loop systems. However, reducing the closed-loop safety properties into I/O properties for DNN is often quite challenging. Other methods approximate sensor models to prove safety properties based on low-dimensional sensor data [16]. However, when applied to systems dependent on rich sensors generating high-dimensional inputs, these methods become impractical.

Recent work has also focused on automatically synthesizing perception contracts to ensure closed-loop safety, using symbolic learning algorithms [1], and on bottom-up approaches such as [23] to generate weak assumptions for DNNs. However, these methodologies lack consideration for dynamic environments and face significant scalability challenges when dealing with multiple DNNs (e.g., both camera and lidar inputs). At some level, all this work relies on the internal structure of DNNs to specify or synthesize the contracts. In contrast, we focus on functional contracts for perception and control components, and their interaction with the environment contracts which specify the assumptions about the dynamic behaviors of objects.

A considerable body of work also addresses the repair of components' specifications when system-level properties are violated. For example, [14] proposes algorithms that synthesize model predictive controllers for ADS by identifying reasons for system-level contract infeasibility and suggesting repairs to the controller specification. [20] automatically searches and repairs contracts based on refinements from a contract library (and implementations) to formalize robotic mission planning specifications when new requirements or inconsistencies arise. [9] presents a simulation-based falsification framework that iteratively constructs contracts as constraints to synthesize safe controllers. However, none of this work considers ML components in their closed-loop analysis or validates the approaches in high-fidelity simulators. In our approach, we manually write the contracts, which could, in principle, be complemented by these methods to repair component contracts when system-level properties are violated. Nevertheless, due to the black-box nature of ML components, repairing their contracts without an implementation remains a significant challenge.

Contributions: Although the proposed methodology contains various elements studied in the literature as summarized above, it integrates them in a unique flow that presents various novel aspects:

- the contract-based design of autonomous systems is supported with contract reasoning that formally ensures specification correctness, while many previous works monitor properties without verifying contract refinement.
- the contracts are monitored on off-the-shelf AI solutions interfacing with perception and control components, while in the many previous works the monitors were applied at the system level.
- formalization of contracts on concrete perception and control components by specifying safety conditions on bounding box representations of cars.
- to the best of our knowledge, this is the first end-to-end methodology for the simulation-based verification of autonomous systems that starts from the design of the components to their monitoring.

3 Background

Transition Systems. Given a finite set V of variables with a (potentially infinite) domain D, we denote with $\Sigma(V)$ the set of assignments to V, i.e. mapping from V to D. Let V' denote a copy of the variables V, which are used to represent the values of V after a transition. A *Transition System* (TS), S is a tuple $S = \langle V, I, T \rangle$, where V is a set of (state) variables, I is a formula over V representing the set of initial states, and T is a formula over $V \cup V'$ representing the set of transitions. A state $s \in \Sigma(V)$ of S is an assignment to the variables V.

A trace σ of S is an infinite sequence of states $\sigma = s_0, s_1, \cdots$, where each s_i is an assignment to V, such that s_0 satisfies I and for all $i \geq 0$, the pair $\langle s_i, s_{i+1} \rangle$ satisfies T, i.e., $T(V, V')$ holds when V is assigned according to s_i and V' according to s_{i+1}. Given two transition systems $S_1 = \langle V_1, I_1, T_1 \rangle$ and $S_2 = \langle V_2, I_2, T_2 \rangle$, we define the synchronous product $S_1 \times S_2$ as $\langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2 \rangle$. Since the product is commutative and associative, it can be generalized to a set of transitions systems.

LTL. Given a set of variables V, we assume to be given a set $Expr(V)$ of Boolean expressions over V as in [21]. In particular, in this paper, we consider standard arithmetic predicates $(<, \leq, >, \geq, \ldots)$ and functions $(+, -, \ldots)$ over integer and real variables. We define the set of LTL formulas over the variables V with the following grammar rule:

$$\phi := p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid X\phi \mid \phi U \phi$$

where p ranges over $Expr(V)$. We use the following standard abbreviations: $\top := p \vee \neg p$, $\bot := \neg \top$, $\phi \rightarrow \psi := (\neg \phi) \vee \psi$, $F\phi := \top U \phi$, $G\phi := \neg F \neg \phi$.

LTL formulas are interpreted over traces over V, which are infinite sequences of assignments to V. We refer the reader to [8] for a formal definition of the semantics.

Given a transition system $S = \langle V, I, T \rangle$ and an LTL formula ϕ over V, $S \models \phi$ if and only if for all traces σ of S, $\sigma \models \phi$. Note that we are considering in general infinite-state transition systems for which the problem is undecidable. Our methods are based on SMT-based algorithms as those implemented in nuXmv [3].

At runtime, we evaluate a formula over the prefix of an infinite execution trace using the standard semantics of runtime verification [17]. In particular, a finite trace π violates a formula ϕ iff, for any suffix π', the concatenation of π and π' violates ϕ.

Contract Refinement. To simplify the presentation, in this paper, we define a contract refinement independently from the component interfaces. In practice, in OCRA [6], contracts are specified in terms of component input/output ports and the refinement has to take into account the connections among ports in component decomposition.

A contract C over the variables V is a pair $\langle \mathsf{A}, \mathsf{G} \rangle$ of LTL formulas over V representing respectively an *assumption* and a *guarantee*. We also denote A by

$\mathcal{A}(C)$, G by $\mathcal{G}(C)$, and $\neg\mathsf{A} \vee \mathsf{G}$ by $nf(C)$. Let $C = \langle \mathsf{A}, \mathsf{G} \rangle$ be a contract over V. Let J and E be the two TSs over V. We say that J is a correct implementation of C iff $J \models \mathsf{A} \rightarrow \mathsf{G}$. We say that E is a correct environment of C iff $E \models \mathsf{A}$. We denote by $\mathcal{I}(C)$ and $\mathcal{E}(C)$, respectively, the set of correct implementations and the set of correct environments of C. Given two contracts C and C' over V, we say that C refines C' (denoted by $C \preceq C'$) iff $\mathcal{I}(C') \subseteq \mathcal{I}(C)$ and $\mathcal{E}(C) \subseteq \mathcal{E}(C')$.

Given a contract C and a set of contracts $Sub = \{C_1, \ldots, C_n\}$, we say that Sub is a refinement of C, written $Sub \preceq C$, iff the following conditions hold:

1. the correct implementations of the sub-contracts form a correct implementation of C: let V be the variables of C, $\{S_1 \times \ldots \times S_n \mid S_1 \in \mathcal{I}(C_1), \ldots, S_n \in \mathcal{I}(C_n)\} \subseteq \mathcal{I}(C)$;
2. for every $C_i \in Sub$, the correct implementation of the other sub-contracts and a correct environment of C form a correct environment of C_i: let V_i be the variables of C_i, $\{E \times S_1 \times \ldots \times S_{j \neq i} \times \ldots \times S_n \mid E \in \mathcal{E}(C),\ \text{for all } j, 1 \leq j \leq n, j \neq i, S_j \in \mathcal{I}(C_j)\} \subseteq \mathcal{E}(C_i)$.

In [7,5], it was proved that the refinement is correct if and only if the following proof obligations ($PO(Sub, C)$) are valid temporal formulas:

$$\begin{aligned} & nf(C_1) \wedge \ldots \wedge nf(C_n) \rightarrow nf(C) \\ & \mathsf{A} \wedge \bigwedge_{1 \leq j \leq n, j \neq i} nf(C_j) \rightarrow \mathsf{A}_i \text{ (for every } i, 1 \leq i \leq n) \end{aligned}$$

4 Contract-Based Methodology

4.1 VIVAS methodology

VIVAS is a V&V framework for generating test cases for autonomous systems (possibly using AI/ML components) via a combination of system-level simulation and symbolic model checking. VIVAS makes use of a symbolic model of the system under verification. This is inherently a transition system, written in the SMV language, which defines an abstract representation of the system behavior. On top of such model, the coverage criteria for the tests and the properties to verify are specified. The approach follows the flow depicted in Fig. 1 (right part): abstract test scenarios are generated from the symbolic model with model checking techniques based on the coverage criteria; the abstract test scenarios are then instantiated by the concretization of the abstract parameters to provide concrete scenarios to be executed on a system-level simulator encompassing AI/ML models; the simulation traces are in turn analyzed to check the properties defined on the symbolic model. The output of the framework is a V&V result consisting of coverage statistics of the executed traces with respect to the symbolic models, and quantitative and qualitative information for each use case.

An important observation is that the simulation traces can be instrumented to contain information on the autonomous system internal state as well as the

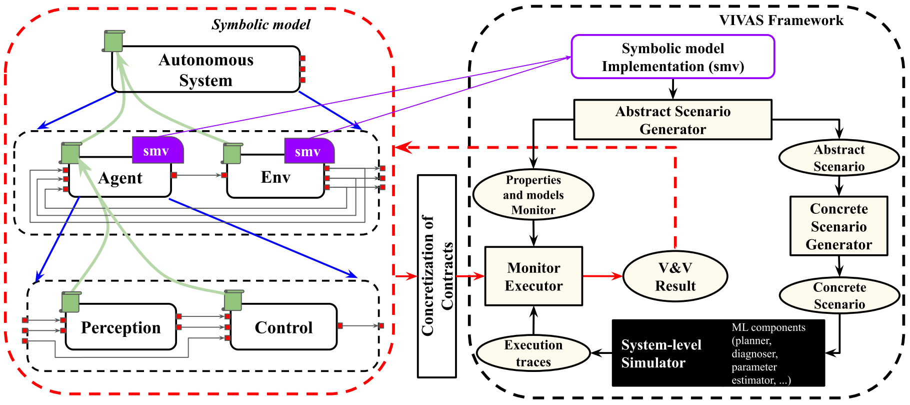
Symbolic model
Autonomous System
Agent
smv
Env
smv
Perception
Control
Concretization of Contracts
VIVAS Framework
Symbolic model Implementation (smv)
Abstract Scenario Generator
Abstract Scenario
Properties and models Monitor
Concrete Scenario Generator
Monitor Executor
V&V Result
Concrete Scenario
Execution traces
System-level Simulator
ML components (planner, diagnoser, parameter estimator, ...)

Fig. 1. V&V framework.

environment state. Thus, the properties that are evaluated on such traces can compare the internal representation of the autonomous system with the ground truth provided by the simulator.

4.2 Contract-based extension

We extend the VIVAS framework with contract-based design of an autonomous system. We start from an OCRA specification of the system, as depicted in the left part of Fig. 1. The autonomous system (AS) is decomposed into an environment (Env) and an agent (Agent) component. The environment represents the physical world, including external and internal parts of the autonomous system (e.g., road structure, autonomous car, and other cars) and their dynamics. It models the evolution of the actual physical system states, henceforth called the ground truth, in response to control signals it receives from the agent. The agent receives the ground truth data from the environment (e.g., positions of other cars) and provides commands (e.g., steering and acceleration data).

The agent is in turn decomposed into a perception and a control component. The interfaces of such components are detailed based on the specific application and depend on the actual internal representation that the control has of the environment (e.g., position of the objects). They can also be further decomposed based on the internal structure of the agent.

Specific requirements of the autonomous system are formalized as LTL properties and structured into contracts. Each component in the hierarchy is associated with a set of contracts (depicted in green), specifying the acceptable behaviors for the component and its environment. Both the environment and the agent have a high-level description of their behavior, modeled as transition systems in SMV (depicted in purple), which can be composed to generate the system's behavior. This is verified compositionally against the top-level contracts. Given the above contract-based design, the verification is enhanced with the following checks:

- The contract refinement is proved correct for each decomposition level.
- The SMV implementations of Env and Agent are proved correct with respect to the local contracts.
- The contracts are checked at runtime on the finite simulation traces.

The verification of the last point can fail for various reasons, leading to the changes in the contract specification or to actual bugs in the Agent code as discussed in the following section.

4.3 Analyses of monitoring failures

We first focus on the Env component. If one or more Env contracts fail regardless of the system level contracts result, either of the following reasons would apply:

- e.1: the contracts are too strong and a realistic behavior in the simulation violates them; thus, we need to weaken them to allow such behavior.
- e.2: the contracts are violated by a behavior that we indeed discarded with the contracts because we excluded that from that verification (e.g., dangerous situations created by other agents); in this case, the result is accepted.
- e.3: the contracts are violated because of some constants used in the model of the dynamics; in this case, the model must be fixed adjusting such constants.

Focusing on the Autonomous system, agent and other subcomponents, the following outcomes are possible for each decomposition level:

- a.1: the parent contract is satisfied, but a child contract violated; in this case, we may have different causes:
  - a.1.1: the child component is indeed buggy, but the fault did not lead to the failure of parent component; we report this as a problem.
  - a.1.2: the contract of the child components are too strong and the behavior seen in the simulation could be actually accepted; thus, the contract must be weakened to allow such behavior.
  - a.1.3: the contracts are violated because of some constants used in the contracts (e.g., to specify safety margins); in this case, the model must be fixed adjusting such constants.
- a.2: the parent contract is violated as well as one or more child contracts; in this case, we identified the faulty components, and we report it as a problem.
- a.3: the parent contract is violated and all child contracts are satisfied; this must not happen, else there is a mistake in the framework, in the symbolic model (e.g., some connections), or in the concretization.
- a.4: finally, orthogonally to previous points, if a leaf component is failing in several simulation traces, we may want to decompose further to have a better localization of the fault; in this case, we may analyze the Agent's code to identify subcomponents and their contracts.

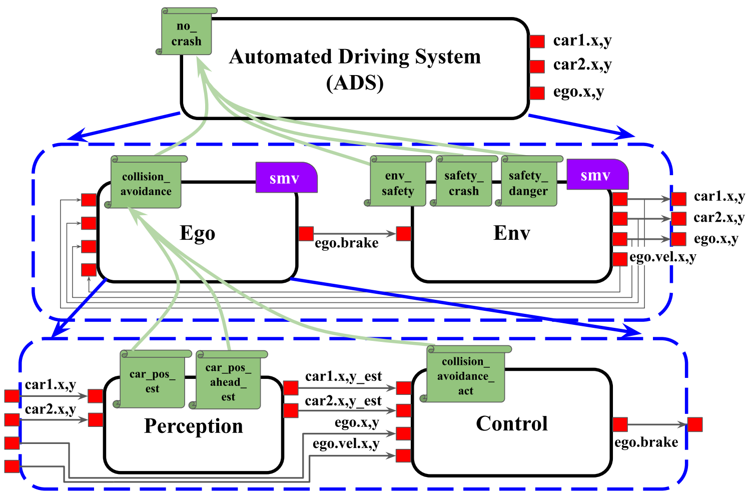

Fig. 2. Contract-based system architecture of Automated Driving System.

5 Automated Driving Use Case

In this section, we present a case study that applies our compositional approach, outlined in the previous section, to an automated driving system (ADS) using the CARLA simulation platform.

5.1 Contract based design

We use OCRA [6], an extension of nuXmv for contract-based specification and reasoning, to partition our symbolic model into components equipped with assume-guarantee contracts. Fig. 2 shows the decomposition of ADS into different components and subcomponents. These components communicate through input and output ports, depicted in red. For simplicity, we have combined the x and y coordinate ports into a single port. In the following, we describe such decomposition, providing the main details of the contracts for each component. For the contract specifications, here we only show the guarantees provided by the contracts. All assumptions are considered True unless stated otherwise.

ADS. The top level system component has an interface with no inputs and six output ports, which are mapped from its Env component. These ports provide the x, y (longitudinal and lateral) positions of the two non-ego vehicles and an ego at the ground truth level, with all vehicles represented point-wise. The top-level system contract `no_crash` (Eq. (1)) guarantees the absence of crashes, and is further refined by the contracts of Env and Ego (`Env.env_safety`, `Env.safety_danger`, `Env.safety_crash`, `Ego.collision_avoidance`). The predicate `crash` (Eq. (2)) is True only when the ego vehicle collides with any non-ego vehicle in front.

$$\texttt{no\_crash} : \mathbf{G}(\neg crash); \tag{1}$$

$$\begin{aligned} crash := & \bigvee_{i=1,2} (car[i].x > ego.x) \wedge (car[i].y = ego.y) \\ & \wedge (ego.velocity.x > 0) \wedge (car[i].y = \mathbf{X}(car[i].y)) \\ & \wedge (ego.y = \mathbf{X}(ego.y)) \wedge (\mathbf{X}(car[i].x \leq ego.x \wedge car[i].y = ego.y)) \end{aligned} \tag{2}$$

Ego. This component models the key properties for the autonomous vehicle to navigate safely in the environment. It takes as inputs from Env the current positions of non-ego and ego vehicles along with ego's speed, and produces a boolean brake command as output. The `collision_avoidance` contract (Eq. (3)) guarantees the brake application whenever ego is in a dangerous condition (Eq. (4)).

$$\texttt{collision\_avoidance} : \mathbf{G}(danger\_gt \rightarrow ego.brake); \tag{3}$$

$$\begin{aligned} danger\_gt := & ((car\_ahead\_gt) \wedge (ego.velocity.x > 0) \wedge \\ & (d_{min} \leq (braking\_distance + danger\_threshold))) \end{aligned} \tag{4}$$

$$car\_ahead\_gt := \bigvee_{i=1,2} (car[i].x > ego.x \ \wedge \ car[i].y = ego.y) \tag{5}$$

$$braking\_distance := (ego.velocity.x)^2 / (2 * Max\_Braking) \tag{6}$$

where d_{min} is the distance to the closest car ahead of ego in its lane, *braking_distance* is the distance covered by ego while braking with Max_Braking[3] from its current position, computed using Newtonian physics in Eq. (6). The Ego is further decomposed into perception and control components, and its contract is therefore refined by the corresponding contracts of these subcomponents.

Perception. The perception component represents the ego's knowledge of the environment. It takes as input the ground truth positions of the non-ego vehicles, and provides as output their estimated positions. It has two contracts, `car_pos_est` and `car_pos_ahead_est` in Eq. (7) and (8) respectively, providing bounds on the estimate errors.[4]

$$\texttt{car\_pos\_est} : \ \mathbf{G}(\bigwedge_{i=1,2} (car[i].x - car[i].x\_est \leq \varepsilon) \wedge (car[i].y = car[i].y\_est)) \tag{7}$$

$$\texttt{car\_pos\_ahead\_est} : \ \mathbf{G}(\bigwedge_{i=1,2} ((car[i].x \geq ego.x) \rightarrow (car[i].x\_est \geq ego.x))) \tag{8}$$

[3] Max_Braking $= 4.6\ m/s^2$, averaged (eyeballed) from literature.

[4] the ε parameter in (7) is the estimation error (in meters), which is set to 2 in the actual implementation when using InterFuser.

Control. The control component takes as input the current estimated positions of the non-ego vehicles from the perception, and the current position and speed of ego vehicle from the environment at the ground truth level. Its contract (Eq. (9)) guarantees the application of brake whenever a dangerous situation is detected.

$$\texttt{collision\_avoidance\_act} : \mathbf{G}(danger\_est \rightarrow ego.brake) \tag{9}$$

$$\begin{aligned} danger\_est :=&(car\_ahead\_est \wedge (ego.velocity.x > 0) \wedge \\ &(d_{min}\_est \leq (braking\_distance + danger\_threshold))) \end{aligned} \tag{10}$$

$$car\_ahead\_est \ := \bigvee_{i=1,2} (car[i].x\_est > ego.x \ \wedge \ car[i].y\_est = ego.y) \tag{11}$$

where $d_{min}\_est = min_{i=1,2}\{(car[i].x\_est - ego.x - \varepsilon)\}$.

Env. This component represents the physical world, providing as output the ground truth positions and speeds of all the vehicles in the environment. The component has one input port, *ego.brake*, which is an output of the Ego component. We define three contracts for Env. The first, `env_safety` (Eq. (12)), ensures that if the ego starts in a safe condition and applies the brake, it cannot reach an unsafe condition:

$$\begin{aligned} \texttt{env\_safety} : &\ safe\_condition \wedge \\ &\mathbf{G}(((safe\_condition) \wedge (ego.brake > 0)) \rightarrow \mathbf{X}(safe\_condition)) \end{aligned} \tag{12}$$

$$safe\_condition := \ (car\_ahead\_gt \rightarrow (d_{min} > braking\_distance)) \tag{13}$$

Fig. 3 shows the difference between safe_condition, danger_gt and braking_distance. In Ego's contract reasoning, $ego.brake = \top$ when $danger\_gt = \top$. Due to the presence of *danger_treshold* (set to 1m), the model checker produces a counterexample of the system level contract in which there is a transition from $safe\_condition \wedge \neg danger\_gt$ to $\neg safe\_condition$. To avoid this situation, we formulate another contract for the Env, `safety_danger` in Eq. (14) which guarantees that the above situation does not occur. The last Env contract, `safety_crash` in Eq. (15), connects *safe_condition* with the *crash* predicate, guaranteeing that if the ego is in a safe condition, it is not involved in a crash.

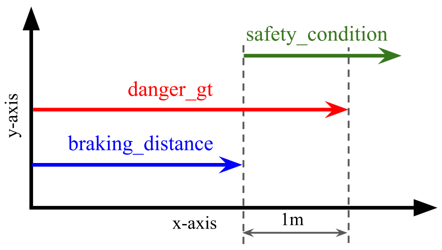

Fig. 3. Safe and dangerous conditions.

$$\texttt{safety\_danger} : \mathbf{G}((safe\_condition \wedge \neg danger\_gt) \rightarrow \mathbf{X}(env\_safety)) \tag{14}$$

$$\texttt{safety\_crash} : \mathbf{G}(safe\_condition \rightarrow \neg crash) \tag{15}$$

5.2 Implementation in SMV

We specify the abstract model of the ADS as a synchronous symbolic transition system, written in the language of the nuXmv [3] model checker. The system consists of two key components: the environment (Env) and the ego vehicle (Ego), as shown in Fig. 2. For abstract scenario generation, the Env implementation consists of 3 vehicles (one "ego" car, representing the autonomous agent under test, and two other cars) moving on a highway with 3 lanes, in which all the vehicles drive in the same direction. The two "non-ego" cars can move freely on the highway, with arbitrary accelerations, braking, and lane change maneuvers (subject to physical constraints about min/max acceleration rates and speed limits), but are not allowed to crash into each other or the ego. The Ego, on the other hand, always drives in the middle lane, aiming to maintain a set cruise speed. It brakes when necessary to avoid collisions with other vehicles and may accelerate when needed to reach the target speed. We are interested in generating test cases to check whether the ego can reach a given destination without crashing into other vehicles.

We use a discrete model of time, where each transition of the system corresponds to a time-lapse of 1 second. A shorter time step would improve the precision of generated abstract scenarios, however, at the cost of computational overhead. We found the choice of 1 second time step to be adequate enough for an effective concretization of the abstract scenarios on the real system. We use the theory of real arithmetic to encode the transition relation of the system, using mostly linear constraints to compute the updates to the speed and locations of the vehicles (thanks to the discretization of time). The concrete implementation of the AI-based vehicle under test in CARLA, instead, is based on the InterFuser agent [25], taken from CARLA autonomous driving leaderboard database [2].

5.3 Concretizing the contracts on the real system

When checking the contracts defined above on the simulation traces produced by CARLA, we must take into account the fact that the symbolic model provides an abstract view of the system, in which several details have been omitted. From the point of view of contract satisfaction, one key difference between our symbolic model and the ground truth is that CARLA gives as output the positions and dimensions of the bounding boxes of all the agents in the environment at every time instant in x-y coordinate plane (the vehicle navigation plane), with the ego always centered at the origin. In contrast, our symbolic model represents the agents' positions as single points in an absolute coordinate system (with the origin at the beginning of the road). If we compare only the center of estimated bounding boxes with the ground truth ones, the contract satisfaction may be dubious in some cases, e.g. when one or more vertices of the car are in ego's lane and the center lies in another lane.

In the following, we describe how the high-level contracts are mapped to those verified on concrete executions. First, the crash is evaluated by using the crash sensor mounted on the ego vehicle in CARLA for detecting collisions, considering only the instances where the ego vehicle collides with a non-ego vehicle in front. We opt for this method over defining collisions using ground-truth bounding boxes, as the latter caused many false positives due to simulator inaccuracies. Furthermore, we modify the definitions of car_ahead_gt (Eq. (5)), car_ahead_est (Eq. (11)) and the variables d_{min} and d_{min}_est introduced in §5.1 to account for the change of coordinates and the use of bounding boxes.

Closest distance. The center of the ego vehicle is taken at origin of the 2D plane, with ego being oriented longitudinally towards positive x-axis. Let $v_i(j) = (x_{i,j}, y_{i,j})$ denote the jth bounding-box vertex of the ith non-ego vehicle; the closest distance d_{min} of the ego vehicle from the non-ego vehicles is defined as:

$$(-lane\_width/2 \leq y_{i,j} \leq lane\_width/2) \land (x_{i,j} \geq 0) \tag{16}$$

$$d_{min} = min\{x_{i,j} \mid (x_{i,j}, y_{i,j}) \texttt{ satisfies } (16)\} \tag{17}$$

Car ahead of ego. Let nc be the nearest car to ego, defined by:

$$\begin{aligned} \forall i (i = nc) \implies ((\exists j (x_{i,j}, y_{i,j}) \texttt{ satisfies } (16) \land \\ \forall i, j' \ (x_{i,j'}, y_{i,j'}) \texttt{ satisfies } (16)) \implies x_{i,j} \leq x_{i,j'}) \end{aligned} \tag{18}$$

then the distance to the closest vertex is given by $d_{v_{min}} = d(nc)$, where the distance $d(i)$ for the i-th car is defined as:

$$\begin{aligned} \forall i, j \ . \ x_{i,j} = d(i) \implies \exists j' (x_{i,j'}, y_{i,j'}) \texttt{ satisfies } (16) \land \\ x_{i,j} \geq 0 \ \land \forall j'' \ (x_{i,j''} \geq 0 \rightarrow x_{i,j} \leq x_{i,j''}) \end{aligned} \tag{19}$$

We then define the predicate car_ahead_gt as follows:

$$(0 \leq d_{v_{min}} \leq \delta) \leftrightarrow car\_ahead\_gt \tag{20}$$

where δ is the look-ahead distance of the AI-based perception module, beyond which it does not detect anything in the environment. [5]

[5] For InterFuser, this distance is 20 meters, so $\delta = 20$.

Closest estimated distance. The definition of d_{min}_est is similar to the one of d_{min} (Eq. (17)), except that we use the coordinates of the estimated bounding boxes $(x_{est_{i,j}}, y_{est_{i,j}})$ and take into account the estimation error ε:

$$(-lane\_width/2 \leq y_{est_{i,j}} \leq lane\_width/2) \wedge (0 \leq x_{est_{i,j}} \leq d_{v_{min}} + \varepsilon) \tag{21}$$

$$d_{min}\_est = min\{x_{est_{i,j}} \mid (x_{est_{i,j}}, y_{est_{i,j}}) \texttt{ satisfies } (21)\} \tag{22}$$

Car estimate ahead of ego. For all the bounding boxes estimated by the perception module, if there exists a vertex satisfying Eq. (21), while $0 \leq d_{v_{min}} \leq \delta$, then the predicate $car\_ahead\_est = \top$.

6 Experimental Evaluation

In this section, we evaluate our methodology on the ADS application, leveraging the VIVAS framework within the CARLA simulator, as detailed in [15]. We have expanded this framework with contract models, refinement checks, and monitoring, as discussed earlier. Contract refinement is checked using OCRA, employing nuXmv for LTL model checking. Refinement checks for the specified contracts and implementations of the Env and Ego components are conducted using the ic3 algorithm, taking only a few seconds. For abstract scenario generation, we use Bounded Model Checking (BMC) algorithm of the nuXmv [3] symbolic model checker. Due to the presence of non-linear constraints in our symbolic model, specifically in computing ego's braking distance, we leverage the z3 SMT solver [22] for checking the satisfiability of the BMC queries.

Experimentation was carried out on an Intel i7 with NVIDIA GeForce RTX 2080 8GB GPU, meeting the minimum requirements for running CARLA simulations with AI models. All experiments required approximately 22 hours to complete. Model checking completed in under 25 seconds on average per abstract scenario instance, with a timeout of 200 seconds being never reached during our experiments. However, the main performance bottleneck arose from concrete scenario instantiation in CARLA and subsequent simulations, averaging 4.5 minutes per scenario. The code and data necessary for reproducing our experiments are available at `https://es-static.fbk.eu/people/sgoyal/sefm24`.

6.1 Scenario generation

In order to enumerate abstract scenarios, we followed the same specification defined in [15], which we report here for completeness. We define for each non-ego car a set of predicates specifying its position relative to the ego, in terms of occupation of cells of an abstract 3x3 "grid" centered on the ego. We then define an abstract scenario as a combination of constraints about the different positions of the non-ego cars on the grid at different points in time (ref. Fig. 3

in [15]). More specifically, each abstract scenario is specified as an LTL property of the following form:

$$\begin{aligned}\neg\mathbf{F}(&\mathrm{car1\_grid\_pos} = \mathrm{CELL\_A1} \wedge \mathrm{car2\_grid\_pos} = \mathrm{CELL\_A2} \wedge \\ &\mathbf{X}(\mathbf{F}(\mathrm{car1\_grid\_pos} = \mathrm{CELL\_B1} \wedge \mathrm{car2\_grid\_pos} = \mathrm{CELL\_B2})))\end{aligned} \tag{23}$$

where car$i$_grid_pos encodes the position of the i-th non-ego car in the grid corresponding its cell number. We define our coverage criterion by selecting a subset of abstract scenarios of interest, consisting of various combinations of the traffic situations that can be modeled by positioning the non-ego cars in the grid around the ego. In total, for the experiments, we defined 144 such scenarios, concretizing them on two weather conditions.

In order to generate a concrete traffic scenario for the simulator, the abstract counterexample trace generated by the model checker is parsed for relevant information to be fed as input to the simulator. As an interface to the CARLA simulator, we used the CARLA module ScenarioRunner. It takes as input the route that needs to be followed by AI-based ego and a traffic scenario composed of non-egos' behaviors. Every state of the abstract scenario trace is concretized into the corresponding behavior of every non-ego agent. We do not need to extract ego's behavior from the abstract trace, since it is expected to make decisions autonomously in the simulator. Only its initial spawn position and destination need to be extracted to follow the route. For more details on the experimental setup for the scenario generation, please refer Sec. 5.1 of [15].

6.2 Evaluating contracts at runtime

The number of contract violations are reported in Table 1, column "Contract Violations". In total, we evaluate 288 simulation traces. In case of a crash, the results are saved only up to its first occurrence while rest of the trace is discarded. We will refer to the methodology in §4.3 to analyze the results from here on.

ADS. The system level contract, i.e., `no_crash` is violated 62 times. In 3 of these cases, ego's and all its subcomponents' contracts are true. These cases belong to the analyses class *a.3*, implying a mistake in our framework. These scenarios involve a non-ego unexpectedly rear-ending the ego while attempting to change lanes, then dragging ego forward from the side. This leads to $car\_ahead\_gt = \bot$ since none of the non-ego's vertices are in front of ego, yet its center moves ahead of ego (which is not aligned with the driving direction), triggering crash detection. The issue stems from an incorrect concretization of the crash condition in the simulator, unaccounted for in our symbolic framework. We should only consider the cases where the ego crashes from the front without any prior collisions.

Env. All the Env's contracts are violated in multiple executions. Since the Env contracts guarantee the physical laws, we need further analyses: all the violations

Table 1. Analysis of contract violations.

Composite component	**Sub component**	**Contracts**	**Contract count** ($\perp$)	**Crash Sequence** ($\perp$)
ADS		no-crash	62	62
ADS	Env	env_safety	10	0
		safety_danger	146	0
		safety_crash	2	0
	Ego	collision_avoidance	151	57
ADS.Ego	Perception	(7)$\wedge$(8)	169	58
	Control	collision_avoidance_act	127	52

Sequence leading to Crash (Total crashes = 62)			
Analysis class (§4.3)	**Perception**	**Control**	**Perception $\wedge$ Control**
a.1 ($Ego \wedge \neg Child$)	2	0	0
a.2 ($\neg Ego \wedge \neg Child$)	56	52	51
a.3 ($\neg Ego \wedge \bigwedge Child$)	-	-	0
Avoidable Crashes			1

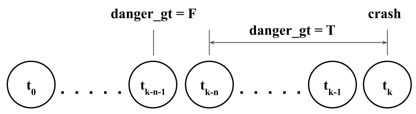

Fig. 4. Sequence leading to a crash.

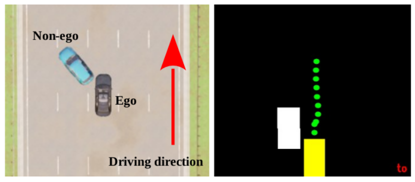

Fig. 5. Avoidable crash; (left): Bird's-eye view in CARLA; (right) AI-based ego's perception component's output.

belong to the class *e.2*, where, (a) non-egos cut in front of ego in unsafe conditions, violating `env_safety` and `safety_danger`, (b) they first hit ego from behind, violating `safety_crash`. One could add the predicate, *car_ahead_gt* in the assumption of former two Env contracts (following *e.1*), however, that will weaken the environment guarantees. We leave the contracts of the environment as they are (without the *car_ahead_gt* assumption), as they require non-ego cars to not cut in violating the safety conditions; this can happen in the ground truth and the monitoring tells us if we are in these violations.

Each simulation trace is at least 30 seconds long. As our formulations necessitate contract satisfaction throughout execution (using the "Globally" LTL operator), one single violation would signify a contract violation. Given that a scenario encompasses various traffic configurations around the ego vehicle, monitoring child components across the entire simulation trace becomes impractical, especially in crash scenarios. Our focus lies in identifying the specific situations leading to a crash and attributing failure to the responsible component(s).

Evaluating contracts violated in the sequence leading up to crash. Many component violations are not linked to crash situations. In order to better localize the cause of crashes, we consider the contract violations only during the last sequence of time instances before the crash where $danger\_gt = \top$ consecutively. To be more precise, we consider the conditions inside the **G** operator of contracts and check their violation on the time sequence from t_{k-n} to t_k, where t_k is the crash state and t_{k-n} is the first danger state in the sequence (see Fig. 4). The results are reported in column "Crash Sequence" of Table 1. None of the Env contracts are violated during this sequence, implying that our assumptions on the environment are correct. Note that two global contract violations of `safety_crash` are not taken in account since $danger\_gt = \bot$ in those cases.

Ego. We further analyze contract violations of Ego and its subcomponents according to our classes, shown in Table 1. Class *a.1*: two cases of perception failure, not resulting in ego's violation; *a.2*: pinpoints the bug in both the *perception* and *control* subcomponents; *a.3*: implies no fault in the framework.

Avoidable Crashes. If $safe\_condition = \top$ (Eq. (13)) at any time instant in any sequence leading to a crash, ego vehicle has the possibility to brake and avoid the crash. Out of 62 crashes, we found 1 case where ego had this possibility. Fig. 5 shows snapshot of crash from Carla simulator, with wrongly estimated non-ego's location and safe trajectory by InterFuser on the right.

6.3 Updating refinements

In Table 1, the control contract is violated in the majority of situations, though not at the same time instants as the perception's violations, since danger situation is a conjunction of the correct detection of non-egos in front and their closest distance from the ego. We need to further analyze such erratic behavior of the controller, since it bases its decision on the AI module's state estimations.

Further analysis of the AI module revealed its role in predicting the ego's next 10 waypoints based on a coarse input route, estimating a collision-free distance (called "safe distance" by its designers) for the ego to drive forward. This distance guides the ego's collision avoidance maneuvers. However, the AI may predict a collision-free trajectory in situations where, for example, a non-ego is between two lanes. Despite this conservative approach, we argue that an autonomous vehicle should still apply the brakes in such scenarios. While violations of the Ego's contract may not always result in a system-level crash, if they do, we can identify the failure source. To formalize this, we decompose the ego's control component into planner and decision subcomponents (Fig. 6). Therefore, the contract `collision_avoidance_act` is further refined by the contracts of planner and decision (`Planner.plan` and `Decision.brake`, respectively).

Planner: The role of the planner is to mimic the black-box of AI-based ego's module which computes the "safe distance", d_{min}_AI. At the symbolic level, this

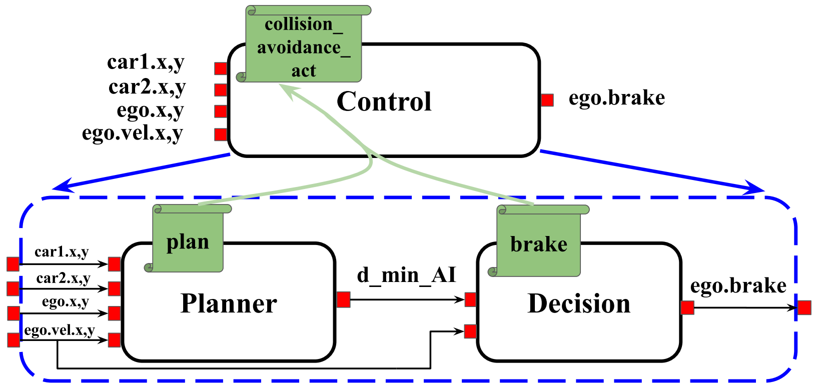

Fig. 6. Refinement of Control component.

distance is computed by the model checker in such a way that its value satisfies the contract `plan` in Eq. (25), always guaranteeing the detection of a dangerous situation, *danger_AI*, if $danger\_est = \top$.

$$\begin{aligned} danger\_AI &:= d_{min}\_AI <= (braking\_distance + 1) \\ danger\_est &:= ((car\_ahead\_est) \wedge (ego.velocity.x > 0) \wedge \\ &\quad (d_{min}\_est <= (braking\_distance + 1))) \end{aligned} \tag{24}$$

$$\texttt{plan} : \mathbf{G}(danger\_est \rightarrow danger\_AI) \tag{25}$$

$$\texttt{brake} : \mathbf{G}(danger\_AI \rightarrow ego.brake) \tag{26}$$

Decision: Input to this component is d_{min}_AI computed by the planner component. The output is *ego.brake* which is subsequently propagated upstream to its composite component, control, and then further up to the level of Ego and Env. This contract (Eq. (26)) always guarantees that the braking occurs whenever $danger\_AI = \top$.

6.4 Pinpointing Failure

Table 2 shows the violations of control contract's refinements, across all executions and crash sequences. Analyzing these based on our classification, *a.1*: no violations (empty set); *a.2*: pinpoints the source of failure to (a) the decision component, which failed to apply brakes despite the AI module detecting a dangerous situation, and (b) the planner component, implying incorrect computation of d_{min}_AI by the AI module, despite correctly estimating the objects ahead in the ego vehicle's lane. These errors occur when the lateral positions and/or orientations of non-egos are not estimated correctly, particularly when only part of the non-ego vehicle is within the ego's lane.

Table 2. Analysis of Control contract violations.

Composite component	**Sub component**	**Contracts**	**Contract count($\perp$)**	**Crash Sequence($\perp$)**
ADS.Ego.Control	Planner	plan	123	50
	Decision	brake	105	45

Sequence leading to Crash (Total crashes = 62)			
Analysis class (§4.3)	**Planner**	**Decision**	**Planner $\wedge$ Decision**
a.1: ($Control \wedge \neg Child$)	0	0	0
a.2: ($\neg Control \wedge \neg Child$)	49	44	41
a.3: ($\neg Control \wedge \bigwedge Child$)	-	-	0

7 Conclusions and Future Work

In this paper, we have proposed an integration of contract-based design with test case generation for the system-level simulation-based verification of automated driving systems. System-level properties of the automated cars are decomposed into contracts of the ego components and assumptions on the environment. We introduced a methodology that leverages formal methods for contract reasoning to monitor the properties on the simulation executions to better diagnose issues in the autonomous system, and iteratively refine the contracts specification. Our experiments, conducted using the CARLA simulator and off-the-shelf AI agents, have provided empirical evidence of the effectiveness of contract-based refinement and runtime monitoring in enhancing the verification of ADS components.

There are several directions for future research and development. For example, we will explore the scalability and applicability of our approach to more complex ADS architectures and scenarios. Moreover, we will explore ways to incorporate uncertainty quantification methods into the contracts specification.

References

1. Astorga, A., Hsieh, C., Madhusudan, P., Mitra, S.: Perception Contracts for Safety of ML-Enabled Systems. Proc. ACM Program. Lang. **7**(OOPSLA2) (oct 2023). https://doi.org/10.1145/3622875
2. CARLA Team: CARLA Autonomous Driving Leaderboard. https://leaderboard.carla.org/leaderboard/, accessed: 2023-08-30
3. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: CAV. LNCS, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
4. Chen, D., Krahenbuhl, P.: Learning from All Vehicles. In: CVPR. pp. 17201–17210. IEEE (jun 2022). https://doi.org/10.1109/CVPR52688.2022.01671
5. Cimatti, A., Tonetta, S.: Contracts-refinement Proof System for Component-based Embedded Systems. Science of Computer Programming **97**, 333–348 (Jan 2015). https://doi.org/10.1016/j.scico.2014.06.011

6. Cimatti, A., Dorigatti, M., Tonetta, S.: OCRA: A Tool for Checking the Refinement of Temporal Contracts. In: ASE. vol. 4144, pp. 702–705. IEEE/ACM (Nov 2013). https://doi.org/10.1109/ase.2013.6693137
7. Cimatti, A., Tonetta, S.: A Property-Based Proof System for Contract-Based Design. In: 2012 38th Euromicro Conference on Software Engineering and Advanced Applications. pp. 21–28. IEEE (Sep 2012). https://doi.org/10.1109/seaa.2012.68
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge, MA, USA (2000)
9. DeCastro, J.A., Liebenwein, L., Vasile, C.I., Tedrake, R., Karaman, S., Rus, D.: Counterexample-Guided Safety Contracts for Autonomous Driving. In: Algorithmic Foundations of Robotics XIII. p. 939–955. Springer (2020). https://doi.org/10.1007/978-3-030-44051-0_54
10. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: An Open Urban Driving Simulator. In: Levine, S., Vanhoucke, V., Goldberg, K. (eds.) Proceedings of the 1st Annual Conference on Robot Learning. PMLR, vol. 78, pp. 1–16 (13–15 Nov 2017). https://doi.org/10.48550/arXiv.1711.03938
11. Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., Seshia, S.A.: VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In: Dillig, I., Tasiran, S. (eds.) CAV. pp. 432–442. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_25
12. Fratini, S., Fleith, P., Policella, N., Griggio, A., Tonetta, S., Goyal, S., Le, T.T.H., Kimblad, J., Tian, C., Kapellos, K., et al.: Verification and Validation of Autonomous Systems with Embedded AI: The VIVAS Approach. In: ASTRA. ESA (2023), https://az659834.vo.msecnd.net/eventsairwesteuprod/production-atpi-public/070740b67e5b4a32a9be94228c9ac40d
13. Fremont, D.J., Kim, E., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: A Language for Scenario Specification and Data Generation. Machine Learning **112**(10), 3805–3849 (feb 2022). https://doi.org/10.1007/s10994-021-06120-5
14. Ghosh, S., Sadigh, D., Nuzzo, P., Raman, V., Donzé, A., Sangiovanni-Vincentelli, A.L., Sastry, S.S., Seshia, S.A.: Diagnosis and repair for synthesis from signal temporal logic specifications. In: HSCC. pp. 31–40. HSCC'16, ACM (Apr 2016). https://doi.org/10.1145/2883817.2883847
15. Goyal, S., Griggio, A., Kimblad, J., Tonetta, S.: Automatic Generation of Scenarios for System-level Simulation-based Verification of Autonomous Driving Systems. In: FMAS@iFM. EPTCS, vol. 395, pp. 113–129 (2023). https://doi.org/10.4204/EPTCS.395.8
16. Ivanov, R., Jothimurugan, K., Hsu, S., Vaidya, S., Alur, R., Bastani, O.: Compositional Learning and Verification of Neural Network Controllers. ACM Trans. Embed. Comput. Syst. **20**(5s), 1–26 (sep 2021). https://doi.org/10.1145/3477023
17. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
18. Liu, S., Saoud, A., Jagtap, P., Dimarogonas, D.V., Zamani, M.: Compositional Synthesis of Signal Temporal Logic Tasks via Assume-Guarantee Contracts. In: CDC. IEEE (dec 2022). https://doi.org/10.1109/cdc51059.2022.9992715
19. Majumdar, R., Mathur, A., Pirron, M., Stegner, L., Zufferey, D.: Paracosm: A test framework for autonomous driving simulations. In: Guerra, E., Stoelinga, M. (eds.) FASE. Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_9

20. Mallozzi, P., Incer, I., Nuzzo, P., Sangiovanni-Vincentelli, A.: Contract-based Specification Refinement and Repair for Mission Planning. In: FormaliSE. vol. 45, pp. 29–38. IEEE (May 2023). https://doi.org/10.1109/formalise58978.2023.00011
21. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag New York, Inc., New York, NY, USA (1992)
22. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
23. Păsăreanu, C.S., Mangal, R., Gopinath, D., Getir Yaman, S., Imrie, C., Calinescu, R., Yu, H.: Closed-Loop Analysis of Vision-Based Autonomous Systems: A Case Study. In: Enea, C., Lal, A. (eds.) CAV. pp. 289–303. LNCS, Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37706-8_15
24. Phan-Minh, T.: Contract-Based Design: Theories and Applications. Ph.D. thesis, California Institute of Technology (2021)
25. Shao, H., Wang, L., Chen, R., Li, H., Liu, Y.: Safety-Enhanced Autonomous Driving Using Interpretable Sensor Fusion Transformer. In: Liu, K., Kulic, D., Ichnowski, J. (eds.) Proceedings of The 6th Conference on Robot Learning. PMLR, vol. 205, pp. 726–737 (14–18 Dec 2023). https://doi.org/10.48550/arXiv.2207.14024
26. Sharf, M., Besselink, B., Molin, A., Zhao, Q., Henrik Johansson, K.: Assume/Guarantee Contracts for Dynamical Systems: Theory and Computational Tools. IFAC-PapersOnLine **54**(5), 25–30 (2021). https://doi.org/10.1016/j.ifacol.2021.08.469, 7th IFAC Conference on Analysis and Design of Hybrid Systems ADHS 2021
27. Vin, E., Kashiwa, S., Rhea, M., Fremont, D.J., Kim, E., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: 3D Environment Modeling for Falsification and Beyond with Scenic 3.0. In: Enea, C., Lal, A. (eds.) CAV. pp. 253–265. LNCS, Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37706-8_13
28. Wu, P., Jia, X., Chen, L., Yan, J., Li, H., Qiao, Y.: Trajectory-guided Control Prediction for End-to-end Autonomous Driving: A Simple yet Strong Baseline. In: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (eds.) Advances in NeurIPS. vol. 35, pp. 6119–6132. Curran Associates, Inc. (2022). https://doi.org/10.48550/arXiv.2206.0812