Test Specification and Generation for Connected and Autonomous Vehicle in Virtual Environments

BAEKGYU KIM, TAKATO MASUDA, and SHINICHI SHIRAISHI,
Toyota InfoTechnology Center, U.S.A.

The trend of connected/autonomous features adds significant complexity to the traditional automotive systems to improve driving safety and comfort. Engineers are facing significant challenges in designing test environments that are more complex than ever. We propose a test framework that allows one to automatically generate various virtual road environments from the path and behavior specifications. The path specification intends to characterize geometric paths that an environmental object (e.g., a roadway or a pedestrian) needs to be visualized or move over. We characterize this aspect in the form of constraints of 3-Dimensional (3D) coordinates. Then, we introduce a test coverage, called an area coverage, to quantify the quality of the generated paths in terms of how diverse of an area the generated paths can cover. We propose an algorithm that automatically generates such paths using an SMT (Satisfiability Modulo Theories) solver. However, the behavioral specification intends to characterize how an environmental object changes its mode over time by interacting with other objects (e.g., a pedestrian waits for a signal or starts crossing). We characterize this aspect in the form of timed automata. Then, we introduce a test coverage, called an edge/location coverage, to quantify the quality of the generated mode changes in terms of how many modes or transitions are visited. We propose a method that automatically generates many different mode changes using a model-checking. To demonstrate the test framework, we developed the right-turn pedestrian warning system in intersection scenarios and generated many different types of pedestrian paths and behaviors to analyze the effectiveness of the system.

CCS Concepts: • **Computer systems organization → Embedded software**;

Additional Key Words and Phrases: Test specification, test coverage, test generation, virtual prototyping, autonomous vehicle, connected vehicle, SMT solver, timed automata, system safety

ACM Reference format:
Baekgyu Kim, Takato Masuda, and Shinichi Shiraishi. 2019. Test Specification and Generation for Connected and Autonomous Vehicle in Virtual Environments. *ACM Trans. Cyber-Phys. Syst.* 4, 1, Article 8 (October 2019), 26 pages.
https://doi.org/10.1145/3311954

1 INTRODUCTION

The recent trend of the automotive Cyber-Physical Systems (CPS) adds significant complexity to the traditional automotive system. To improve driving safety and comfort, vehicles are expected

Authors' addresses: B. Kim, T. Masuda, and S. Shiraishi, 465 Bernardo Avenue, Mountain View, CA 94043; emails: baekgyu.kim@toyota.com, {tmasuda, sshiraishi}@us.toyota-itc.com.

2378-962X/2019/10-ART8 $15.00
https://doi.org/10.1145/3311954

to drive autonomously and/or to communicate with each other and infrastructures. Those features include not only triggering warning signals such as lane departure or pedestrian presence in the crosswalk, but also controlling vehicle such as adaptive cruise control or pre-collision system, possibly by communicating with other vehicles or infrastructures (e.g., roadside units). Such complexity makes it harder for engineers to test correctness, performance, or effectiveness of those driving features in the physical environment.

To test new vehicle features, OEMs traditionally have been integrating a software with a physical vehicle platform and then deploying the whole system to the physical environment. The examples of such a physical environment include test tracks constructed by OEMs or public roads where OEMs have permissions to test drive from government authorities. However, the inherent limitation of the physical test makes it harder for OEMs to claim safety of such advanced features in the automotive CPS. First, the physical test does not scale well to build complex road environments where the advanced features need to be tested. Some features need to be tested under diverse geometric designs of roadways (e.g., adaptive cruise control or lane keep assist) and some others need to be tested along with infrastructures that can communicate with vehicles (e.g., a red traffic light warning or an intersection management). It is practically difficult for OEMs to build such physical roadways or to modify existing public roadways to meet their test requirements. Second, it is risky to test some features in the physical environment due to safety concerns. For example, the right-turn collision warning system enables a vehicle entering an intersection to receive a warning signal when pedestrians cross a crosswalk to mitigate a collision hazard. To appropriately test the effectiveness of this system, one needs to consider a test environment where pedestrians walk in many different ways on the crosswalk. However, it is difficult to use real pedestrians in the test for safety concerns. Third, it is difficult to reproduce the test result in the physical environment. After finding issues through a test, another test needs to be followed to validate the correction under the same environment. However, some conditions are inherently difficult to reproduce, such as weather conditions or particular traffic flows.

To handle the test complexity in the automotive CPS, we need an alternative or a complementary way to prove the safety. The virtual test is one way to prove the safety of advanced features with less concerns about the aforementioned issues. The basic idea is that software is integrated with a virtual vehicle platform (e.g., a 3D CAD model) that has vehicle physics close to a real vehicle; then, the virtual vehicle is to be tested in various virtual road environments. Several vehicle simulation tools are available, such as CarSim, PreScan, or Unity3D, to test software in such virtual road environments. Such tools allow one to freely create road environments where a vehicle needs to be tested. However, if those environments are created in an ad hoc fashion, one may miss some tricky scenarios that will expose a critical issue of a system or may repeat unnecessary tests that will reduce the test efficiency. Our research problem consists of how to systematically visualize such road environments in a way that the test result can be interpreted in a more rigorous way.

In our prior work [Kim et al. 2016], we studied how to formalize various curvy roadways and generate them in an automatic fashion for the purpose of testing particular vehicle features such as an adaptive cruise control or a curve warning system. However, we believe that the test generation framework needs to be more generalized to accommodate a broader range of vehicle features. Such generalization requires deeper understanding of the road environment, which consequently adds more complexity to the test generation framework.

To this end, we propose a test generation framework that allows such road environments to be generated automatically from the formal test specification. We first introduce the static objects and dynamic objects that compose the road environment. A static object does not change its characteristic over time once visualized in the environment. The examples of the static object are curvy roadways, intersections, merge/exit points, and bridges. However, a dynamic object changes its

characteristics over time once visualized in the environment. Examples of the dynamic object are vehicles, bicycles, and pedestrians.

Then, we propose a way to formally specify tests in the form of an object path and an object behavior. An object path characterizes the geometric aspects that are of interest to test a feature. For example, an adaptive cruise control needs to be tested under various curvatures of roadways (i.e., a static object) visualized on a certain path [ISO15622 2010]; a pedestrian or a vehicle (i.e., dynamic objects) may move on a certain path of a crosswalk or a lane. We formalize the parametrized SMT constraints to characterize the geometric aspects of an object path. The formula includes unknown 3D coordinates on which a path is visualized and known parameters to appropriately constrain how a path is visualized (e.g., min/max curvatures or distances between two curves). Those 3D coordinates are determined later by the test generation algorithm. Unlike the static objects, the dynamic objects change its behavior by interacting with each other. For example, a pedestrian may start walking in a crosswalk after a signal change (i.e., a behavior change from waiting to walking); a vehicle may slow down or speed up depending on the distance from the forward vehicle or pedestrians (i.e., a behavior change from slowdown to speed-up). We additionally consider such behavioral aspects and formalize them in the form of timed automata [Alur and Dill 1994].

Finally, we propose the test generation algorithm for the static and dynamic objects from the formalized test specification. Our algorithm takes the path and behavior specifications as input, and outputs necessary information to visualize the static and dynamic objects. From the path specification, the algorithm determines a set of 3D coordinates to visualize the corresponding path. From the behavior specification, the algorithm generates a set of runs to visualize a series of interactions with other objects over time. We demonstrate the applicability and empirical result of the test generation framework with a case study of the right-turn pedestrian collision warning system.

We make the following contributions:

- Formalization of the test specification from the categorization of the static and dynamic objects;
- Formalization of the area coverage criteria to quantify the quality of the generated object paths;
- Development of the SMT-based path generation algorithm to automatically determine 3D coordinate sets of paths based on the distance metric;
- Development of the model-checking-based object run generation for dynamic objects based on the edge/location coverage criteria;
- Demonstration of the case study of the right-turn pedestrian collision warning system.

This article is organized as follows: We explain the problem statement and the overview of the approach in Section 2. In Section 3 and Section 4, we explain the path and behavioral specification and generation. We present a case study in Section 5, related work in Section 6, and conclude in Section 7.

2 PROBLEM STATEMENTS AND APPROACH OVERVIEW

2.1 Problem Statements

Visualizing both static and dynamic objects needs to consider their geometrical aspects on 3D coordinate system. Figure 2 illustrates how static objects (e.g., roadways) and dynamic objects (e.g., pedestrians and vehicles) are typically visualized in the simulation tools, such as Carsim, Prescan, or Unity3D. A roadway is visualized from a set of 3D coordinates according to some interpolation methods (e.g., a linear or a spline interpolation). For example, five coordinates are determined in Figure 2(b); an interpolant that crosses the five coordinates is visualized as a line; the interpolant is further textured as a roadway as shown in Figure 2(a).

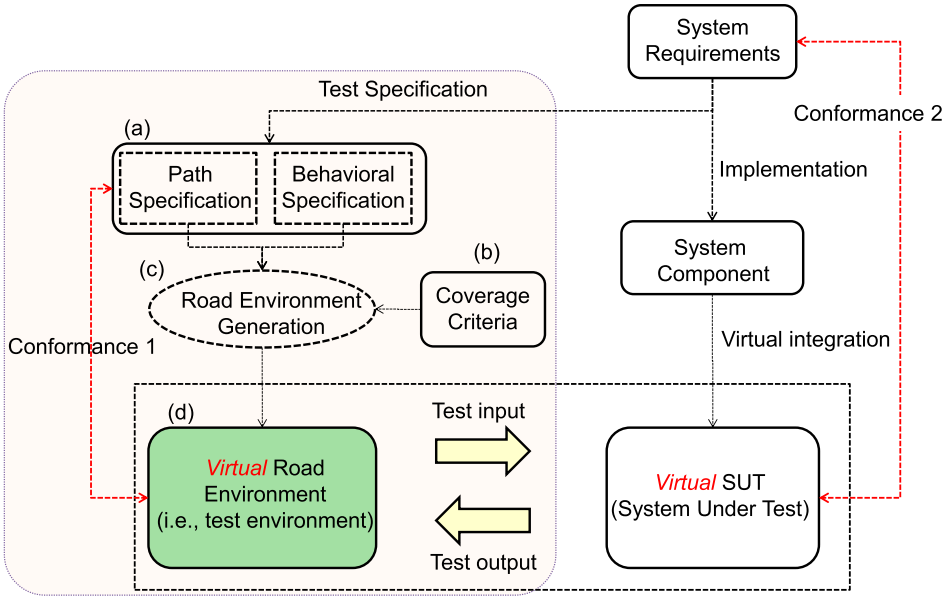

Fig. 1. Illustration of the virtual test: The paper scope is highlighted in the dotted box.

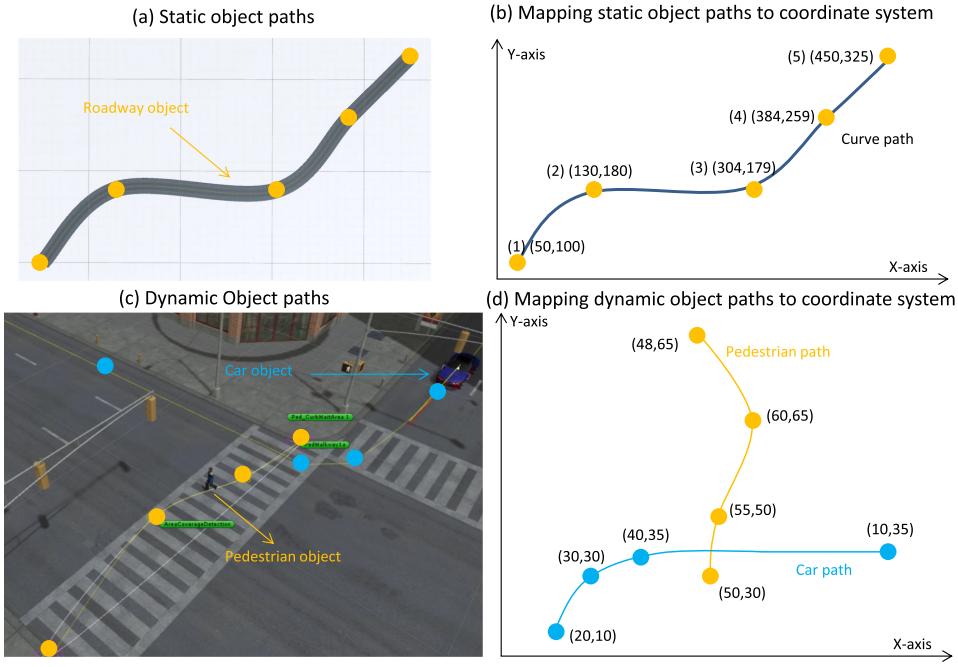

Fig. 2. Illustration of object paths and mapping to the coordinate system.

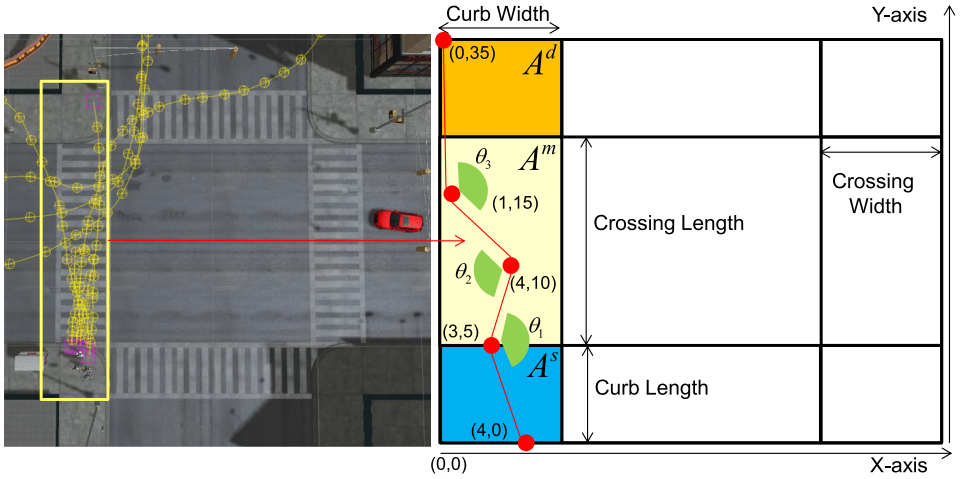

Fig. 3. Illustration of a path specification.

A pedestrian or a vehicle are similarly visualized. Suppose 3D images of a pedestrian and a vehicle are given (e.g., CAD models). In Figure 3(d), the pedestrian path (yellow line) and the vehicle path (blue line) are visualized from their respective sets of 3D coordinates. The 3D images of the pedestrian and the vehicle move over the paths during the simulation. One difference from the static object visualization is that the dynamic objects have their own behavior that changes over time. For example, the pedestrian (the vehicle) may follow the yellow path (the blue path) when the traffic light is green (red), while it may stop (proceed) on the yellow path (the blue path) on the red light (green light). In addition, the dynamic object behavior may include other aspects such as a speed change.

Our observation is that some parts of the visualization process are already implemented and automated in the state-of-the art simulation tools, such as the 3D texturization, the 3D coordinate interpolation, and so on; those tools typically provide users with APIs to execute such processes. However, some information should be manually decided and integrated with such existing processes by engineers, such as 3D coordinates. Based on this observation, we consider a test problem to formally specify and generate the necessary information (but not all the information from scratch), which can then be integrated with the existing mechanisms toward the fully automated visualization.

Consider a vehicle to be tested under various road environments that consist of many different static and dynamic objects. To build such a test environment, engineers manually choose the 3D coordinates for the paths of static or dynamic objects in a way that meets a certain test scenario; in case of dynamic objects, one needs to additionally implement their specific behavior over the paths. When many different test environments need to be created, it is time-consuming to manually choose those 3D coordinates and to implement their behavior. Also, it is difficult to guarantee that the manually visualized environment actually conforms to the test scenario. A better way to construct the test environment would be to provide a framework that allows one to formally specify the tests at a higher level of abstraction and to automatically visualize many different test environments that are guaranteed to conform to the test specification.

The summary of the problem is as follows:

- (Test specification) How to formally define the road environment in terms of the path and the behavior of the static and dynamic objects?

- (Test generation) How to automatically visualize many different road environments from the formal specification?

2.2 Approach Overview

Figure 1 illustrates the proposed test framework; the dotted box at the left side is the focus of this article. Given system requirements, suppose a system component is implemented (e.g., control algorithms or vision algorithms) and integrated with a virtual vehicle (i.e., virtual SUT). The virtual vehicle needs to be tested under various virtual road environments that include static and/or dynamic objects to show it meets the system requirements (i.e., conformance 2 in Figure 1). To automatically generate such road environments, we propose a way to specify the road environments that are of interest to the system requirements (i.e., test specification). We propose two types of test specification: the path specification and the behavioral specification (Figure 1(a)). The path specification intends to characterize geometric paths that static or dynamic object needs to be visualized or follow in the form of constraints that impose a set of 3D coordinates to be located in a particular way. The behavioral specification intends to characterize how a dynamic object changes its mode over time by interacting with other objects in the form of timed automata. Depending on how a test is specified, there may exist infinitely many road environments that conform to the test specification. We present coverage criteria (Figure 1(b)) for each path specification and behavioral specification that allows one to measure if generated road environments are sufficient enough to show the conformance to the system requirements (i.e., conformance 2). Finally, we propose algorithms that can automatically generate road environments that meet the test specification and criteria. The generated road environments are guaranteed to meet the test specification (i.e., conformance 1).

3 PATH SPECIFICATION AND GENERATION

This section explains how to specify paths of a static or a dynamic object in terms of 3D coordinates. Then, we explain a coverage criteria to quantify the quality of a set of paths in terms of the size of the geometric area visited by the paths. Finally, we present an algorithm that automatically generates a set of paths that meet the path specification. We present the empirical result to show how much area coverage can be achieved by the generated paths.

3.1 Path Specification

Both static and dynamic objects require a path information for their respective characterization. That is, a static object (e.g., a roadway) is to be textured over a path and a dynamic object (e.g., a car or a pedestrian) is to move over a path. We define an object path as follows:

Definition 3.1 (Object Path). An object path p^{i} is an interpolating function generated from an ordered coordinate set $\{(x_1^i, y_1^i, z_1^i), (x_2^i, y_2^i, z_2^i), \ldots, (x_n^i, y_n^i, z_n^i)\}$. A collection of paths is denoted as $\mathrm{P} = \{\mathrm{p}^1, \mathrm{p}^2, \ldots, \mathrm{p}^{\mathrm{m}}\}$.

When a path is associated with a static object, it implies the geometric design of the roadways (e.g., curves, hills, or intersections). When a path is associated with a dynamic object, it implies the route that the dynamic object moves over time. For example, the object paths in Figure 2 are specified as follows:[1]

[1]Note that the object path is defined as 3-dimensional spaces, but we use 2-dimensional spaces to simplify the explanation in the rest of the article.

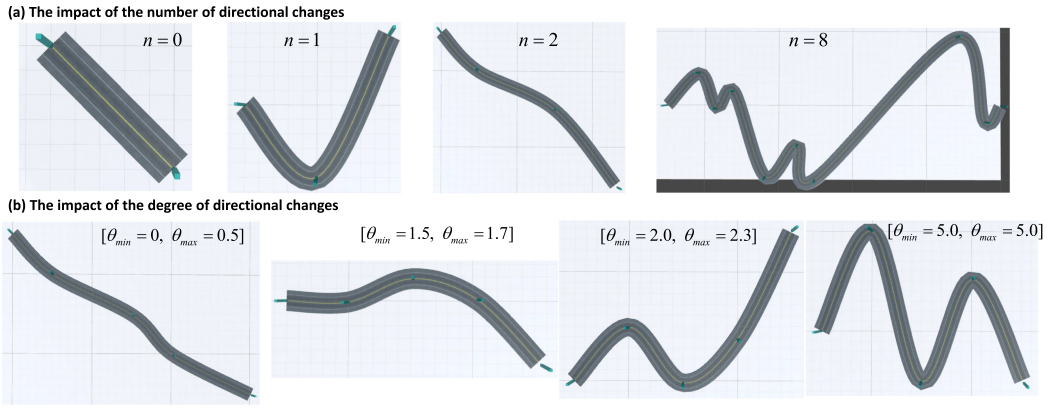

Fig. 4. Illustration of the number and the degree of directional changes.

Example 3.2. Figure 2 has three object paths $P = \{p^{curve}, p^{ped}, p^{car}\}$, where p^{curve} = {(50,100), (130,180), (304,179), (384,259), (450,325)}, p^{ped} = {(50,30), (55,50), (60,65), (48,65)}, p^{car} = {(20,10), (30,30), (40,35), (10,35)}.

Our goal is to formalize a specification from which such a path can be automatically generated. One important requirement of such a specification is to allow one to characterize diverse patterns of paths. For example, an adaptive cruise control needs to be tested under various types of curvatures [ISO15622 2010]; a lane change decision system needs to tested with other surrounding cars that change their lanes in various patterns [ISO17387 2008]. We characterize such a diversity by specifying how often and how severely a path changes its direction. To this end, a path specification is defined as follows:

Definition 3.3 (A Path Specification). A path specification S^p is a set of constraints that impose coordinates to be determined within the range of the boundary parameters; the boundary parameter is a tuple of ⟨A, N, T, D⟩; A defines an area where an object may stay; N is the min/max number of directional changes of an object $[n_{min}, n_{max}]$; T is the min/max degree of each directional change $[\theta_{min}, \theta_{max}]$; D is the min/max distance of any two successive directional changes $[d_{min}, d_{max}]$.

An area A consists of three sub-areas: A^s, where an object may start; A^d, where an object may reach; and A^m, where an object may stay to reach A^d from A^s. Figure 3 illustrates an example path (the red line) that can be generated from the specification of a pedestrian object. We introduce several templates of constraints and one can extend them depending on systems to be tested.

(Constraint 1) the number of waypoints: The number of waypoints determines the number of directional changes on a path. For example, p^{curve}, p^{ped}, and p^{car} in Figure 2 have three, two, and two waypoints, respectively, in between their first and the last coordinates; the directional change occurs on each waypoint. The following constraint is to specify the number of such waypoints to be generated:

$$n_{min} \leq n \leq n_{max}.$$

The constraint is specified with the known boundary parameters (i.e., n_{min} and n_{max}) and the unknown parameter (i.e., n). Intuitively, n is to be determined within the range of n_{min} and n_{max}. Figure 4(a) illustrates the impact of the number of waypoints. The way the unknown parameter is determined will be explained in a later section.

(Constraint 2) source and destination areas: From the number of waypoints determined by Constraint 1, one can know the index of the first and last waypoint in a path. The coordinates of

these two waypoints (i.e., (x_1, y_1), (x_n, y_n)) are called the source and destination of a path. The following constraint specifies the rectangular area[2] (i.e., A^s and A^d) where the source and destination waypoints are to appear:

$$(x^s_{min} \leq x_1 \leq x^s_{max}) \wedge (y^s_{min} \leq y_1 \leq y^s_{max}) \wedge$$
$$(x^d_{min} \leq x_n \leq x^d_{max}) \wedge (y^d_{min} \leq y_n \leq y^d_{max}).$$

The constraint is specified with the known boundary parameters of the source and destination area (i.e., x^s_{min}, x^s_{max}, y^s_{min}, y^s_{max}, x^d_{min}, x^d_{max}, y^d_{min}, y^d_{max}) and the unknown coordinates of the source and destination (i.e., x_1, y_1, x_n, y_n).

(Constraint 3) the degree of directional changes: The degree of directional changes are characterized with any three consecutive waypoints on a path. For example, the degree of the three directional changes, θ_1, θ_2, and θ_3, in Figure 3 are characterized in terms of the following three sets of coordinates: {(4,0), (3,5), (4,10)}, {(3,5), (4,10), (1,15)}, and {(4,10), (1,15), (0,35)}. The corresponding constraints are expressed setting those coordinates as unknown variables, while setting the minimum and maximum bounds of directional changes as known variables. The following is the example constraint:

$$\forall\ 0 \leq i \leq n-3,$$
$$\theta_{min} \leq \left(\left| \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} \right| \right) \leq \theta_{max}$$
$$\theta_{min} \leq \left(\left| \frac{z_{i+1} - z_i}{x_{i+1} - x_i} - \frac{z_{i+2} - z_{i+1}}{x_{i+2} - x_{i+1}} \right| \right) \leq \theta_{max}.$$

The above constraint is specified with the known boundary parameters (i.e., θ_{min} and θ_{max}) and the unknown three consecutive coordinates (e.g., x_i, x_{i+1}, x_{i+2}). The degree is characterized as a difference between two successive slopes and such difference should be bounded as the min/max parameters. Consequently, this constraint will determine the coordinates in a way that the degree of any directional changes become larger than the min parameter (θ_{min}), but smaller than the max parameter (θ_{max}). Figure 4(b) illustrates the impact of degree changes by varying the boundary parameters for the roadway generation; as θ_{min} and θ_{max} increase, the curvature becomes bigger.

(Constraint 4) the distance between successive directional changes: A directional change occurs on each waypoint of a path. Therefore, a distance between successive directional changes is characterized in terms of two coordinates of the waypoints. There are several ways to characterize a distance. The following constraint is based on the Euclidean distance:

$$d_{min} \leq \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (z_{i+1} - z_i)^2} \leq d_{max}.$$

Here, the unknown parameters are the coordinates (e.g., x_i, x_{i+1}) and the known parameters are the boundary parameters (i.e., d_{min}, d_{max}). Note that this constraint expresses the absolute distance between two directional changes, but does not express the distance on a particular axis. A certain vehicle feature needs to be tested on a roadway that stretches out in a particular direction. For example, a vision module that recognizes road signs, pedestrians, or other cars (e.g., a machine learning algorithm) is to be tested under various brightness levels to check its accuracy. In this case, it is necessary to design a roadway in a way that a vehicle drives toward the light source or drives away from the light source (e.g., the sun). In case one needs to consider the distance on a particular axis, the following constraint can be used:

[2]More complex types of area can be similarly defined with more parameters.

$$\forall\ 0 \le i \le n-1, \mathsf{d}_{\min} \le |x_{i+1} - x_i| \le \mathsf{d}_{\max}$$
$$\forall\ 0 \le i \le n-1, \mathsf{d}_{\min} \le |y_{i+1} - y_i| \le \mathsf{d}_{\max}$$
$$\forall\ 0 \le i \le n-1, \mathsf{d}_{\min} \le |z_{i+1} - z_i| \le \mathsf{d}_{\max}.$$

With the aforementioned constraints, we believe that one can express fairly complex paths for static and dynamic objects. In addition, these constraints can be extended in various ways, such as a path that has a closed form, a path that has alternating directional changes, a path that has uphills or downhills, and so on.

3.2 Test Criteria: Area Coverage

The path specification consists of four types of parametric bounds (i.e., A, N, T, D) to define the geometric shapes of paths to be generated. Note that there exists a number of paths that conform to a given path specification. Suppose a path specification specifies D = [1, 10], which implies the distance of any successive waypoints should be greater than 1 meter and smaller than 10 meters. If N is fixed as 5 (there exists 4 pairs of successive waypoints), then the bound of D will generate at least 4^{10} different paths, assuming the distance is limited as integer. The combination will exponentially grow as the ranges of the parametric bounds become larger.

Practically, it may not be quite meaningful to build a test environment generating all possible paths, because many of them may be very similar to each other. Then, the question is how to formally define the similarity of different paths to quantify the quality of generated paths. To this end, we define a test criteria called the area coverage as follows:

Definition 3.4 (Area Coverage). Given an area of interest A that contains multiple unit-regions with a unit-region size of w^{r} and l^{r}, an area coverge $C(\mathsf{A}, \mathsf{w}^{\mathsf{r}}, \mathsf{l}^{\mathsf{r}})$ is defined as the number of regions visited by paths among all regions.[3]

Intuitively, the area of interest A is divided into a range of smaller regions based on a chosen unit-region size. A path goes through multiple regions, and the area coverage is calculated based on the number of visited regions among all regions. When multiple paths are generated, a region is considered visited if and only if at least one path goes through the region.

According to Definition 3.4, a higher area coverage implies that paths are generated covering diverse regions within the area of interest. This is an important criterion in building a virtual test environment for autonomous features, because it is hard to expect any static or dynamic objects in the real environment to have uniform geometricity in their shapes or movements. For example, a lane keep assist feature needs to be tested on multiple roadways that have curves in various directions and positions; an adaptive cruise control needs to be tested with various movements of the preceding vehicle within the lane; a pedestrian warning system needs to be tested under various movements of pedestrians on crosswalks. Under a given path specification, the area coverage intends to achieve such geometric diversity in generating paths.

Here is the example of the area coverage:

Example 3.5 (Area Coverage). Consider the area of interest A in Figure 5 with a unit-region size $\mathsf{w}^{\mathsf{r}} = 2$ and $\mathsf{l}^{\mathsf{r}} = 2$. Consider three paths, $\mathsf{p}^{\mathsf{red}}$, $\mathsf{p}^{\mathsf{green}}$, and $\mathsf{p}^{\mathsf{blue}}$. Then the area coverage of $\mathsf{p}^{\mathsf{red}}$ and $\mathsf{p}^{\mathsf{green}}$ in the left figure (a) is (14/32)*100 = 43.75%. The area coverage of $\mathsf{p}^{\mathsf{red}}$ and $\mathsf{p}^{\mathsf{blue}}$ in the right figure (b) is (21/32)*100 = 65.62%.

[3]Note that we give a definition for 2-Dimensional case to simplify the explanation. However, this definition can be easily extended to 3-Dimensional case. In case of 3-Dimension, the unit-region will be a cube as opposed to a rectangle in 2-Dimension.

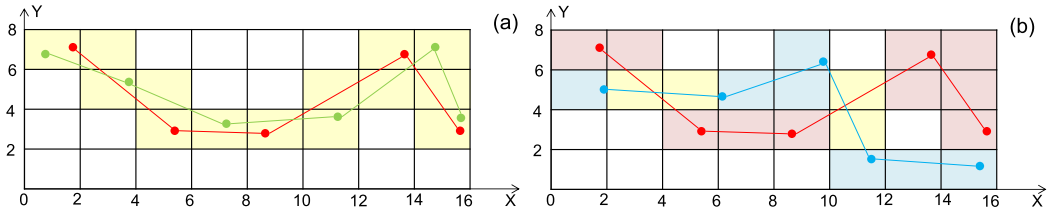

Fig. 5. Illustration of the area coverage for three paths p^{red}, p^{green}, and p^{blue}: yellow boxes are covered by any two paths at the same time; red boxes are covered by p^{red} only; blue boxes are covered by p^{blue} only.

In Figure 5(a), the area coverage of p^{red} only is the same as the area coverage of p^{red} and p^{green} together. In other words, p^{green} does not improve the area coverage from the one with p^{red} only, since p^{green} visits exactly same regions with those of p^{red}. In Figure 5(b), the area coverage of p^{red} and p^{blue} is greater by 21.87% (65.62% − 43.75%) than the one with p^{red} only, since p^{blue} (p^{red}) visits some regions not covered by p^{red} (p^{blue}).

This implies that when a path is generated in a way that covers similar regions covered by another path, it does not contribute to the improvement of the area coverage. Therefore, the path generation algorithm should be designed to choose a path, if possible, not to cover the regions already covered by another path to improve the area coverage. In the next subsection, we introduce the SMT-based path generation algorithm and show how much area coverage it can achieve through the empirical results.

3.3 Path Generation Algorithm

We propose the path generation algorithm using SMT (Satisfiability Modulo Theories) solver. The algorithm is given in Algorithm 1. The algorithm takes inputs of a path specification (S^p), a path distance parameter ($\mathcal{K}$), and the number of paths to be generated (q). The path specification is given as a set of SMT formulas (the examples are given in Section 3.1). The path distance parameter specifies the degree of dissimilarity of the next path from the previously generated paths. The number of paths specifies how many paths are expected to be generated from a given path specification. The algorithm returns a set of paths (P). We give the algorithmic detail below.

A random seed is generated to make the SMT solver search a solution (i.e., a set of coordinates that compose a path) in different ways (line 2). The SMT formula characterizing the path specification $\varphi(S^p)$, the chosen random seed r, and the timeout parameter t_{max} are given to the solver (line 4). There are three different scenarios when the SMT solver tries to find a solution. The first case is that the solver finds a solution (lines 5–7). The solution (p^i) is added to the solution set P. Then, a new constraint ($\varphi(\mathcal{K}, P)$) is added to the path specification. This new constraint determines how much the next solution should be dissimilar from the previously obtained solutions according to the path distance parameter $\mathcal{K}$. The path distance parameter will be explained later. As long as the while loop falls in the first case, those solutions will be added to P. The second case is that the solver cannot find a solution (lines 8–9). This happens when there exists no solution anymore and triggers exception *No-Sol* (line 16). In this case, the number of solutions is less than q under the given path specification (i.e., there exist no solutions as many as q). The third case is timeout (lines 10–11). Note that, in general, the problem is undecidable when the constraint involves non-linear constraints [Ratschan 2006]. Therefore, the solve may not terminate in some cases when the path specification involves non-linear constraints (e.g., Constraint 3). The timeout parameter makes the algorithm terminate, triggering the exception *TimeOut*. In this case, we may have the number of solutions less than q, but we do not know if there exist solutions as many as q.

ALGORITHM 1: PathGen

Input: S^p (a path specification), $\mathcal{K}$ (a path distance parameter), q (the number of paths to be generated)
Output: P (a set of paths)

```
try;
  r ← rand(); cnt ← 0;
  while(cnt < q);
    switch(p^i ← Solver (φ(S^p), r, t_max));
    case SAT;
      P ← P ∪ p^i; φ(S^p) ← φ(S^p) ∧ φ(K, P); cnt ← cnt + 1;
      break;
    case UNSAT;
      Exception(No-Sol); break;
    case TIMEOUT;
      r ← rand(); break;
    endswitch;
  endwhile;
  return P;
catch(TimeOut){Exceptionhandler};
catch(No-Sol){Exceptionhandler};
```

Theorem 3.6. *Any* $p \in P$ *generated from the path generation algorithm conforms to the path specification* S^p.

Proof. Suppose there exists a path p^i that does not conform to S^p. p^i must have been generated within at most q iterations of the algorithm. If p^i was generated in the very first iteration, then it must have conformed, since the solver uses the original S^p input. If p^i was generated thereafter, then it must have conformed to an updated constraint. However, the constraint is updated only by adding $\varphi(\mathcal{K}, p^i)$ in conjunction with the previous constraints. This means that the solver keeps the original S^p throughout q iterations of the algorithm. Hence, p^i must conform to the original S^p as well. This contradicts the assumption. □

Now, we explain the dissimilarity of a pair of paths using the path distance function. The intuition behind the notion of dissimilarity is to separate a path from another one by a certain amount of distance so a set of generated paths has a higher chance to cover diverse unit-regions, which will increase the area coverage. There is a number of ways to define the distance of paths; for example, we may consider different subsets of 3D coordinates that compose a path or different distance metrics such as *L0*, *L1,* or *L2* norms. Note that it is out of the scope of this article to study what number of coordinates and which distance metric are the most appropriate combination to maximize the area coverage. Instead, we introduce two example distance functions as follows:

Definition 3.7 (Object Path Distance). Let $\text{proj}_s^k(p^r)$ be the projection of kth waypoint of a path p^r onto a coordinate s, where s is either X-axis, Y-axis, or Z-axis. $Dist_{min}(p^i, p^j)$ is defined as $\min_{\forall s,k} f_d\,(\text{proj}_s^k(p^i), \text{proj}_s^k(p^j))$; $Dist_{max}(p^i, p^j)$ is defined as $\max_{\forall s,k} f_d\,(\text{proj}_s^k(p^i), \text{proj}_s^k(p^j))$, where f_d is the distance metric defined as $f_d : \mathbb{Z} \times \mathbb{Z} \rightarrow [0, \infty)$.

Intuitively, $Dist_{min}(p^i, p^j)$ takes the minimum difference of two coordinates among all waypoints in any axis so any waypoint in a pair of paths is separated at least by $Dist_{min}(p^i, p^j)$. However, $Dist_{max}(p^i, p^j)$ takes the maximum difference of two coordinates among all waypoints in any axis so any waypoint in a pair of paths is separated at most by $Dist_{max}(p^i, p^j)$. One can selectively

use $Dist_{max}(p^i, p^j)$ or $Dist_{max}(p^i, p^j)$ as $Dist(p^i, p^j)$; also, other distance metrics can be used as $Dist(p^i, p^j)$. We do not discuss all possible distance metrics here.

Example 3.8. Consider two paths p^1 = {(0,1), (1,1), (2,1), (3,1), (4,1)}, p^2 = {(0,2), (1,2), (2,7), (3,2), (4,2)}. Consider a distance metric $f_d = |\text{proj}_s^k(p^i) - \text{proj}_s^k(p^j)|$. $Dist_{min}(p^i, p^j) = 1$ and $Dist_{max}(p^i, p^j) = 6$.

The path generation algorithm also guarantees the dissimilarity of any pair of paths in the solution set.

Theorem 3.9. *For any pair of* p^i *and* $p^j \in P$ *generated from the path generation algorithm,* $Dist(p^i, p^j) \geq \mathcal{K}$.

Proof. Suppose there exist some p^i and p^j such that $Dist(p^i, p^j) < \mathcal{K}$. Suppose p^i was generated before p^j. Then, p^i is added to P; P is passed to $\varphi(\mathcal{K}, P)$ to include a constraint that the distance of the next generated path should be equal to $\mathcal{K}$ or greater than $\mathcal{K}$. Since P is additive throughout the loop, p^i remains in P throughout the rest of the loop. When p^j is generated after p^i, the solver finds a solution that is dissimilar from p^i at least by $\mathcal{K}$. Hence, the distance of the newly generated p^j from p^i must be equal or greater than $\mathcal{K}$. This contradicts the assumption that $Dist(p^i, p^j) < \mathcal{K}$. □

Selection of $\mathcal{K}$ determines the granularity of the dissimilarity, consequently impacting the area coverage. In general, a larger $\mathcal{K}$ requires a pair of paths to be separated farther in some axis, making its area coverage grow faster. However, a larger $\mathcal{K}$ does not necessarily mean the generated paths eventually achieve a higher area coverage. This is because the path specification and the path distances are negatively co-related. The path specification determines the geometric constraint as to how a path should be generated using several bounded parameters. However, the path distance aims at generating paths being separated by each other as much as it is specified as $\mathcal{K}$ within the geometric constraint defined by the path specification. Therefore, a larger $\mathcal{K}$ may not be able to generate as many paths as smaller $\mathcal{K}$ eventually. However, if the number of generated paths is sufficiently large with a smaller $\mathcal{K}$, it may reach a higher area coverage than a larger $\mathcal{K}$.

Figure 7 is the experiment result to show the impact of selection of $\mathcal{K}$. Given the same path specification and the same unit-region size for the area coverage, we generated paths varying $\mathcal{K}$ from 0 to 3. With $\mathcal{K} = 0$, it reaches approximately 50% of the area coverage after generating 100 paths (blue line). The similar area coverage could be reached only with 15 paths when $\mathcal{K} = 2$ (green line). However, when $\mathcal{K} = 2$, it could not generate more than 15 paths, since there exists no solution that meets the path specification and the path distance criteria. With $\mathcal{K} = 1$, however, it could reach approximately 80% of the area coverage with 40 paths (red line).

Figure 6 show the visualization of the generated paths in case of $\mathcal{K} = 0$ and $\mathcal{K} = 1$, respectively. With $\mathcal{K} = 0$, the paths are generated in a close proximity as the number of paths increases, which results in a low area coverage. However, with $\mathcal{K} = 1$, the generated paths tend to be separated by each other more than $\mathcal{K} = 0$, which results in a higher area coverage.

4 BEHAVIOR SPECIFICATION AND GENERATION

This section explains how to specify behavioral aspects of a dynamic object using timed automata. We define an object run that is a sequence of mode changes of dynamic objects. Then, we introduce the edge and location coverage to quantify the quality of the generated object runs. We introduce a model-checking method that can automatically generate object runs that satisfy the coverage criteria.

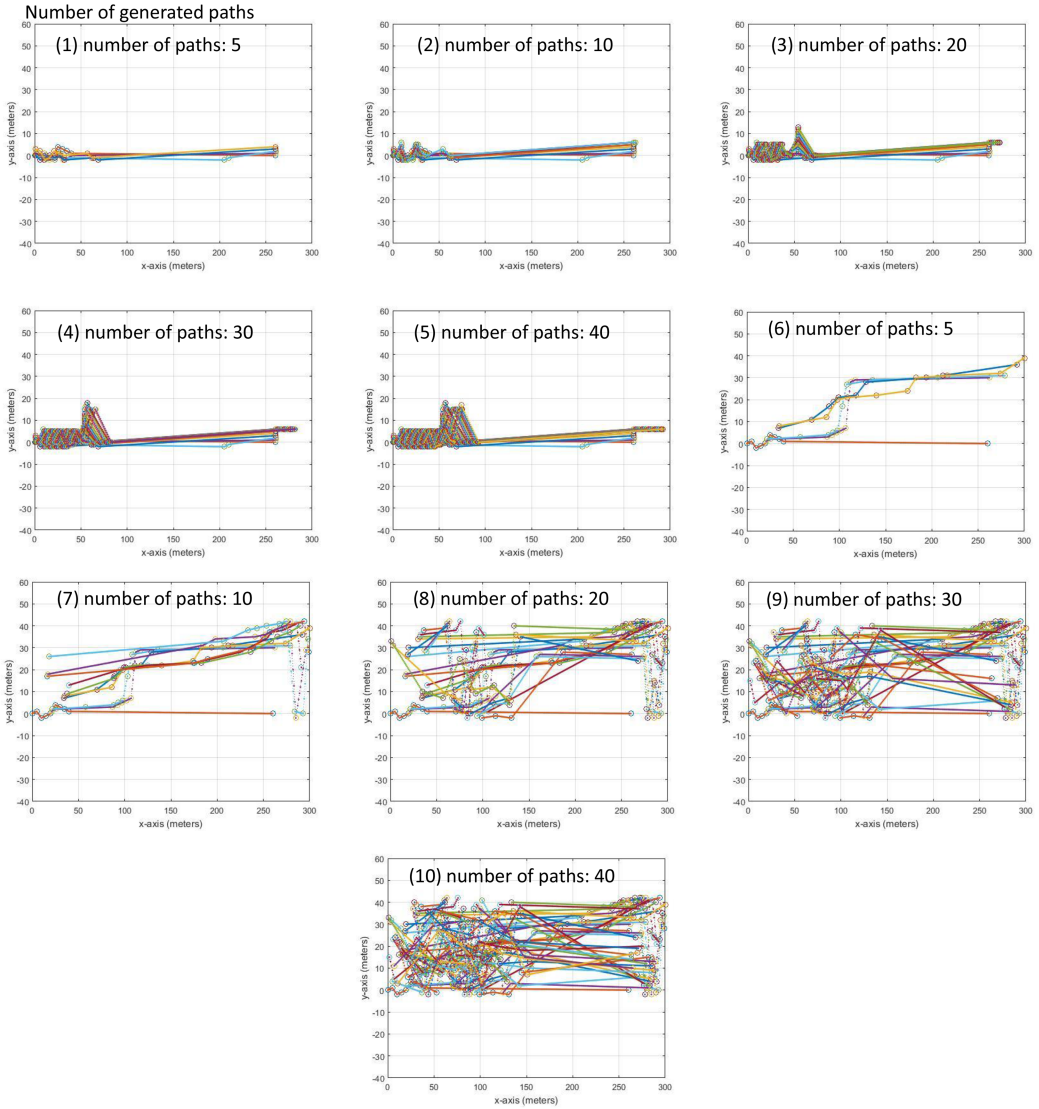

Fig. 6. Experiment results of area coverage: (1)–(5) were obtained from the path distance parameter $\mathcal{K} = 0$, and (6)–(10) were obtained from $\mathcal{K} = 1$.

4.1 Behavior Specification

Our road environment is categorized according to characteristics that remain unchanged (static objects) or change over time (dynamic objects) after their visualization. The object path defined in Section 3 intends to specify some characteristics of both static objects and dynamic objects; that is, a geometric design of a static object (e.g., roadways) and a trajectory of a dynamic object (e.g., vehicles or pedestrians) are characterized from the object path definition. However, when it comes to the dynamic objects, another level of abstraction is necessary to characterize those aspects that change over time. Those aspects include, for example, a vehicle reducing or increasing its speed upon signal changes; a pedestrian stops or starts walking upon signal changes. Such mode changes of dynamic objects are typically triggered by timeout or interactions with other dynamic objects

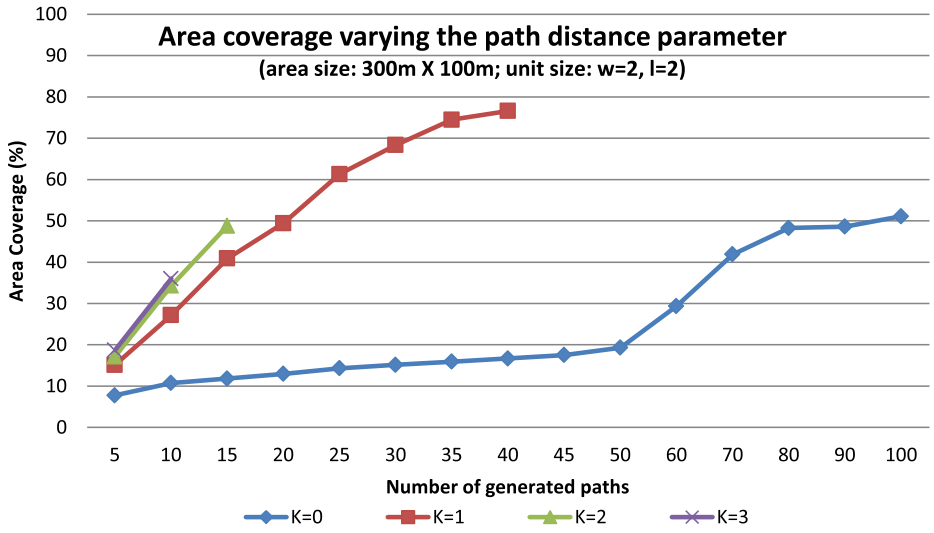

Fig. 7. Experiment results of the area coverage for different path distance parameters.

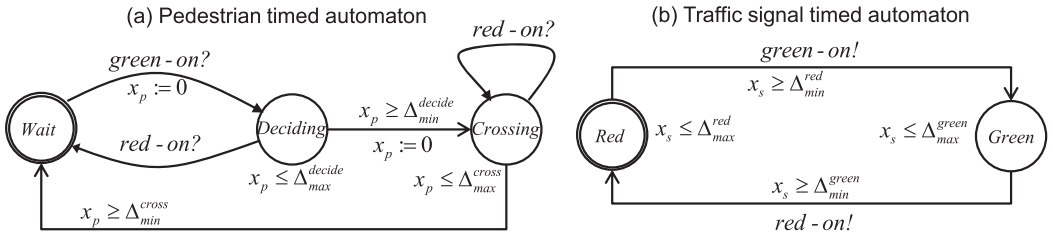

Fig. 8. Example behavior model of pedestrian and traffic signals.

(i.e., vehicles, pedestrians, and traffic signals). The path specification is not sufficient enough to specify those aspects.

In our test framework, such an object behavior is specified as a timed automaton as follows:

Definition 4.1 (Behavior Specification). An object behavioral specification S^b is defined as a timed automaton $\langle L, l_0, C, A, E, I \rangle$, where L is a set of locations; $l_0 \in L$ is an initial location; A is a set of input/output actions; C is a set of clocks; E is a set of transitions between locations; I assigns invariants to locations.

We follow the original semantics of timed automata [Alur and Dill 1994]. Here is the example of the semantics of the behavior specification in Figure 8. The timed automata in the example characterizes the timed behavior of the pedestrian and traffic signal, respectively. The pedestrian timed automaton (a) has three locations: *Wait*, *Deciding*, and *Crossing*; *Wait* is the initial location. This system has a clock variable (x_p) used for invariants or guard conditions. Some locations are associated with an invariant to enforce a system to stay in the location no more than a certain amount of time ($x_p \leq \Delta_{max}^{decide}$ for *Deciding* and $x_p \leq \Delta_{max}^{cross}$ for *Crossing*). There are five transitions associated with the clock variable (x_p) and input actions ($red - on?$ and $green - on?$). The input actions are synchronized with the output actions of the traffic signal timed automaton in 8(b). From *Wait*, the system can take a transition to *Deciding* synchronizing with *green-on!* with a clock reset ($x_p := 0$). From *Deciding*, the system can take a transition back to *Wait* synchronizing with $red - on!$; the system also can take another transition to *Crossing* if the clock value of x_p is equal

or greater than Δ_{min}^{decide}. From *Crossing*, the system can take a self-transition synchronizing with *red-on!*; the system can also take a transition to *Wait* if the clock value of x_p is equal or greater than Δ_{min}^{cross}. If multiple transitions are enabled in a location, then one of either is taken non-deterministically. The semantics of the traffic signal timed automaton can be similarly explained.[4]

We define an object run as follows:

Definition 4.2 (Object Run). An object run b^{i} is a sequence of the form $l_1 \xrightarrow{(a_1,t_1)} l_2 \xrightarrow{(a_2,t_2)} \ldots \xrightarrow{(a_{n-1},t_{n-1})} l_n$, where $l_i \in L$, a_i is the input/output action that triggers the transition and t_i is the time elapse from the moment when l_i enters to the moment l_{i+1} enters (i.e., the duration where an object stays in l_i). A collection of runs is denoted as $\mathrm{B} = \{\mathrm{b}^1, \mathrm{b}^2, \ldots, \mathrm{b}^{\mathrm{m}}\}$.

From the behavior specification, there may exist a number of object runs. Suppose the timing parameters of Figure 8 are determined as follows: $\{\Delta_{min}^{decide} = 2, \Delta_{max}^{decide} = 5, \Delta_{min}^{cross} = 25, \Delta_{min}^{cross} = 30, \Delta_{min}^{red} = 10, \Delta_{max}^{red} = 10, \Delta_{min}^{green} = 30, \Delta_{max}^{green} = 30\}$. Consider following object runs of a pedestrian that can be generated from the behavior specification:

$$Wait \xrightarrow{(green-on,5)} Deciding \xrightarrow{(-,2)} Crossing, \tag{6}$$

$$\begin{aligned} Wait \xrightarrow{(green-on,5)} Deciding \xrightarrow{(red-on,30)} Wait \xrightarrow{(green-on,10)} Deciding \xrightarrow{(-,3)} \\ Crossing \xrightarrow{(red-on,15)} Crossing \xrightarrow{(-,25)} Wait. \end{aligned} \tag{7}$$

The object run 6 can be generated with the above parameter combination. However, the object run 7 cannot be generated with the above parameter combination. One reason is that 7 includes the transition from *Deciding* to *Wait* with *red-on* synchronization. However, this is not possible, since the pedestrian can stay in *Deciding* no longer than 5 time-units while the signal must stay in *Green* for 30 time-units. This can be easily proved via model-checking by writing a query in temporal logic formula that captures the transition is eventually taken. One way to write this query is to add an auxiliary Boolean variable on the transition with its initial value of false, and check the reachability if the auxiliary variable becomes true. However, if Δ_{max}^{decide} is set to 30, the object run 7 can be generated.

4.2 Test Criteria: Edge and Location Coverage

Now, we explain the coverage criteria of the behavioral specification. There are several coverage criteria to generate test cases from timed automata in literatures. In particular, we extend the edge coverage and the location coverage defined in Hessel et al. [2008]. The edge coverage is satisfied if an object run traverses every edge (i.e., transition) of a selected timed automaton. For example, there are five transitions in the pedestrian automaton in Figure 8. The object run 6 does not satisfy the edge coverage, since only two transitions out of five are visited (i.e., 40% of the edge coverage). The object run 7 satisfies the edge coverage, since all five transitions are visited (i.e., 100% of the edge coverage). The location coverage is satisfied if an object run traverses every location of a selected timed automaton. For example, both object runs 6 and 7 satisfy the location coverage, since all three locations are visited.

[4]Note that one can extend this model to express more complex behavior of the pedestrian. For example, a pedestrian may interact with oncoming vehicles in a dangerous way or may make a crossing decision in a roadway where a traffic light is not installed. Such additional behavior can be expressed by adding more locations/transitions or having extra timed automata that is synchronized with Figure 8.

However, there are many cases where an autonomous feature needs to be tested under a sufficiently long object run. For example, a vehicle typically relies on vision sensors (e.g., camera or LIDAR) to detect objects (e.g., a pedestrian). Such a sensor typically has a detection rate that is affected by many aspects, such as detection areas of sensors, orientations of objects, or weather conditions. Some simulation tools provide a model of such a sensor that allows engineers to test their autonomous features under complex parameter combinations that affect the detection rate of a sensor. Hence, even though the autonomous feature passed a test with 100% of edge or location coverage according to the above definition, it does not necessarily mean it will pass in the next test due to such probabilistic nature of sensors. To accommodate such a sufficiency criteria, we add an extension to the definition of edge and location coverages as follows:

Definition 4.3 (Edge Coverage). Suppose each transition in E is assigned a unique identifier. Let a set of minimum number of visits required for each transition $\mathsf{U} = \{u_1, u_2, \ldots, u_n\}$, where u_i implies the minimum number of visits required for ith transition and n is equal to the number of transitions in E. An object run satisfies an edge coverage if it includes more than u_i visits for all $1 \leq i \leq n$.

We informally explain how to generate an object run that meets the edge coverage using UPPAAL [Larsen et al. 1997], which allows a transition to have updates and guard conditions of variables. An auxiliary integer variable v_i with an initial value of zero is created for each transition i. Each transition i is associated with an update statement over the auxiliary variable: $v_i := v_i + 1$. Then, one can write the following logic formula that informally says the system can reach a state where each transition is visited at least the required number of visits:

$$\mathsf{E} \diamond ((v_1 \geq u_1) \wedge (v_2 \geq u_2) \wedge \cdots \wedge (v_2 \geq u_2)). \tag{8}$$

If the above formula verifies, then it means that the edge coverage is achieved; then UPPAAL generates an object run as a witness. If the above formula does not verify, then it means that one cannot generate an object run such that it satisfies the edge coverage under the given behavior specification.

Example 4.4 (Edge Coverage). Suppose the number of visits required for each edge in the pedestrian automaton in Figure 8 is given as $\mathsf{U} = \{5, 5, 5, 5, 5\}$. Further, suppose the timing parameters of Figure 8 are determined as follows: $\{\Delta^{decide}_{min} = 2, \Delta^{decide}_{max} = 30, \Delta^{cross}_{min} = 25, \Delta^{cross}_{min} = 30, \Delta^{red}_{min} = 10, \Delta^{red}_{max} = 10, \Delta^{green}_{min} = 30, \Delta^{green}_{max} = 30\}$. Then, the following is one possible object run generated from the model satisfying the edge coverage:

$$Wait \xrightarrow{(green-on,10)} Deciding \xrightarrow{(-,2)} Crossing \xrightarrow{(red-on,28)} Crossing \xrightarrow{(-,2)}$$
$$Wait \xrightarrow{(green-on,8)} Deciding \xrightarrow{(-,6)} Crossing \xrightarrow{(red-on,24)} Crossing \xrightarrow{(-,3)}$$
$$Wait \xrightarrow{(green-on,7)} Deciding \xrightarrow{(-,12)} Crossing \xrightarrow{(red-on,18)} Crossing \xrightarrow{(-,8)}$$
$$Wait \xrightarrow{(green-on,2)} Deciding \xrightarrow{(-,2)} Crossing \xrightarrow{(red-on,28)} Crossing \xrightarrow{(-,1)}$$
$$Wait \xrightarrow{(green-on,9)} Deciding \xrightarrow{(-,6)} Crossing \xrightarrow{(red-on,24)} Crossing \xrightarrow{(-,4)}$$
$$Wait \xrightarrow{(green-on,6)} Deciding \xrightarrow{(red-on,30)} Wait \xrightarrow{(green-on,10)} Deciding \xrightarrow{(red-on,30)}$$
$$Wait \xrightarrow{(green-on,10)} Deciding \xrightarrow{(red-on,30)} Wait \xrightarrow{(green-on,10)} Deciding \xrightarrow{(red-on,30)}$$
$$Wait \xrightarrow{(green-on,10)} Deciding \xrightarrow{(red-on,30)} Wait.$$

The location coverage and its generation method can be similarly defined, so we do not give the details. One benefit of the edge/location coverage is to allow engineers to get more samples of particular mode changes that are of interest. For example, one may assign a higher number of required visits on the transition from *Deciding* to *Crossing* to perform more tests on the scenario when the pedestrian starts crossing the crosswalk after the traffic light changes.

Note that one can also design more coverage criteria and test generation methods from the system described in timed automata. One possible extension is on the way to sample timing of a transition to be taken. For example, suppose min/max intervals of the transition from *Deciding* to *Crossing* are given as $\Delta_{min}^{decide} = 2$ and $\Delta_{max}^{decide} = 30$. One can design a test criteria and generation method to take the transition at a particular timing. For example, the interval [2, 30] can be divided as a set of sub-intervals based on a given timing granularity, such as {[2, 9], [9, 16], [16, 23], [23, 30]}. One possible criterion is to have an object run to initiate a transition for each sub-interval (e.g., an object run includes transitions taken at 2, 10, 15, 25). Another criterion is to have an object run to initiate a transition more often on a certain sub-interval (e.g., an object run includes transitions taken at 2, 3, 7, 10, 15, 25). We do not discuss all variations of the coverage criteria here, but one can extend the sampling techniques introduced in the literatures [En-Nouaary et al. 2002; Larsen and Yi 1994].

4.3 Test Generation for Dynamic Objects

The test generation of a dynamic object aims at generating test cases by associating the object paths (defined in Section 3.1) and the object runs (defined in Section 4.1). A test case of a dynamic object is defined as follows:

Definition 4.5 (A Test Case). Given a collection of object paths P and a collection of object runs B, a test case of a dynamic object T^i is a pair of (p^i, b^i) where $p^i \in P$ and $b^i \in B$.

Here is the informal semantics of the test case. A dynamic object moves along a path p^i during the test execution. When it reaches the end of the path, it starts over by moving back to the beginning of the path or moving reversely from the end of the path. The dynamic object is also associated with a range of pre-defined modes that correspond to each location of the object run. For example, a pedestrian object may stop in a particular position on the path (in *Wait* or *Deciding* locations) or start moving on the path with a certain speed (in *Crossing* location). Those mode changes happen upon timeout or when it receives an input from other dynamic objects (e.g., *green-on* or *red-on* of a traffic signal).

Algorithm 2 shows the pseudo code for the test generation algorithm of dynamic objects. Suppose the number of dynamic objects is given as q (e.g., 10 vehicles or 20 pedestrians). The algorithm

ALGORITHM 2: TestGen (the test generation algorithm)

Input: P (a set of object paths), B (a set of object runs), q (the number of dynamic objects to be generated)
Output: $T^1, \ldots, T^{size(O)}$ (a test suite)

```
1 O=GenObjectSet(q);
2 for each p^i ∈ P;
3   for each b^j ∈ B;
4     obj^k ← GetObject(O);
5     T^k ← Associate(obj^k, p^i, b^j);
6   endfor;
7 endfor;
```

first generates the specified number of instances of a dynamic object (line 1). Then, an instance of a dynamic object (obj^k) is selected from the set O (line 4). A test case is created by selecting an object instance and associating it with some object path in P and some object run in B (line 5). This process is repeated for all combinations of object paths in P and object runs in B.

If q is different from $|P| \times |B|$, then each pair of an object path and an object run is assigned to an object instance according to the implementation of the function *GetObject* and *Associate*. For example, if q is equal or greater than $|P| \times |B|$, each pair of an object path and an object run is assigned to a unique object instance. If q is smaller than $|P| \times |B|$, then each object instance may be associated with multiple pairs of object paths and object runs. The discussion of all possible implementations of the function *GetObject* and *Associate* is out of the scope of this article.

5 CASE STUDY

5.1 Motivation

According to Office [2009], although intersections represent a very small percentage of U.S. surface road mileage, more than one in five pedestrian deaths are the result of a collision with a vehicle at an intersection. To mitigate or avoid the hazard, stakeholders (e.g., OEMs or government agencies) started considering various pedestrian crash avoidance/mitigation systems. One example is the left- or right-turn warning system at intersections. A sensor that can detect pedestrians is installed on the roadside infrastructure of the intersection. It informs drivers who make a left or right turn at the pedestrian crossing via communication channels, such as Dedicated Short Range Communications (DSRC), which is used to generate visual or audio signals so drivers can avoid the collision. It is important to analyze the effectiveness and potential safety benefits of such pedestrian crash avoidance/mitigation systems according to the recent report of Yanagisawa et al. [2017]. Our case study aims at systematically generating an intersection environment according to the proposed framework to test the effectiveness of this warning system in mitigating the pedestrian collision hazard. The test environment includes the geometric design of the intersection (i.e., a static object), the signal phases of traffic lights (i.e., dynamic objects), and pedestrian behavior (i.e., dynamic objects). In particular, we give details on how to generate pedestrians to analyze the following two aspects:

- How early should the warning signal be sent to the driver to avoid collision with pedestrians?
- How fast should drivers react to the warning signal to avoid collision with pedestrians?

5.2 State-of-the-Art Test Standards for the Pedestrian Collision Avoidance System

Several international standards have been proposed to test the pedestrian collision avoidance system in many different driving scenarios. Those are designed to test how well a vehicle detects a pedestrian under various conditions and applies a brake in a timely manner to avoid a collision. We briefly introduce ISO 19237 (Pedestrian detection and collision mitigation systems—Performance requirements and test procedures) [ISO19237 2017] and EURO NCAP standards (European New Car Assessment Programme Test Protocol—AEB VRU systems) [NCAP 2017] to explain the general test procedure to be configured in the physical environment.

Figure 9 illustrates the three representative test scenarios from the standards.[5] A physical mockup of a pedestrian is placed in (a) or (b) or (c), depending on the driving scenarios to be tested. The first scenario is to test if the SUT (d) can avoid a collision with an adult walking or

[5]Note that this figure includes both ISO 19237 and EURO NCAP together for illustration purposes. Each standard may selectively include some of the illustrated aspects.

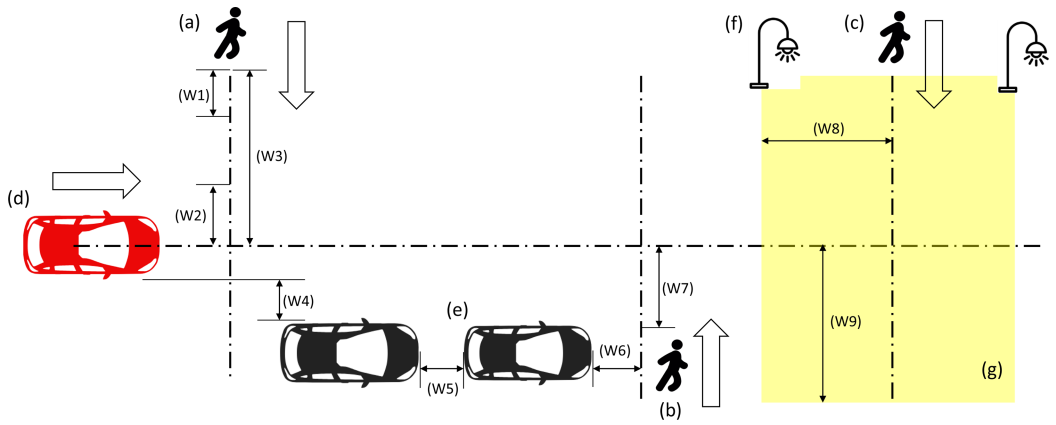

Fig. 9. Illustration of the standardized test environment in EURO NCAP Test Protocol (AEB VRU systems) [NCAP 2017] and ISO 19237 [ISO19237 2017].

running from its near side or far side; the corresponding test procedure is that the pedestrian crosses the road straight from the location (a), and the SUT driving toward the pedestrian is expected to detect the pedestrian and stop accordingly to avoid the collision.

The second scenario is to test if the SUT can avoid a collision with a child running into the road from obstruction vehicles; the corresponding test procedure is that the pedestrian crosses the road straight from the location (b) where the SUT cannot detect the pedestrian from the beginning due to the obstruction vehicle (e); the SUT is expected to detect the sudden appearance of the pedestrian and stop accordingly to avoid the collision.

The third scenario is to test if the SUT can avoid a collision with a pedestrian under various illumination situations where some make the SUT harder to detect the pedestrian; the corresponding test procedure is that the pedestrian is exposed in both bright and dark conditions in the area (g) that can be controlled by multiple street lights; the pedestrian crosses the road straight from the location (c) under various illumination situations where the SUT is expected to detect the pedestrian to avoid collision.

Note that there are various parameters (W1–W9) considered in designing the above test environment, such as the distance between the pedestrian and the SUT, the relative position of the pedestrian from the roadway, the speed of the SUT, and the pedestrian and the degree of the illuminance. Those parameters are given from the standards.

One can test the pedestrian collision avoidance system according to the test procedure defined in the above standards. However, there are several ways to improve the test procedure to accommodate more realistic aspects of the pedestrian. For example, one can consider diverse types of pedestrian paths, a higher number of pedestrians, or different geometric designs of the roadways (e.g., intersections). We explain our case study below to demonstrate the capability of the proposed approach in automatically generating more complex test environments in the virtual environment.

5.3 Procedure of Test Generation and Execution

System Under Test: We implemented the hypothetical pedestrian warning system in Unity3D simulation engine as follows: First, a pedestrian detection mechanism is implemented on the roadside unit (i.e., traffic light). This roadside unit is able to detect any pedestrian presence within a certain area (e.g., the crosswalk). Upon detection, it sends a warning message to the vehicles within the warning zones. We consider three different warning zones as illustrated in Figure 10. The size

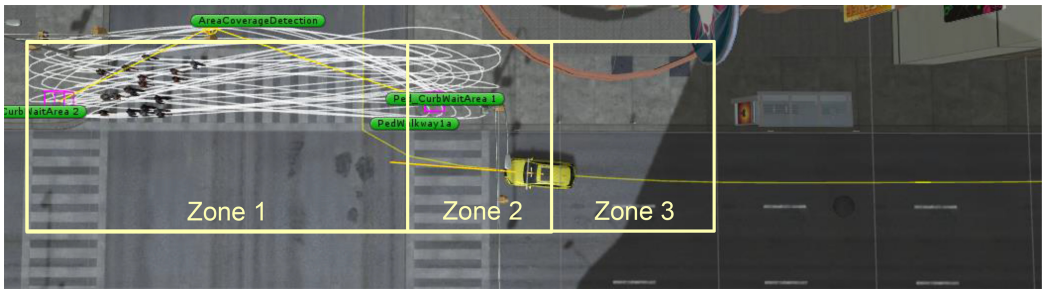

Fig. 10. Illustration of warning zones.

Table 1. Experiment Parameters of the Path Specification and the Behavior Specification for Pedestrians

(a) Parameters of six path specifications for pedestrian object

Path Parameters	Number of direction changes	Distance between two adjacent direction changes (meters)	Margin size (meters)	# of paths
Path Spec 1	0	(Min: 1 Max: 5)	0	20
Path Spec 2	3	(Min: 1 Max: 5)	0	20
Path Spec 3	0	(Min: 1 Max: 5)	3	20
Path Spec 4	3	(Min: 1 Max: 5)	3	20
Path Spec 5	1	(Min: 1 Max: 5)	2	20
Path Spec 6	2	(Min: 1 Max: 5)	2	20

(b) Parameters of six behavior specifications for pedestrian object

Behavior Parameters	Traffic law violation at start	Min deciding time	Max deciding time	Walk time
Behavior Spec 1	No	0	0	Finish within green
Behavior Spec 2	No	2	5	Finish within green
Behavior Spec 3	Yes	-2	-2	Finish within green

of warning zone determines how early a vehicle approaching the intersection can receive the warning signal. The prototype of the following autonomous features is implemented in six vehicles. Each vehicle makes multiple right turns in a four-way intersection at various speeds. The vehicle receiving the warning message from the roadside unit automatically slows down its speed. Each vehicle is programmed to have different reaction times to simulate various aspects, such as network delays between the vehicle and the roadside unit, the detection delay of the pedestrians, and so on. The programmed reaction time ranges from 0 (i.e., immediately apply the brake) to 3 seconds (i.e., apply the brake 3 seconds after the warning message is received).

Test Generation: To test the aforementioned system, we created six path specifications (introduced in Section 3) and three behavior specifications (introduced in Section 4) for the pedestrian generation. Table 1 summarizes several parameters used in each specification. The path specifications allow pedestrian paths to be generated by varying the number of directional changes, the distance between successive directional changes, and the area where the pedestrian is allowed to walk. For example, the path specification 4 implies that each of 20 generated pedestrian paths has three directional changes on the crosswalk; the distance between any successive directional change is determined between one meter and five meters; the pedestrian is allowed to walk outside of the crosswalk by three meters.

The behavior specifications allow pedestrians to change their modes interacting with the traffic signal according to the semantics of the timed automata model in Figure 8. The traffic signal changes from green to red and red to green. A pedestrian is expected to cross the crosswalk on the green light but not on the red light. We considered several timings as to when the pedestrian starts crossing in response to the traffic signal changes. For example, the behavior specification 1 implies that each pedestrian should cross the crosswalk immediately after the signal change. The behavior specification 2 implies that each pedestrian may start crossing in between two and five seconds after the signal change. The behavior specification 3 implies that each pedestrian may start crossing two seconds before the signal change (i.e., the pedestrian violates the traffic law).

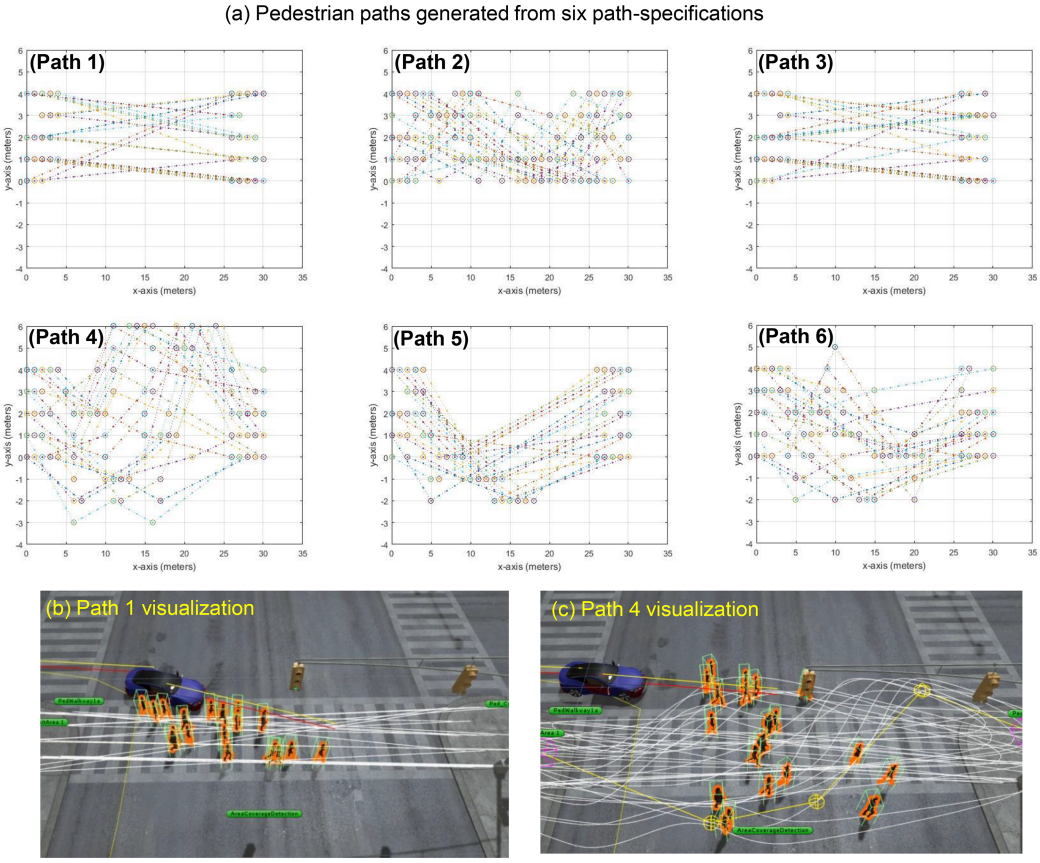

Fig. 11. Example of pedestrian paths and visualization.

Once those pedestrians start crossing, all the specifications have the pedestrians finish crossing before the signal changes from green to red.

The generated pedestrian paths and behavior are visualized in Unity3D along with the above system under test. Figure 11 shows the visualization of the generated paths from the six path specifications. Figure 11(a) plots the generated paths from the six path specifications where X-axis and Y-axis indicate the position of the path with respect to the position of the intersection. Figure 11(b) and Figure 11(c) show how the paths generated from the path specifications 1 and 4 are visualized in Unity3D. Twenty visual objects of pedestrians are created and mapped to each path. In addition, each pedestrian object is associated with a logic that implements the semantics of the behavior run generated from each behavior specification. As a result, each pedestrian moves along the assigned path (generated from the path specification) according to the behavior (generated from the behavior specification).

5.4 Test Result

Each vehicle makes 30 right turns under the test environment where pedestrians move on the crosswalk according to the path specification and the behavior specification. We created 18 types of pedestrians from the combination of the six path specifications and three behavior specifications in Figure 1. The test system records the accident event whenever there exists a collision

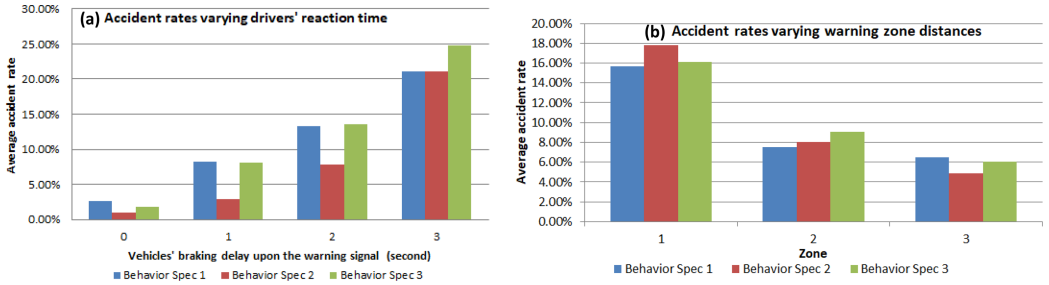

Fig. 12. Experiment results of the right-turn warning system: (a) varying drivers' reaction times, (b) varying warning zones.

between a vehicle and a pedestrian. The accident rates under several different scenarios are given in Figure 12. Figure 12(a) shows the accident rates by varying vehicles' reaction times in response to the warning message; here, the reaction time means the delay between the timing when the roadside unit detects the pedestrian and the timing when the vehicle actually applies the brake. Figure 12(b) shows the accident rates by varying zones where the warning message is sent. For each reaction time (a) or zone (b), there are three bars (blue, red, and green). Each bar indicates the average accident rate for each behavior specification; each average is calculated from the pedestrian's collision generated from the six path specifications.

Here is the analysis of the test results: The accident rate increases as the vehicle's response time increases. When the reaction time is zero, the average accident rates are below 5%. However, when the reaction time increases up to three seconds, the average accident rate reaches to 25%. This is because the vehicle does not stop fast enough to avoid the collision with the pedestrians. Note that even though the vehicle applies the brake with zero delay (i.e., the vehicle was able to apply the brake immediately after detecting the pedestrian crossing), it leads to some accidents. This is because there is a time gap due to the brake distance (i.e., the delay between the timing when the brake is applied and the timing when the vehicle actually stops), which makes the crash unavoidable.

The accident rate also increases as the warning zone is closer to the crosswalk (the driver reaction time is fixed as two seconds in this case). For Zone 1, the vehicle is notified only if it enters the crosswalk, so it gives little time for the vehicle to stop. In this case, the accident rates increase up to 18%. However, the accident rates reduce as the warning zone covers a wider area.

We showed how the proposed test generation framework can be used to test the pedestrian warning system. Design and analysis of such a warning system is non-trivial in the real world, since many aspects from different components are correlated in a complex way, as shown in the test results. Those aspects include pedestrian behavior, vehicle reaction time, the communication delay between the vehicle and the infrastructure, the geometric design of the roadway, and so on. Since those aspects cannot be easily controlled in the real world, we believe the test results from virtual prototyping can be useful feedback in improving the quality of the system.

6 RELATED WORK AND DISCUSSION

6.1 Related Work

There are several related works in the road generation domain. Zhao et al. evaluated the commercial tool called Creator Road Tool, which allows users to specify road parameters, such as the entry/exit point of a curve and its radius, to create horizontal and vertical curves [Institute and (U.S.) 2011]. However, those parameters are associated with a single curve (cf., our parameters

characterize a set of curves), so it does not automatically generate road segments that consist of multiple curves (i.e., a user has to manually enter new parameters for each curve). Some work studied a way to develop analytic road representations using curve-fitting algorithms based on real road data [Zhao and Farrell 2011; Jiménez et al. 2009]. Unlike our approach, their works intend to generate curves from the actual data sets (e.g., trajectory of real vehicles) obtained from actual driving on real roads. This means that their approach aims at generating roads that the vehicle has driven in the real world, while our work focuses on generating roads that meet the testing standard. Other works proposed a way to generate road surface irregularity to test a certain vehicle component (e.g., suspension systems) [Kavanagh and Ramanathan 1982; Jian and Ruichen 1999]. Generating such road surfaces is out of the scope of this work, but we believe our work can complement such works to generate more realistic roads. Galin et al. proposed the framework that takes a nature scene (e.g., a mountainous scene) as an input and generates roads in a way that takes into account the slope of the terrain and natural obstacles (e.g., trees or rivers) [Galin et al. 2010]. Even though this work does not address how to generate curves from the coverage criteria defined in our work, the way this work takes into account background scene can complement our approach to create a more realistic road environment.

Our work was largely motivated by the testing domain. Most of those works aim at generating test cases to find bugs or to prove the correctness of a program, but we believe that the fundamental techniques used in their work and our work share some commonalities, so we introduce some of those works here.

Some survey works introduced a number of techniques that can be applicable for test generation using symbolic execution [Păsăreanu and Visser 2009; Cadar et al. 2011]. In particular, the SMT decision procedures for combinations of theories can be used for test case generation achieving a high degree of coverage. Such a method allows one to specify test specification so the symbolic execution can automatically generate the test cases that meet the specification. Even though scalability is still the main obstacle of applying the symbolic execution techniques in large scale, it indicates that much research is going on to tackle the issue such as parallelizing the analysis. Peleska et al. studied an approach for automated test data generation for concurrent real-time systems based on the abstract interpretation algorithm [Peleska et al. 2011]. Their main contribution is that, given an initial state of the real-time models and the test case specification, it generates the test cases up to the user-specified limit, and those test cases are guaranteed to conform to the specification. To evaluate the performance of their algorithm, they generated test cases using different techniques and measured the execution time, which shows the considerable performance improvements where abstract interpretation is used. Godefroid et al. [2012] applied the test case generation method to the security domain. One way for the attackers to find security vulnerability is to randomly mutate well-formed program inputs and test the program with the inputs to trigger a bug such as a buffer overflow. This work proposed a whitebox fuzzing method that symbolically executes the program gathering constraints on inputs from conditional branches encountered along the execution. Then, the collected constraints are negated and solved with a constraint solver such as SMT solvers; the solutions are mapped to new inputs that exercise different program execution paths. This approach can automatically discover and test corner cases that lead to security vulnerabilities.

There are also other works that automate the test case generation process based on constraints or criteria. Li et al. [2012] proposed a framework called GKLEE that can analyze C++ programs to identify bugs that affect the performance of GPU. Krichen and Tripakis [2004] and Larsen et al. [2005] proposed a framework for black-box conformance testing of real-time systems, where specifications are abstracted as timed automata with different definitions of conformance relations between the specifications and implementations.

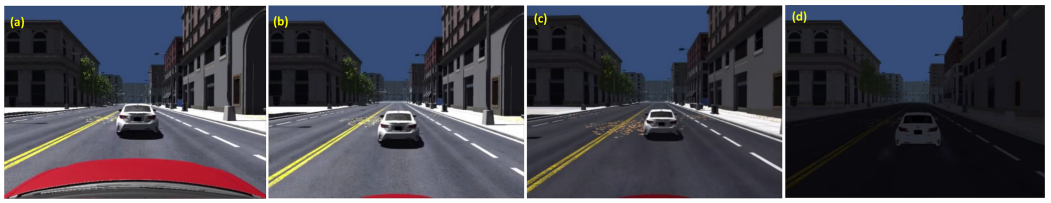

Fig. 13. Example manipulation of the virtual images to test the front vehicle detection (the image is from our prior work [Kim et al. 2017]).

6.2 Discussion: Test Generation for Machine Learning Component

Even though the proposed test specification can express the geometric and behavioral aspects of the test environment, there are also other environmental aspects that are important to be automated. In particular, we introduce how our test framework can be applied and extended to automatically generate the test environment for a Machine Learning (ML) component of the automotive system.

An ML component plays an important role in modern vehicles to implement various safety features in combination with other sensors. The role of the ML component is to classify the context of the road environment from a series of road images taken by an on-board camera, such as pedestrian crossing, traffic signal changes, and so on; those contexts are used to determine if a safety feature shall be activated or not under a certain driving condition (e.g., activate a hard braking to avoid a collision with a pedestrian). Hence, it is important to have a sufficient classification accuracy of the ML component to be used for the safety features.

The development process of an ML component is typically divided into two phases: the training phase and the testing phase. The training phase is to construct an ML model from a labeled training data set, while the test phase is to validate if the constructed ML model has an expected accuracy from a test data set. Those data sets have been typically obtained from real-world images. However, the virtual images generated from the simulation tools (e.g., game engines) have great potential for training and testing an ML model as well, and some industries already adopt this concept in their development process [Madrigal 2017].

Engineers can generate a massive amount of virtual images in a controlled way in the virtual environment. For example, one can create a scene where the pre-collision system is to be activated to avoid a collision with a front vehicle as illustrated in Figure 13. From this scene, one can generate a series of camera images by varying the angles (a, b) and illuminances (c, d) where the pre-collision system should be activated; note that those generated images can also be automatically labeled with some programming effort in the simulation environment. To extend the concept of the proposed test specification and generation in terms of the ML component, the research question is how to express and generate the virtual road environment from which a large number of road images can be taken for training and testing ML components. For example, one can formulate the constraints that affect the accuracy of the ML component such as camera angles, illuminances, reflection rates, partially invisible lanes/traffic signs, or weather conditions (e.g., rainy or foggy areas). Even though those constraints would be expressed in different types of the parametric combinations from the one used for the path and behavior specification, we believe that our proposed test generation concept can be similarly applied to generate road environments for training and testing ML components.

7 CONCLUSIONS

In the automotive CPS, engineers are facing test challenges to show the correctness, performance, or effectiveness due to the high complexity originating from autonomous and connected features.

Traditional system-level tests have been performed in a physical world that is difficult to control by engineers. In addition, we observe that many international test standards seem to be designed assuming such a physical road environment; consequently, this seems to simplify the scope of test environment due to the inherent limitation of physical test. To test more complex vehicle features, we believe that the proposed virtual test can complement the physical test in many ways, since it is free from many limitations of the physical test. By utilizing such a less-restrictive environment, our test framework allows road environments to be generated in a more rigorous and efficient manner compared to the manual way. This ensures the generated environment conforms to the test specification, consequently providing engineers with higher confidence on the test results.

We consider two future research directions. One direction is to formalize constraints of the geometric design for more complex objects, such as merging, intersections, or roundabouts. Having such richer templates of constraints will enable engineers to use this test generation framework for a wide range of applications. Depending on the vehicle features, there may be other coverage types that are of interest. Another direction is to develop more test coverage criteria by studying such additional vehicle features. We believe that the aforementioned additional research will improve the usability of the test generation framework.

REFERENCES

Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theor. Comput. Sci.* 126, 2 (1994), 183–235. DOI: https://doi.org/10.1016/0304-3975(94)90010-8

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*.

A. En-Nouaary, R. Dssouli, and F. Khendek. 2002. Timed Wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.* 28, 11 (Nov. 2002), 1023–1038. DOI: https://doi.org/10.1109/TSE.2002.1049402

Eric Galin, Adrien Peytavie, Nicolas Maréchal, and Eric Guérin. 2010. Procedural generation of roads. *Comput. Graph. For.* 29, 2 (2010), 429–438.

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages.

Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. 2008. Testing real-time systems using UPPAAL. *Formal Methods and Testing*. Springer-Verlag, Berlin, 77–117. Retrieved from: http://dl.acm.org/citation.cfm?id=1806209.1806212.

Texas Transportation Institute and Southwest Region University Transportation Center (U.S.). 2011. *Automated Generation of Virtual Scenarios in Driving Simulator from Highway Design Data*. https://rosap.ntl.bts.gov/view/dot/23113.

ISO15622. 2018. Intelligent transport systems—Adaptive Cruise Control systems—Performance requirements and test procedures. https://www.iso.org/standard/71515.html.

ISO17387. 2008. Intelligent transport systems—Lane change decision aid systems (LCDAS)—Performance requirements and test procedures. https://www.iso.org/standard/43654.html.

ISO19237. 2017. Intelligent transport systems—Pedestrian detection and collision mitigation systems (PDCMS)—Performance requirements and test procedures. https://www.iso.org/standard/64111.html.

Song Jian and Jin Ruichen. 1999. Generation of virtual road surfaces and simulation of nonlinear vibration of vehicles. In *Proceedings of the Vehicle Electronics Conference (IVEC'99)*.

Felipe Jiménez, Francisco Aparicio, and Gonzalo Estrada. 2009. Measurement uncertainty determination and curve-fitting algorithms for development of accurate digital maps for advanced driver assistance systems. *Transport. Res. Part C: Emerg. Technol.* 17, 3 (2009), 225–239.

R. J. Kavanagh and R. Ramanathan. 1982. Computer simulation of road surface profiles for a four-wheeled vehicle. In *Proceedings of the 14th Conference on Winter Simulation—Volume 1*.

B. Kim, A. Jarandikar, J. Shum, S. Shiraishi, and M. Yamaura. 2016. The SMT-based automatic road network generation in vehicle simulation environment. In *Proceedings of the International Conference on Embedded Software (EMSOFT'16)*. 1–10. DOI: https://doi.org/10.1145/2968478.2968498

B. Kim, Y. Kashiba, S. Dai, and S. Shiraishi. 2017. Testing autonomous vehicle software in the virtual prototyping environment. *IEEE Embed. Syst. Lett.* 9, 1 (Mar. 2017), 5–8. DOI: https://doi.org/10.1109/LES.2016.2644619

Moez Krichen and Stavros Tripakis. 2004. Black-box conformance testing for real-time systems. In *Proceedings of the 11th International SPIN Workshop: Model Checking Software*. Springer Berlin, 109–126.

Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. 2005. Online testing of real-time systems using Uppaal. In *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing (FATES'04)*. Springer Berlin, 79–94.

Kim G. Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *Int. J. Softw. Tools Technol. Trans.* 1, 1 (1997), 134–152.

Kim G. Larsen and Wang Yi. 1994. *Time Abstracted Bisimulation: Implicit Specifications and Decidability*. Springer Berlin, 160–176. DOI: https://doi.org/10.1007/3-540-58027-1_8

Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic verification and test generation for GPUs. *SIGPLAN Not.* 47, 8 (Feb. 2012), 215–224.

A. Madrigal. 2017. Inside Waymo's secret world for training self-driving cars. *The Atlantic,* Aug. 23 (2017). Retrieved from: https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/.

Euro NCAP. 2017. *European New Car Assessment Programme (Euro NCAP)—Test Protocol—AEB VRU systems*. Technical Report.

Federal Highway Administration Office. 2009. *Pedestrian Safety at Intersections*. Technical Report.

Corina S. Păsăreanu and Willem Visser. 2009. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Trans.* 11, 4 (2009), 339–353.

Jan Peleska, Elena Vorobev, and Florian Lapschies. 2011. Automated test case generation with SMT-solving and abstract interpretation. In *Proceedings of the NASA Formal Methods Symposium (NFM'11)*.

Stefan Ratschan. 2006. Efficient solving of quantified inequality constraints over the real numbers. *ACM Trans. Comput. Log.* 7, 4 (2006), 723–748.

Mikio Yanagisawa, Elizabeth D. Swanson, Philip Azeredo, and Wassim Najm. 2017. *Estimation of Potential Safety Benefits for Pedestrian Crash Avoidance/Mitigation Systems (Report No. DOT HS 812 400)*. Technical Report. John A. Volpe National Transportation Systems Center U.S. Department of Transportation. https://rosap.ntl.bts.gov/view/dot/12475/dot_12475_DS1.pdf.

S. Zhao and J. A. Farrell. 2011. Optimization-based road curve fitting. In *Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference*.

Received September 2017; revised August 2018; accepted February 2019