Compiling OpenSCENARIO 2.1 for Scenario-Based Testing in CARLA

Thoshitha Gamage
Department of Computer Science
Southern Illinois University Edwardsville
Edwardsville, IL
tgamage@siue.edu

Lasanthi Gamage
Department of Computer Science
Webster University
Webster Groves, MO
lasanthigamage67@webster.edu

Abstract

While the ASAM OpenSCENARIO 2.1 Domain-Specific Language (DSL) enables declarative, intent-driven authoring for Scenario-Based Testing (SBT), its integration into open-source simulators like CARLA remains limited by legacy parsers. We propose a multi-pass modern compiler architecture that translates the OpenSCENARIO 2.1 DSL directly into executable CARLA behaviors. The pipeline features an ANTLR4 frontend for Abstract Syntax Tree (AST) generation, a semantic middle-end, and a runtime backend that synthesizes deterministic `py_trees` *behavior trees. Mapping the standardized domain ontology directly to CARLA's procedural API via a custom method registry eliminates the need for external logic solvers. A demonstrative multi-actor cut-in and evasive maneuver, selected from a wider suite of validated scenarios, confirms the compiler's ability to process concurrent actions, dynamic mathematical expressions, and asynchronous signaling. This framework establishes a functional baseline for reproducible, large-scale SBT, paving the way for future C++ optimizations to mitigate current Python-based computational overhead.*

1. Introduction

The transition toward highly automated and autonomous vehicles necessitates a paradigm shift in verification and validation methodologies. Relying on traditional, distance-based field tests to uncover safety-critical edge cases is economically infeasible and statistically insufficient [10]. Consequently, the industry has widely adopted Scenario-Based Testing (SBT), a paradigm that accelerates evaluation by systematically isolating and simulating complex traffic interactions within high-fidelity virtual simulation environments like CARLA [6].

SBT demands a standardized, machine-readable language capable of describing these interactions with both precision and flexibility. Historically, the industry relied on ASAM's (Association for Standardization of Automation and Measuring Systems) *de facto* standard for interoperability: ASAM OpenSCENARIO XML 1.x [1]. While this XML schema is optimized for defining predictable trajectory playback, it forces a trade-off between expressive flexibility and maintainability as scenario complexity grows. To address this, ASAM bifurcated the standard by releasing ASAM OpenSCENARIO Domain Specific Language (DSL) standard. The first version (v2.0) was released in July 2020 and a more mature and a comprehensive version (v2.1) was released in March 2024 [2][1]. The DSL standard introduces a human-readable, declarative syntax that enables intent-driven scenario authoring. It allows researchers to express complex maneuvers as reusable, parameterized logic across abstract, logical, and concrete layers, significantly reducing boilerplate. Table 1 provides a comparison between these two standards to highlight this point.

Integrating the OpenSCENARIO DSL into established open-source engines, nonetheless, has revealed a substantial implementation gap. Widely used orchestration frameworks, such as `ScenarioRunner` [5], which operate on top of simulation engines like CARLA, historically provided only partial support. Furthermore, most existing implementations lack a robust compiler for the finalized OpenSCENARIO v2.1 specification. Within these frameworks, the legacy interpreters utilize brittle parsing architectures that trigger syntax errors when attempting to ingest the bifurcated, finalized standard libraries (`types.osc` and `domain.osc`). The net result is the inability to leverage the full expressive power of the finalized DSL for research and industrial-grade scenario validation.

This paper addresses this bottleneck by presenting a fully compliant OpenSCENARIO v2.1 compiler architecture integrated directly with the CARLA/`ScenarioRunner` environment. Our work utilizes ANTLR4 [13] to implement an

---

[1] Since finalizing OpenSCENARIO v2.1 standard, ASAM continues to refine the DSL, currently fielding public reviews for the 2.2.0 release candidate as of early 2026. However, v2.1 remains the foundational finalized library supported by this architecture.

Table 1. Comparison of OpenSCENARIO v1.3 (XML) and v2.1 (DSL)

Feature	OSC 1.3 (XML)	OSC 2.1 (DSL)
Format	Schema-based (XML)	Text-based DSL (EBNF)
Hierarchy	Storyboard → Act → Maneuver → Event → Action	Declarative, intent-driven structure
Logic	Trigger-based conditions	Full expressions, conditions, and logic
Parameterization	Parameters and catalogs	Native variables and reusable abstractions
Modularity	Limited reuse via catalogs	High modularity and composability
Behavior Modeling	Explicit action definitions	Actions + Modifiers + Conditions separation
Semantic Validation	Syntax (XSD) only	Supports ontology-based semantic validation
Readability	Verbose (∼150–300 lines)	Concise (∼20–50 lines)
Integration	Widely supported in tools	Requires DSL parser/compiler, limited support
Execution Model	Event-driven storyboard execution	Intent-driven, composable execution model
Extensibility	Limited	High (DSL + custom constructs)

EBNF-driven compiler. This multi-stage system generates an Abstract Syntax Tree (AST) that ensures type safety and semantic validity, ultimately mapping domain modifiers to CARLA's atomic behaviors through deterministic Behavior Trees (`py_trees`). The following sections detail a three-stage compiler architecture and outline the simulation ontology features supported by the implementation. Furthermore, a complete scenario file is provided to demonstrate a functional example of these features in practice.

2. Related Work

The OpenSCENARIO DSL formalization joins an expanding body of research into domain-specific languages for autonomous driving. It complements existing frameworks like the probabilistic Scenic [7], the map-centric GeoScenario [14], and testing-focused DSLs such as Paracosm [12]. These approaches offer complementary strengths, including stochastic scenario generation, high-fidelity geographic modeling, and parameterized test case design. However, they are typically developed as standalone frameworks and often lack standardized semantics, limiting interoperability across tools and simulation platforms.

In contrast, the OpenSCENARIO DSL aim to unify scenario representation through a declarative DSL that integrates *spatial*, *temporal*, and *behavioral* semantics within a single framework. The OpenSCENARIO v2.1 standardizes many of these concepts, introducing composition operators for orchestrating complex multi-actor interactions and enabling the specification of Key Performance Indicators (KPIs) directly within scenario definitions [2]. Furthermore, it is designed to interoperate with complementary standards in the ASAM OpenX ecosystem, such as OpenDRIVE and OpenLABEL, and can be integrated with simulation ontologies to support semantic reasoning about physical feasibility and domain-specific constraints.

Parsing OpenSCENARIO DSL grammar into an Abstract Syntax Tree (AST) relies heavily on modern ANTLR4-based lexical analyzers, most notably the community-driven *py-osc2* framework [11]. Once parsed, translating this declarative logic into continuous simulation execution is a critical bottleneck. RoadLogic [3] proposes translating the OpenSCENARIO DSL into a symbolic automaton representation, utilizing Answer Set Programming (ASP) to solve high-level planning constraints before passing them to a motion planner. While logically rigorous, this approach introduces the computational overhead of an external logic solver.

Similarly, the Yase framework [9] explores a multi-stage compiler architecture that maps an AST to a behavior tree, but functions primarily as an agnostic middle-end integrated into the openPASS ecosystem. Other recent frameworks, such as VIVAS [8] and BeSimulator [16], have demonstrated the viability of utilizing *py_trees* for managing complex agent perception and action states dynamically in simulators like CARLA. In contrast to architectures requiring intermediate external solvers, our approach directly compiles generated AST into execution-ready *py_trees* natively within CARLA, allowing for real-time, tick-by-tick simulation by mapping the `domain.osc` actions to CARLA's localized Python API.

Recent literature also highlights the use of Large Language Models (LLMs) to automatically generate DSL scripts from natural language. Frameworks such as Text2Scenario [4] utilize models like GPT-4 to parse scenario descriptions and output OpenSCENARIO DSL scripts intended for simulators like CARLA. Similarly, platforms like Chat2Scenario [17] and top-down generation pipelines such as LeGEND [15] demonstrate the capability of LLMs to transform unstructured traffic data, crash reports, and abstract functional scenarios into formal logical parameters. However, these generative approaches focus entirely on the frontend creation of the scenario text, fundamentally assuming a functioning backend compiler exists to execute the output script. Our architecture provides the underlying execution engine required to make such LLM-driven generation pipelines practically viable.

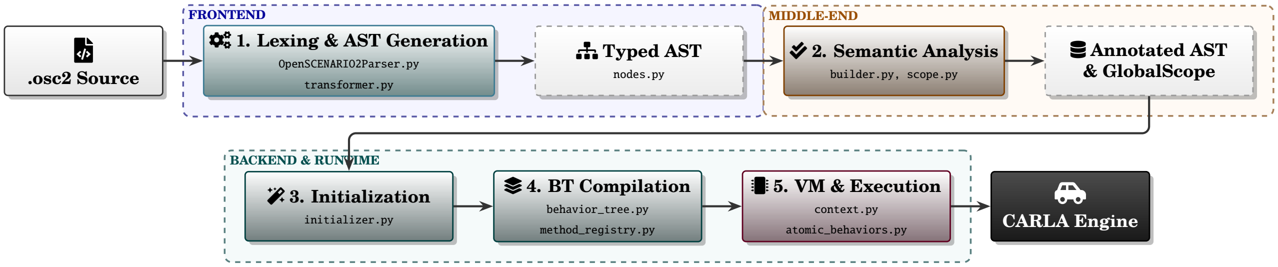

Figure 1. The 3-stage OSC2 Execution Pipeline mapping DSL definitions to CARLA behaviors.

3. New Compiler Architecture

Despite aforementioned advances, practical adoption of DSLs remains limited by a lack of robust tooling for formal parsing, semantic validation, and simulation integration. To bridge this gap, we introduce a multi-pass compiler architecture that translates OpenSCENARIO v2.1 *(hereafter OSC2)* into executable CARLA behaviors, providing a seamless workflow from standard-compliant authoring to simulation. As illustrated in Figure 1, the pipeline is structured around three primary compilation stages – Frontend Parsing, Semantic Analysis, and Backend Execution Generation – supported dynamically by a Runtime Context Manager. This decoupled approach ensures strict compliance with the OSC2 domain model while allowing flexible mapping to the underlying physics and navigation engines.

3.1. Lexing and AST Generation

The first stage of the compilation pipeline is the Frontend, responsible for lexical and syntax analysis. ANTLR4 is utilized to process the raw OSC2 source code against the standard Extended Backus-Naur Form (EBNF) grammar, producing a raw parse tree.

To map the concrete syntax to the internal logic of the compiler, an `ASTTransformer` applies the Visitor pattern to the parse tree, generating a typed Abstract Syntax Tree (AST). The AST is implemented utilizing strongly-typed data structures to ensure compile-time type safety. At this stage, the transformation is purely syntactic; no scoping, type checking, or symbol resolution is performed. This guarantees that the AST accurately represents the syntactic structure of the source file independently of its semantic validity.

3.2. Semantic Analysis and Symbol Resolution

Once generated, the syntactic AST is passed to the `ModelBuilder` for Semantic Analysis. Because the OSC2 language supports hierarchical ontological structures—such as actor inheritance (e.g., `vehicle` inheriting from `traffic_participant`), physical type definitions, and forward declarations—the semantic analyzer operates via a two-pass methodology:

1. **Definition Pass:** The compiler traverses the AST to populate the Symbol Table. It constructs a global scope alongside nested lexical scopes for namespaces, structures, and actors. All named entities (variables, methods, actions, and modifiers) are registered as abstract symbols.
2. **Resolution Pass:** The compiler traverses the AST a second time to bind symbols to their corresponding type definitions, resolve inheritance chains, and validate constraints. During this phase, implicit fallbacks to standard library domains (e.g., the `stdtypes` namespace from `types.osc`) are safely resolved.

Upon successful completion of these passes, the AST is considered semantically valid, annotated with scope contexts, and ready for runtime evaluation.

3.3. Initialization, BT Compilation, and Execution

The final stage of the pipeline translates the semantically validated AST into an executable format. Rather than generating static bytecode, the compiler backend synthesizes a dynamic Behavior Tree (BT) utilizing the `py_trees` framework. This generation is orchestrated through two primary components:

- **Behavior Tree Builder:** This module acts as the code generator, mapping OSC2 control flow directives (`do`, `serial`, `parallel`, `one_of`) into corresponding Behavior Tree composites. It wraps durative actions with timeout constraints and maps OSC2 event triggers to native condition checkers (e.g., edge detection for `rise`/`fall` expressions).
- **Method Registry:** To decouple the OSC2 language ontology from the CARLA API, a decorator-based `MethodRegistry` dynamically dispatches abstract AST actions (e.g., `vehicle.drive()`) to concrete, atomic Python behaviors (e.g., `WaypointFollower`, `ChangeTargetSpeed`). This registry centralizes the resolution of universal movement constraints, ensuring consistent application of collision avoidance and traffic rule

adherence across all generated behaviors.

Runtime Context Management: Because certain parameters, such as relative speeds or dynamically calculated lane offsets, cannot be fully resolved at compile time, the backend relies on an `ExecutionContext`. Functioning as a lightweight Runtime State Manager, it recursively evaluates AST expression nodes during execution. By implementing $\mathcal{O}(1)$ caching mechanisms for live actor lookups and real-time physical unit conversions, declarative statements can be continuously re-evaluated as mathematical primitives within the simulation loop.

Finally, before the primary Behavior Tree is ticked, a `ScenarioInitializer` parses the AST for constraints designated with the `at: start` modifier. It calculates spatial dependencies and executes the lazy-spawning or teleportation of actors, ensuring the concrete simulation state strictly aligns with the declarative initial conditions of the OSC2 script.

4. Simulation Ontology & Execution Mapping

Executing a OSC2 scenario requires mapping the corresponding domain model/simulation ontology (`domain.osc`) to discrete simulation commands. This is achieved by binding the abstract ontology directly to the CARLA API via the intermediate atomic behaviors registered within the `MethodRegistry` [2, 6].

To define the exact scope of this integration, Table 2 provides a comprehensive checklist of the supported actions, modifiers, and condition semantics successfully implemented within the compiler framework.

4.1. Execution of Actions and Modifiers

To satisfy the declarative constraints of OSC2 DSL, core movement actions (e.g., `vehicle.drive()`, `person.walk()`) are systematically mapped to underlying atomic behaviors. The runtime evaluation engine dynamically delegates parameters to these behaviors based on the active modifiers present in the AST.

As detailed in Table 2, the compiler handles two primary categories of dynamic modifiers to execute these actions:

- **Kinematic Modifiers:** Constraints such as `speed`, `change_speed`, and `acceleration` calculate target scalar velocities or force profiles. These are executed via continuous dynamic adjustments using Proportional-Integral-Derivative (PID) control, allowing for relative evaluations (e.g., `faster_than`) against other simulated entities.
- **Spatial Modifiers:** Constraints such as `position`, `lane`, `keep_lane`, and `change_lane` utilize OpenDRIVE standard logic and topological map projections. This enables the localized planner to calculate lateral offsets, dynamic splines, and target waypoints while evaluating collision avoidance and traffic light compliance.

This explicit mapping supports a comprehensive matrix of atomic behaviors, effectively transforming declarative OpenSCENARIO constraints into continuous, tick-by-tick waypoint tracking, velocity control, and environmental state loops.

5. Case Study: Dynamic Cut-In and Evasive Maneuver

To validate the expressiveness of the developed compiler architecture and its adherence to the standard, a comprehensive multi-actor test scenario[2] is presented and evaluated herewith. In this particular case, the scenario specifies an adversarial interaction wherein a Heavy Goods Vehicle (HGV) rapidly overtakes and cuts in front of an *ego* vehicle (the "*hero*"). The HGV subsequently performs a sudden deceleration *(a.k.a. brake check)*, forcing the *ego* vehicle to execute a concurrent evasive lane change and visual warning (flashing high beams). Finally, both vehicles synchronize their deceleration to stop safely before a static obstacle.

This scenario tests the runtime's ability to evaluate dynamic mathematical expressions, continuous spatial queries, and multi-actor event synchronization.

5.1. Declarative Initialization and Topology

The scenario script begins by defining the ontological entities, resolving dynamic state variables, and projecting the actors onto the topological road network.

```
import "domain.osc"
use std.stdtypes

scenario hello_world:
  # --- DEFINITIONS ---
  carla_map: map with:
    keep(it.map_file == "Town06")

  env: environment

  hero: vehicle with:
    keep(it.model == "vehicle.tesla.model3")
    keep(it.name == "hero")

  npc: vehicle with:
    keep(it.model == "vehicle.carlamotors.
     european_hgv")
    keep(it.name == "npc")
    keep(it.color == "0,128,0")

  obstacle: stationary_object with:
    keep(it.name == "obstacle")
    keep(it.model == "static.prop.trafficwarning")

  # --- GLOBAL VARIABLES ---
  var v_hero: speed = 35kph
  var v_npc_fast: speed = v_hero + 12.42mph
  var v_npc_slow: speed = v_hero - 10kph
  var v_npc_catchup: speed = v_hero * 10kph

  var lag: length = 5m
  var gap: length = lag * 3
  var safety_gap: length = gap - 3m

```

---

[2] A video demonstration of this scenario execution is available at: https://youtu.be/XrHTOlMSTpg

Table 2. Extended capability checklist aligned with OCS2, detailing supported actions, modifiers, and condition semantics within the implementation.

Category	Capability	Description	Implementation Method	Supported
Action Framework	Action lifecycle	Start, end, fail states	Behavior Tree status tracking	✓
	Actor binding	Map actions to actors	Runtime execution context	✓
	Composition	Sequential/parallel	Tree composites (serial/parallel)	✓
	Duration handling	Time bounds	Parallel timeout wrappers	✓
Movement	Move/drive/walk	Motion primitives	LocalPlanner/WaypointFollower	✓
	Speed control	Adjust speed	PID longitudinal control	✓
	Acceleration control	Adjust acceleration	Direct physics/PID application	✓
	Stationary	Hold position	Zero-velocity kinematic lock	✓
Position & Path	Assign position	Set location	Absolute spatial transforms	✓
	Assign orientation	Set rotation	Rotational math application	✓
	Follow path	Move along path	Spline interpolation routing	✓
	Follow trajectory	Move along trajectory	Time-parameterized tracking	✓
Interaction	Time gap	Time-based spacing	Dynamic setpoint interpolation	✓
	Space gap	Distance-based spacing	Topological route tracing	✓
	Headway	Relative positioning	Vector projections	✓
Environment	Weather control	Change weather	Direct simulation API	✓
	Traffic signals	Control traffic lights	Semantic/Group state dispatch	✓
	Road conditions	Change surface/state	Friction trigger spawning	✓
Modifiers	Speed modifier	Velocity profile	Dynamic profile parsing	✓
	Acceleration modifier	Acceleration profile	PID interpolation	✓
	Position modifier	Spatial constraints	Initialization placement context	✓
	Lateral modifier	Lane/side shift	OpenDRIVE lane offsets	✓
	Physical movement	Physics toggle	Simulation engine override	✓
	Temporal modifiers	Time limits/delays	Elapsed time evaluation	✓
	Relative modifiers	Relative offsets	Dynamic object referencing	✓
	Orientation modifiers	Rotation offsets	Yaw/Pitch/Roll application	✓
Coordinate Systems	World coordinates	Global frame	Cartesian (x-y-z) transformations	✓
	Relative coordinates	Actor frame	Reference-based projection	✓
	Route-based (s-t)	Road frame (s-t)	OpenDRIVE coordinate mapping	✓
Scenario Composition	Serial execution	Ordered steps	Sequence node construction	✓
	Parallel execution	Concurrent steps	Parallel node synchronization	✓
	Conditional triggers	Event-driven execution	Blackboard signal evaluation	✓
Extensibility	Custom actions	Custom behavior	MethodRegistry decorators	✓
	Conflict resolution	Handle conflicts	Blackboard arbitration logic	✓
	Semantic validation	Check logic	AST resolution pass	✓
Condition & Expression	one_of	Pick one value	`SuccessOnOne` tree policy	✓
	rise	False → True	Edge detection condition	✓
	fall	True → False	Edge detection condition	✓

```
  do serial:
    # --- TWILIGHT SETTINGS ---
    env.assign_celestial_position(azimuth: 270deg,
     elevation: 12deg)
    hero.set_lights(mode: "auto")

    hero.assign_position() with:
      lane(1, at: start)
      speed(0kph, at: start)

    npc.assign_position() with:
      lane(side: right, side_of: hero, at: start)
      position(distance: lag, behind: hero, at:
    start)
      speed(0kph, at: start)

    obstacle.assign_position() with:
```

```
49        position(x: 478.93, y: -14.07, z: 0.00, h: -1.57rad, at: start)
```

Listing 1. OSC2 Declarations, Global Variables, and Initialization Phase.

As shown in Listing 1, the `ExecutionContext` processes relative assignments (e.g., `v_npc_fast = v_hero + 12.42mph`) prior to execution. Leveraging `stdtypes` definitions, the compiler recursively evaluates and converts mixed dimensional quantities (e.g., mph, kph) into CARLA's native SI metric units (m/s, m) before passing values to the simulation engine.

During initialization (`at: start`), the `ScenarioInitializer` executes dynamic actor placement across three instantiation paradigms:

- **Default Map:** Generic constraints (e.g., `hero`) query the CARLA API for predefined valid spawn points, ensuring safe default placement *(lines 39–41)*.
- **Relative Topological:** The `npc` is spawned relative to `hero` by resolving the OpenDRIVE spline, applying lateral lane offsets, and projecting backward by the `lag` distance *(lines 43–46)*.
- **Absolute Cartesian:** Static props like the `obstacle` bypass topological tracing, using explicit world coordinates *(lines 48–49)*.

To reliably track entities, the compiler binds the OSC2 `keep(it.name == "...")` constraint to CARLA's `role_name` attribute. This deterministic mapping is critical for ROS 2 integration, enabling the `Ros2SubscriberManager` to automatically configure sensor namespaces (e.g., `/carla/hero/imu_sensor`) and control topics without manual overhead.

Furthermore, the `environment` pseudo-actor declaratively controls the weather. The `assign_celestial_position` action translates azimuth and elevation into CARLA sun angles. As altitude drops, the `MethodRegistry` detects the shift and automatically activates the ego vehicle's dynamic lighting, linking global conditions to atomic behaviors. Crucially, this `env` declaration scales beyond initialization, allowing dynamic mid-simulation injections of precipitation, fog, or time-of-day progression for multi-phase meteorological testing.

Figure 2 illustrates the resulting state, highlighting accurate relative placement and twilight environmental parameters.

5.2. Ego Vehicle (Hero) Execution Logic

Following initialization, the runtime transitions to the main execution blocks. The ego vehicle's behavior tree demonstrates continuous state evaluation and parallel execution capabilities.

```
50  do parallel:
51    # ==========================
52    # --- HERO LOGIC ---
53    # ==========================
54    serial:
55      wait @go_signal
56
57      # --- Phase 1: Cruise ---
58      one_of:
59        hero.drive() with:
60          speed(v_hero)
61        wait fall(npc.position.ahead_of(hero) > safety_gap)
62
63      # --- Phase 2: Hazard Detected - Flash High Beams & Swerve ---
64      parallel:
65        # Task A: The evasive physical maneuver
66        serial:
67          hero.change_lane(num_of_lanes: 1, side: right)
68          emit CRASH_AVOIDED
69
70        # Task B: The visual "Flash" effect
71        serial:
72          hero.set_lights(mode: "high_beam")
73          wait elapsed(0.5s)
74          hero.set_lights(mode: "auto")
75
76      # --- Phase 3: Approach & Synchronize ---
77      one_of:
78        hero.drive() with:
```

(a) Default spectator view depicting the relative longitudinal and lateral offsets.

(b) Frontal perspective illustrating the twilight environmental conditions and automatically engaged daytime running lights.

Figure 2. Simulation state following the declarative initialization phase, captured from multiple perspectives.

```
        speed(v_hero)
      wait @OBSTACLE_DETECTED

    # --- Phase 4: Approach NPC smoothly and pull
    alongside ---
    serial:
      hero.change_speed(target: 25kph,
  rate_profile: smooth)
      one_of:
        hero.drive() with:
          speed(25kph)
        serial:
          wait rise(hero.position.ahead_of(npc)
  >= -1m)

    # --- Phase 5: Final Stop ---
    hero.change_speed(target: 0kph, rate_profile:
    asap)
    wait hero.speed < 0.1kph
    wait elapsed(5s)
```

Listing 2. Ego Vehicle (Hero) behavior logic detailing continuous queries and parallel execution.

Listing 2 highlights the Condition and Expression layer. In Phase 1, the `hero` executes `drive()` within a `one_of` composite alongside a `wait` directive. The `fall(...)` argument forces the runtime to continuously query topological network distances, triggering an edge detector that terminates the drive action and transitions to Phase 2 when the threshold is crossed.

Phase 2 demonstrates the `parallel` composite. The `BehaviorTreeBuilder` generates a concurrent node combining a physical `change_lane` maneuver with a simultaneous vehicle light state update (flashing high beams). An internal `CRASH_AVOIDED` event is emitted as a synchronization flag for the adversarial actor. Figure 3 captures this exact tick, showing both the exterior lateral trajectory and lighting (Figure 3a) alongside the interior visual feedback (Figure 3b).

Phase 3 highlights decoupled synchronization via a blackboard architecture. The `wait @OBSTACLE_DETECTED` directive suspends the sequence until a global flag is raised, permitting the `hero` to cruise indefinitely until an external trigger broadcasts the signal. This facilitates complex inter-actor orchestration without hardcoded temporal dependencies.

Phase 4 illustrates advanced kinematic control. The `MethodRegistry` maps the `rate_profile: smooth` modifier to a highly damped PID controller within the `ChangeTargetSpeed` atomic behavior, ensuring realistic, jerk-limited deceleration. Concurrently, a rising edge condition (`wait rise(...)`) monitors relative spatial queries dynamically to detect when the ego vehicle pulls alongside the `npc`.

Phase 5 concludes with an `asap` speed profile, bypassing smoothed PID gains for maximum braking. The subsequent `wait hero.speed < 0.1kph` directive demonstrates direct member access: the `ExecutionContext` fetches the vehicle's 3D velocity vector per-tick, computes its scalar magnitude, and evaluates the relation. The sequence terminates cleanly via a non-blocking timeout composite generated by `wait elapsed(5s)`.

(a) Exterior perspective capturing the exact moment the ego vehicle initiates the evasive right lane change and activates its high beams.

(b) Driver's point-of-view (POV) at the identical simulation tick, illustrating the visual warning effect from within the cabin.

Figure 3. Phase 2 execution: The ego vehicle concurrently executing a lateral maneuver and a visual warning in response to the spatial trigger.

5.3. Adversarial (NPC) Execution Logic

The adversarial logic executes concurrently with the ego vehicle, relying on dynamic speed interpolation and cross-actor event synchronization.

```
# ==========================
# --- NPC LOGIC ---
# ==========================
serial:
  wait @go_signal

  # --- Phase 1: Accelerate UNTIL ahead of Hero
   ---
  one_of:
    npc.drive() with:
      speed(v_npc_fast)
    serial:
      wait rise(npc.position.ahead_of(hero) >=
   lag * 2)

  # --- Phase 2: Cut In ---
  npc.change_lane(num_of_lanes: 1, side: left)

  # --- Phase 3: Brake Check UNTIL Ego swerves
   ---
  one_of:
    npc.drive() with:
        speed(v_npc_slow)
```

```
      wait @CRASH_AVOIDED

    # --- Phase 4: Recover & Approach Obstacle ---
    one_of:
      npc.drive() with:
          speed(v_npc_catchup, rate_profile: smooth)
      serial:
          wait rise(npc.object_distance(reference: obstacle, direction: euclidean) < 45m)

    # --- Phase 5: Emergency Stop & Emit Event ---
    npc.change_speed(target: 0kph, rate_profile: asap)
    emit OBSTACLE_DETECTED

    # Phase 6: Parked
    wait elapsed(100s)
```

Listing 3. NPC behavior logic demonstrating event synchronization and Euclidean distance queries.

(a) Frontal perspective capturing the synchronized halt of the ego vehicle and the HGV before the static obstacle.

(b) Bird's-eye view confirming the final lateral alignment and longitudinal gap maintenance at the scenario's conclusion.

Figure 4. Phase 5 execution: Both actors synchronize their deceleration profiles to halt safely, demonstrating precise spatial and temporal control.

Listing 3 details the adversarial `npc` logic. Building upon the constructs introduced in the ego vehicle's routine, it highlights dynamic expression evaluation, bidirectional synchronization, and versatile spatial queries.

In Phase 1, the `one_of` composite evaluates a topological distance query against a dynamically computed threshold (`lag * 2`). This demonstrates the `ExecutionContext`'s ability to resolve mathematical expressions continuously at runtime, bypassing the need for static literals. Phase 2 then executes a standard lateral cut-in maneuver.

Phases 3 and 5 showcase asynchronous, cross-actor orchestration. The `npc` maintains a deliberate deceleration until receiving the `CRASH_AVOIDED` signal from the ego vehicle's evasive maneuver. Conversely, the `npc`'s final emergency stop emits the `OBSTACLE_DETECTED` event to release the ego vehicle from its suspended cruise. This bidirectional event handshaking achieves complex synchronization without rigid temporal scripting.

Phase 4 introduces a critical distinction in spatial resolution. While re-applying the previously discussed smoothed PID profile, it executes an `object_distance` query with the `direction: euclidean` parameter. Unlike topological `ahead_of` queries that trace OpenDRIVE road splines, this instructs the `MethodRegistry` to compute a direct 3D Cartesian distance. This mechanism is essential for detecting static props, such as the `obstacle`, which reside outside the bounds of the routable road network. Finally, Phase 6 utilizes a basic `elapsed` directive to indefinitely suspend the actor's execution tree.

6. Discussion and Future Work

This paper presented a multi-pass compiler architecture mapping the declarative OpenSCENARIO v2.1 DSL to the procedural CARLA simulator. By decoupling parsing and semantic analysis from physics execution, the framework translates abstract intent into deterministic behavior trees. The dynamic runtime – comprising the `ExecutionContext` and `MethodRegistry` – enables real-time mathematical evaluation, continuous spatial queries, and multi-actor synchronization without external solvers or static bytecode.

Despite the baseline established, the architecture presents distinct limitations. First, while core kinematic and spatial modifiers are integrated, the expansive OSC2 ontology requires further expansion to cover complex domains like pedestrian intent, intersection right-of-way, and probabilistic weather. Second, continuously evaluating topological and Euclidean queries within a Python-based loop introduces computational overhead, potentially degrading real-time fidelity in high-density traffic scenarios.

Future work will prioritize architectural optimization and further standard compliance. To mitigate latency, expensive spatial queries will be migrated to lower-level C++ CARLA bindings. Additionally, `MethodRegistry` will be expanded to achieve complete ontological coverage and adapt the semantic analyzer for the upcoming OpenSCENARIO v2.2.0 syntax, ultimately facilitating massive-scale, reproducible scenario-based testing.

References

[1] ASAM. ASAM OpenSCENARIO XML, 2020. 1

[2] ASAM. ASAM OpenSCENARIO DSL 2.1.0, 2024. 1, 2, 4

[3] Ezio Bartocci, Alessio Gambi, Felix Gigler, and Dejan Nickovic. Declarative Scenario-based Testing with RoadLogic. *arXiv preprint arXiv:2603.09455*, 2026. 2

[4] Xuan Cai, X Bai, Zhiyong Cui, D Xie, D Fu, H Yu, and Y Ren. Text2Scenario: Text-Driven Scenario Generation for Autonomous Driving Test. *arXiv preprint arXiv:2503.02911*, 2025. 2

[5] CARLA Team. ScenarioRunner for CARLA, 2024. 1

[6] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017. 1, 4

[7] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: A Language for Scenario Specification and Scene Generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, 2019. 2

[8] Srajan Goyal, Alberto Griggio, and Stefano Tonetta. System-level Simulation-based Verification of Autonomous Driving Systems with the VIVAS Framework and CARLA Simulator. *Science of Computer Programming*, 242:103253, 2025. 2

[9] Aaron Kaefer. A Holistic Framework for Scenario-Based Testing of Autonomous Driving Systems. Master's thesis, Technical University of Munich, 2023. 2

[10] Nidhi Kalra and Susan M Paddock. Driving to Safety: How Many Miles of Driving Would it take to Demonstrate Autonomous Vehicle Reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016. 1

[11] Pierre R. Mai. Pmsf py-osc2 framework, 2024. 2

[12] Rupak Majumdar, Aman Mathur, Marcus Pirron, Laura Stegner, and Damien Zufferey. Paracosm: A language and tool for testing autonomous driving systems. *arXiv preprint arXiv:1902.01084*, 2019. 2

[13] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, Raleigh, North Carolina, 2013. Torrossa Resource ID: 5241753. 1

[14] Rodrigo Queiroz, Christian Berger, and Krzysztof Czarnecki. GeoScenario: An open DSL for Autonomous Driving Scenario Representation. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 287–294. IEEE, 2019. 2

[15] Shuncheng Tang, Zhenya Zhang, Jixiang Zhou, Lei Lei, Yuan Zhou, and Yinxing Xue. LeGEND: A Top-Down Approach to Scenario Generation of Autonomous Driving Systems Assisted by Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1497–1508, 2024. 2

[16] Jianan Wang, Bin Li, Jingtao Qi, Xueying Wang, Fu Li, and Hanxun Li. BeSimulator: A Large Language Model Powered Text-based Behavior Simulator. *arXiv preprint arXiv:2409.15865*, 2024. 2

[17] Yongqi Zhao, Wenbo Xiao, Tomislav Mihalj, Jia Hu, and Arno Eichberger. Chat2scenario: Scenario Extraction from Dataset through Utilization of Large Language Model. In *2024 IEEE intelligent vehicles symposium (IV)*, pages 559–566. IEEE, 2024. 2