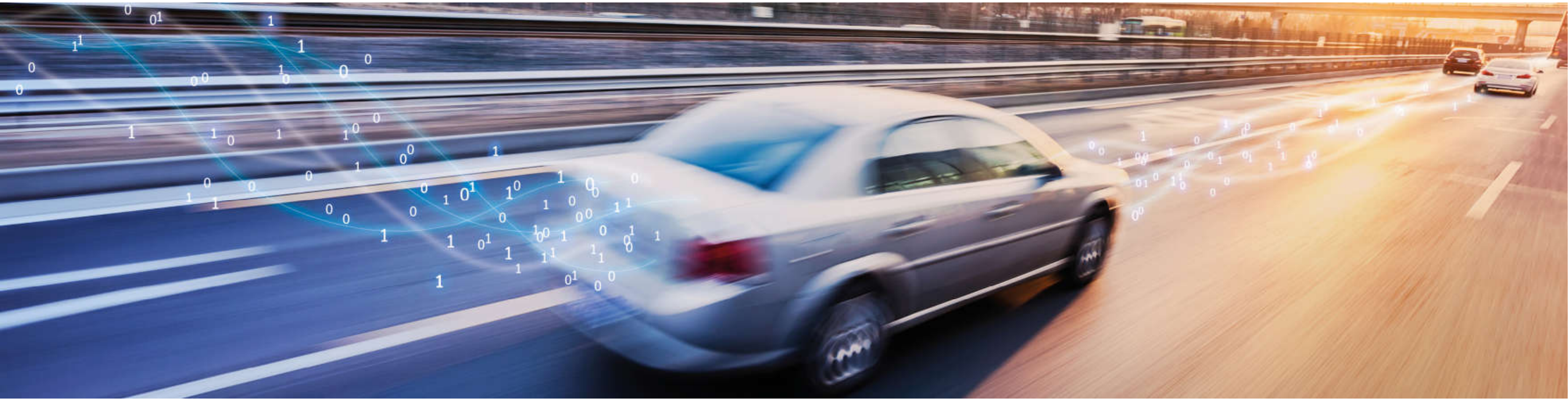
OpenSCENARIO 2.0 Concept Project Status and next steps.

Gil Amid
Foretellix Ltd

18. März 2020
München



Association for Standardization of Automation and Measuring Systems

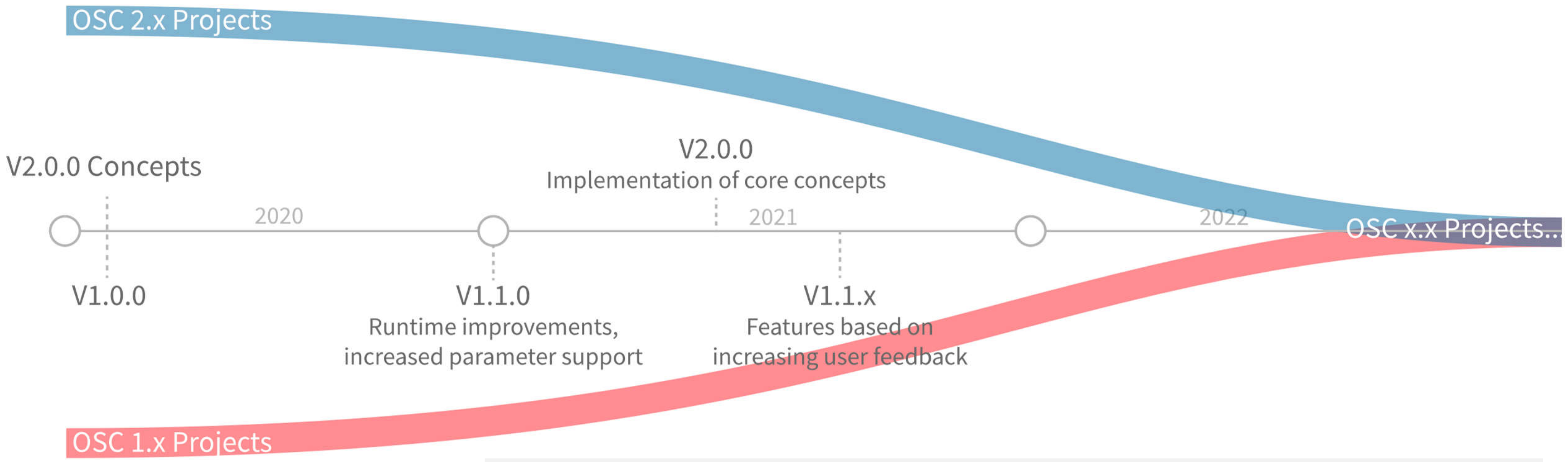
Agenda

Background

Current Status

Technical overview

Roadmap OpenSCENARIO



OSC 2.x Projects
V2.0.0 Concepts
V2.0.0
Implementation of core concepts
2020
2021
2022
OSC x.x Projects...
V1.0.0
V1.1.0
Runtime improvements, increased parameter support
V1.1.x
Features based on increasing user feedback
OSC 1.x Projects

A convergence of the two versions will require further releases of OpenSCENARIO, which will be developed in subsequent OpenSCENARIO 1.x projects.

Roadmap OpenSCENARIO

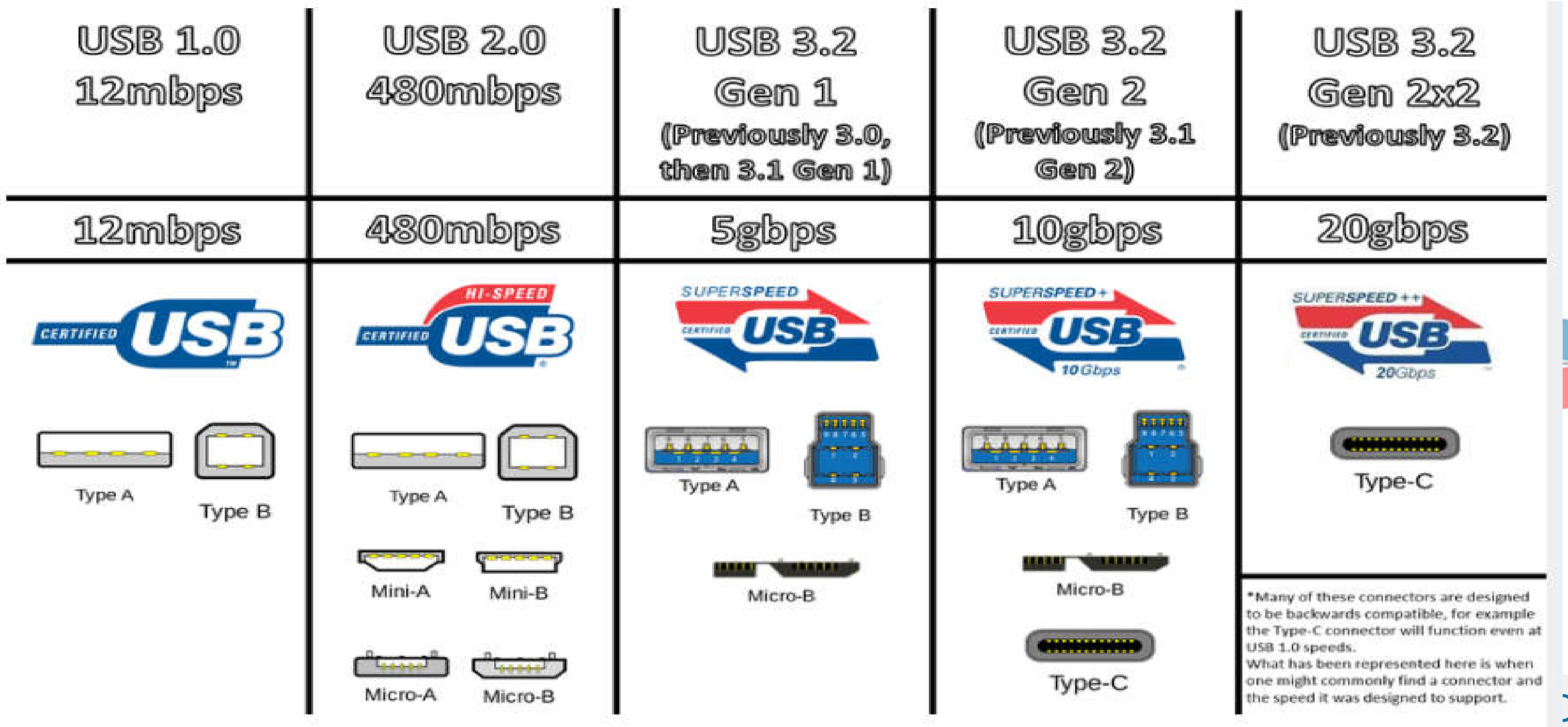
OSC 2.

V2.0.0 Conce

V1.0.0

OSC 1.

OSC x.x Projects...



USB 1.0 12mbps	USB 2.0 480mbps	USB 3.2 Gen 1 (Previously 3.0, then 3.1 Gen 1)	USB 3.2 Gen 2 (Previously 3.1 Gen 2)	USB 3.2 Gen 2x2 (Previously 3.2)
12mbps	480mbps	5gbps	10gbps	20gbps
CERTIFIED USB	HI-SPEED CERTIFIED USB	SUPERSPEED CERTIFIED USB	SUPERSPEED+ CERTIFIED USB 10Gbps	SUPERSPEED++ CERTIFIED USB 20Gbps
Type A, Type B	Type A, Type B, Mini-A, Mini-B, Micro-A, Micro-B	Type A, Type B, Micro-B	Type A, Type B, Micro-B, Type-C	Type-C

*Many of these connectors are designed to be backwards compatible, for example the Type-C connector will function even at USB 1.0 speeds.
What has been represented here is when one might commonly find a connector and the speed it was designed to support.

OpenSCENARIO,
which will be developed in subsequent OpenSCENARIO 1.x projects.

ASAM

Motivation

- AV development and certification requires massive usage of scenario driven simulation. Exhaustive simulation is a MUST HAVE for development and qualification of AD and Autonomous driving systems

- OpenSCENARIO 1.x just completed its transfer to ASAM, and is in its stabilization phase.
  - during various workshop it became clear there are additional needs, which may not be met by evolution of the current format.

- OSC Overall Goal: A standard with all the required features to enable testing and validation of ADAS systems and autonomous vehicles.

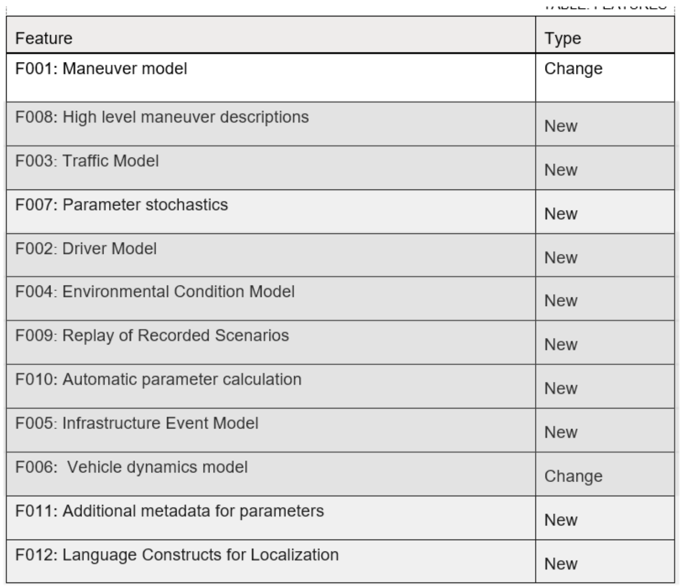
- OpenSCENARIO 2.0 should serve as the format and mechanism to supply dynamic content and functional behavior to all testing and execution platforms, for all driving scenarios ranging from simple motor-way interactions to long-running, complex inner-city traffic scenarios.

Technical Content

- OpenSCENARIO 2.0 needs to support:
  - Definition of tests and scenarios for the full development process of autonomous vehicles
  - the full complexity of real-world scenarios, including complex inner-city traffic.

- Required use cases: span from pure software-based simulation, through SIL, HIL, VIL hybrid testing models, up to test tracks and street driving.

- Concept project focus:
  - Focus on the set of 12 features as defined in the proposal work shop (early 2019)
  - Define architecture for the main scenario models, and interface to other required models (e.g. Environment, Driver, Traffic)
  - Address varying levels of requirements for parametrization, accuracy
  - Address different use cases of scenarios.



Feature	Type
F001: Maneuver model	Change
F008: High level maneuver descriptions	New
F003: Traffic Model	New
F007: Parameter stochastics	New
F002: Driver Model	New
F004: Environmental Condition Model	New
F009: Replay of Recorded Scenarios	New
F010: Automatic parameter calculation	New
F005: Infrastructure Event Model	New
F006: Vehicle dynamics model	Change
F011: Additional metadata for parameters	New
F012: Language Constructs for Localization	New

General Requirements

- The requirements span over many use cases, and many needs.

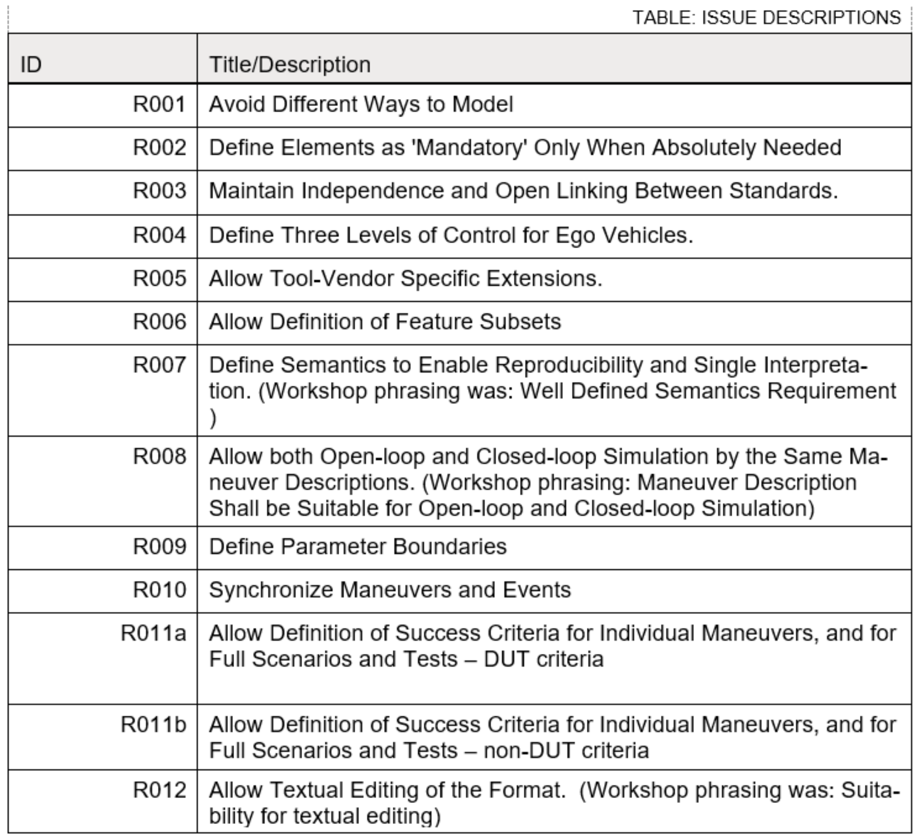


TABLE: ISSUE DESCRIPTIONS

ID	Title/Description
R001	Avoid Different Ways to Model
R002	Define Elements as 'Mandatory' Only When Absolutely Needed
R003	Maintain Independence and Open Linking Between Standards.
R004	Define Three Levels of Control for Ego Vehicles.
R005	Allow Tool-Vendor Specific Extensions.
R006	Allow Definition of Feature Subsets
R007	Define Semantics to Enable Reproducibility and Single Interpreta-tion. (Workshop phrasing was: Well Defined Semantics Requirement)
R008	Allow both Open-loop and Closed-loop Simulation by the Same Ma-neuver Descriptions. (Workshop phrasing: Maneuver Description Shall be Suitable for Open-loop and Closed-loop Simulation)
R009	Define Parameter Boundaries
R010	Synchronize Maneuvers and Events
R011a	Allow Definition of Success Criteria for Individual Maneuvers, and for Full Scenarios and Tests – DUT criteria
R011b	Allow Definition of Success Criteria for Individual Maneuvers, and for Full Scenarios and Tests – non-DUT criteria
R012	Allow Textual Editing of the Format. (Workshop phrasing was: Suita-bility for textual editing)

Current Status

Key Messages -Concept Project Completed

- Project included ~100 engineers from ~50 companies. (about 50% active – attending f2fs)
- Concept Document completed in February, approved by ASAM TSC
- Concept Document is released on ASAM web site.
  Link: https://releases.asam.net/openscenario-2-0-concept/ASAM_OpenSCENARIO_2-0_Concepts.html
- Based on the considerations of the concept group, OpenSCENARIO 2.0 is proposed to be founded on the concept of a domain-specific language, that should support all levels of scenario description, from the very abstract to the very concrete in a suitable way.

Key messages – next steps

- The ASAM OpenSCENARIO 2.0 project is aimed at taking the concepts specified in the OpenSCENARIO concept document and continue and develop the next generation of the OpenSCENARIO standard: OpenSCENARIO 2.0.

- A rough estimate is that the development of such a standard can be achieved within a year, aiming at release in Q2 of 2021.

- OSC 1.x provides a very concrete scenario description format, usable now, that will be directly compatible with the 2.0 project. OpenSCENARIO 2.0 provides a Domain Specific Language (DSL) and aims to significantly extend the domain addressed by 1.x to cover further use cases for AD development. The two groups will also jointly develop a migration mechanism that grants unchanged run-time behavior for OpenSCENARIO 1.x scenarios converted to OpenSCENARIO 2.0.

Next steps

- A proposal workshop (virtual) is set for Mar-26
- Details and registration: https://www.asam.net/conferences-events/detail/asam-openscenario-v20/

Technical overview

Use Cases and User Stories

- The concept project constructed and analyzed more than 40 use cases/user stories , covering various usage mode by test engineers, content model and software developers, system and safety engineers, regulators and type approval authorities (e.g. NHTSA test scenarios)

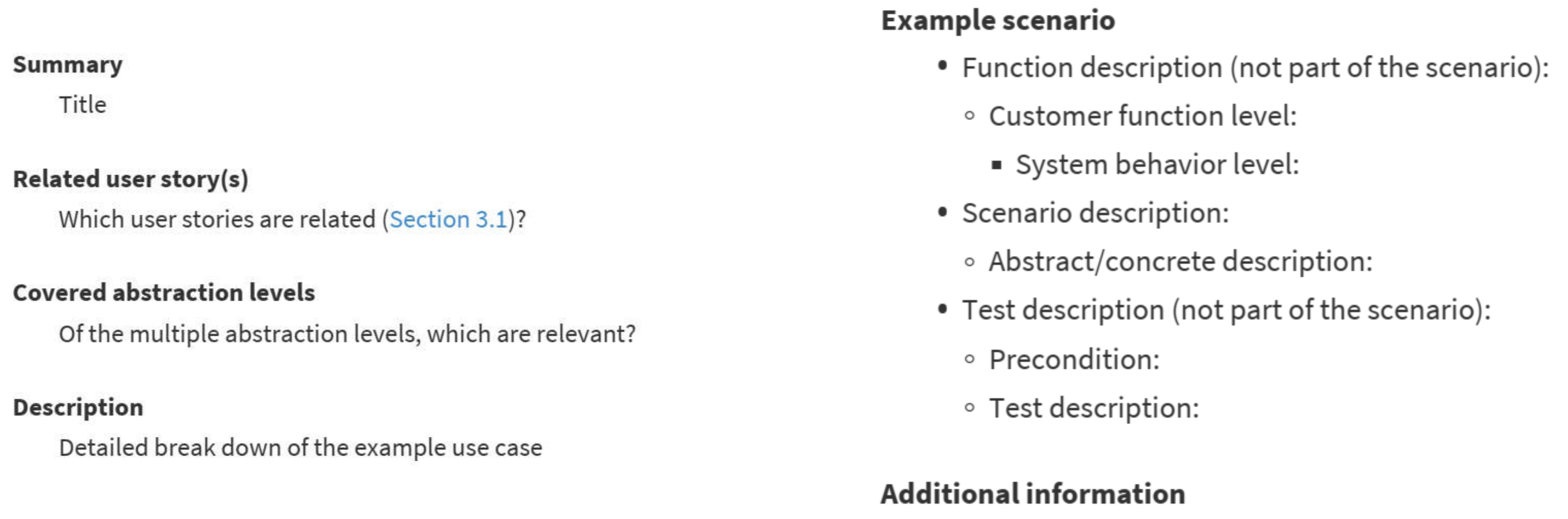
- These serve two goals:
  - the scope of the project is clear - both to the project participants and to anyone not directly involved in the initial development
  - the requirements for the DSL can be easily analysed and extended based on further use cases in the future.

User Stories - Examples

- User stories are phrased from an end-user perspective, who will be interacting with the tools and processes required to define, implement and consume results from test scenarios
- Few Examples:
  - As a test engineer, I can build and run tests as similarly as possible on different execution platforms.
  - As an auditor/regulator, I can compare the outcome of different execution platforms when they have the same OpenSCENARIO input.
  - As an existing tool provider or consumer, I can migrate information from previous versions of OpenSCENARIO into OpenSCENARIO 2.0
  - As a development project lead, I can create scenarios on an abstract level to document the functional behavior for legal reasons.
  - As a SOTIF safety engineer and/or V&V engineer, AV developer, scenario creator, I can use OpenSCENARIO 2.0 to discover scenarios that are going to uncover safety hazards. This refer to SOTIF and safety hazards that can be present even if the system is functioning as designed, without a malfunction.

Use Cases

- The intent of OpenScenario 2.0 is to cover use cases at varying levels of autonomy. Hence, they should represent an adequate level of complexity including maneuvers and ODD features that are not accounted for otherwise
- Use Cases were developed using a template including



Summary

Title

Related user story(s)

Which user stories are related (Section 3.1)?

Covered abstraction levels

Of the multiple abstraction levels, which are relevant?

Description

Detailed break down of the example use case

Example scenario

- Function description (not part of the scenario):
  - Customer function level:
    - System behavior level:
- Scenario description:
  - Abstract/concrete description:
- Test description (not part of the scenario):
  - Precondition:
  - Test description:

Additional information

USE CASES

- Very Simple Example: Scenario definition for entering a roundabout , Concrete, left turn

Summary

A left turn at a roundabout should be described as a scenario

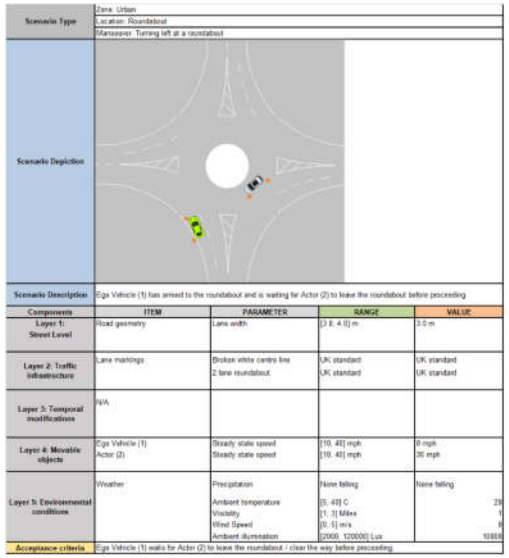
Applicable user story(s)

D1

Covered abstraction levels

Concrete

Description



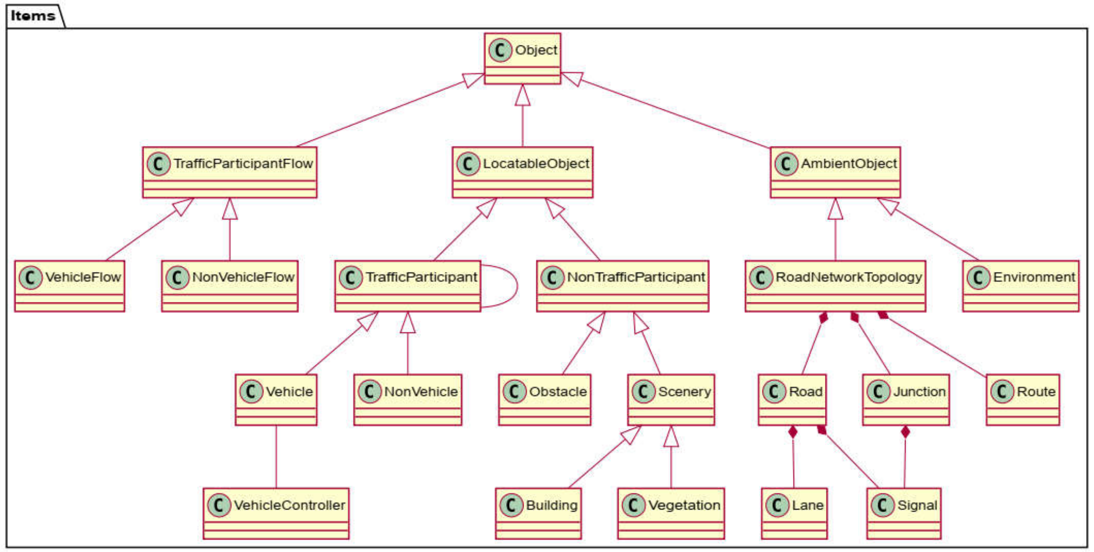
Scenario Type	Zone: Urban			
	Location: Roundabout			
	Manoeuver: Turning left at a roundabout			
Scenario Depiction				
Scenario Description	Ego Vehicle (1) has arrived to the roundabout and is waiting for Actor (2) to leave the roundabout before proceeding			
Components	ITEM	PARAMETER	RANGE	VALUE
Layer 1: Street Level	Road geometry	Lane width	[3.0, 4.0] m	3.0 m
Layer 2: Traffic infrastructure	Lane markings	Broken white centre line 2 lane roundabout	UK standard UK standard	UK standard UK standard
Layer 3: Temporal modifications	N/A			
Layer 4: Movable objects	Ego Vehicle (1) Actor (2)	Steady state speed Steady state speed	[10, 40] mph [10, 40] mph	0 mph 30 mph
Layer 5: Environmental conditions	Weather	Precipitation Ambient temperature Visibility Wind Speed Ambient illumination	None falling [5, 40] C [1, 3] Miles [0, 5] m/s [2000, 120000] Lux	None falling 28 1 8 10000
Acceptance criteria	Ego Vehicle (1) waits for Actor (2) to leave the roundabout / clear the way before proceeding			

```
scenario Scenario014:
    set_map("MFM_map.xodr")
    ego: car
        keep(ego.color == green)
        keep(ego.category == sedan)
        car1: car
        keep(car1.category == sedan)
        p: path
            path_type(p, roundabout)
            path_min_lanes(p, 2)
            path_min_arms(p, 4)
        set_weather: weather(kind: nice, temperature: 20c)
        do parallel:
            get_ahead_ego: ego.drive(p) with:
                speed(0mph, at: start)
                lane(1, in: segment1, at: start)
                lane(1, in: segment2, at: end)
            get_ahead_car1: car1.drive(p) with:
                speed(30mph, at: start)
                lane(-2, in: segment5, at: start)
                lane(1, in: segment2, at: end)
```

DOMAIN MODEL & Entities

- A domain model foundation was developed, defining the key entities needed, and their relations.
- Further development is expected in the standardization project.
- In order to ensure sync with OpenSCENARIO 1.0, a UML diagram visualizing actions defined in OpenSCENARIO 1.0 is included in the document.

- High level Domain Model:



OSC 2.0 - DOMAIN SPECIFIC LANGUAGE

- The foundational concept of OpenSCENARIO 2.0 is to establish a domain specific language of a declarative nature.

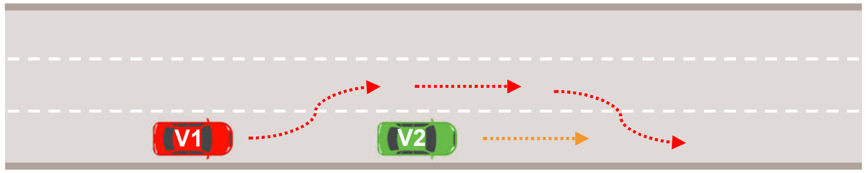
- A declarative language describes what should happen on scenario execution (including the required parameterization/variation), rather than how to do it.

- A declarative language can also have a *dual interpretation*, i.e. provide a single scenario description which can be used to describe both how to make it and how to monitor that it *indeed* happened. (This is important if we want to condition some operation on the fact that some other scenario has happened before, without having to describe in detail *how* to cause that scenario.)

- Foretellix's M-SDL language is used to supply examples in the concept document, and in next slides. Reference manual available on ASAM site: https://releases.asam.net/openscenario-2-0-concept/M-SDL_LRM_OS.pdf

DSL - dual interpretation

- Each scenario definition has two interpretations:
  - Active: Make this scenario happen
  - Passive: Monitor whether it happened



- Why: Because scenarios sometimes don't happen as planned
  - We don't want to take credit if it did not really happen
  - We want to collect coverage according to *actual* values
- Why: Because we want to monitor scenarios we did *not* plan
  - E.g. from recorded drone videos or from the AV's sensors
  - E.g. from simulations not controlled by M-SDL
  - We want to collect coverage on which scenarios *happened*, and with which *parameters*

```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    p: path

    do parallel(duration: [3..20]s):
        v2.drive(p)
        serial:
            A: v1.drive(p) with:
                lane(same_as: v2, at: start)
                lane(left_of: v2, at: end)
                position([10..20]m, behind: v2, at: start)
            B: v1.drive(p)
            C: v1.drive(p) with:
                lane(same_as: v2, at: end)
                position([5..10]m, ahead_of: v2, at: end)
```

DSL concepts

- A *scenario* describes behavior over time
  - Scenarios are defined via a *scenario definition*, and invoked (activated, called) via a *scenario invocation*
  - *Modifiers* can be used to modify (influence) scenario invocations

- There are three kinds of scenarios / maneuvers etc.
  - Scenario operators (e.g. serial) are used to compose other scenarios
  - Atomic maneuvers (e.g. drive) define the basic capabilities of actors
  - User-defined scenarios (e.g. *overtake*)

- Scenario definitions have *parameters* with types
  - Parameters types can be *time* or *speed* or even *car* (i.e. a reference to an actor)
  - Parameter values are set when scenarios are invoked
  - The final value will be set according to the type and the constraints applied

DSL concepts (continued)

- Actors are the “players” in a scenario (e.g. vehicles, people etc.)
  - A scenario can define the behavior of a single actor or orchestrate the behavior of multiple actors
  - An actor can also represent a group of other actors, like *traffic* and *car_group*

- Constraints / modifiers modify (influence) behavior
  - *keep()* defines a constraint which influences one or more parameters
  - modifiers influence scenario behavior even more generally (e.g. determine speeds, paths etc.)

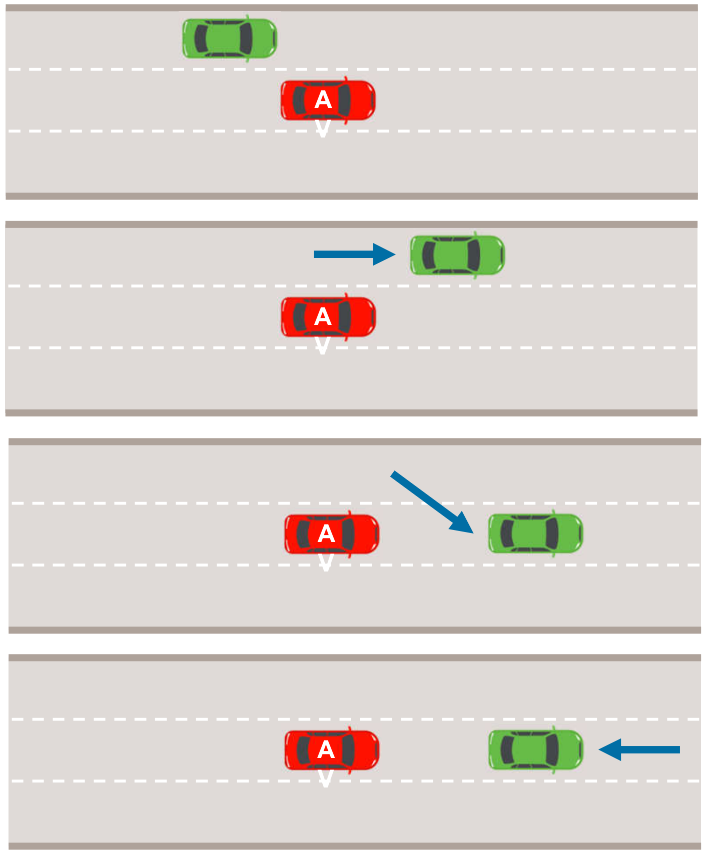
- Coverage definitions define values to be collected for coverage analysis
  - *cover()* defines which values to collect, and which

Constraints, randomization and coverage

- Constraints are used to
  - Narrow down parameter values
  - Connect values of multiple parameters
- When generating a test, for any parameter not assigned a value
  - The system will pick a random value while obeying the constraints
  - By default it will use a flat random distribution, but you can request any other distribution function
- Use modifiers to influence the scenario’s dynamics
  - Like speed, position, where on the map it should happen etc.
- You can go from very concrete to very abstract
- Use coverage to collect values for multi-run analysis

Example: cut_in_and_slow

```
scenario dut.cut_in_and_slow:

    car1: car                          # The other car
    side: av_left_right              # A side: left or right
    path:  path                       # A path in the map
   path_min_lanes(path, 2)        # Path should have at least two lanes

   do serial:
       get_ahead: parallel(duration: in [1..5]s):
            dut.car.drive(path)  with:
                speed([30..70]kph)
            car1.drive(path, adjust: TRUE) with:
                position([5..100]m, behind: dut.car,at: start)
                position([5..15]m, ahead_of: dut.car, at: end)

     change_lane: parallel(duration: in [2..5]s):
             dut.car.drive(path)
             car1.drive(path) with:
                 lane(side_of: dut.car, side: side, at: start)
                 lane(same_as: dut.car, at: end)

     slow: parallel(duration: in [1..5]s):
            dut.car.drive(path)
            car1.drive(path) with:
                speed_change(-[10..15]kph)
```



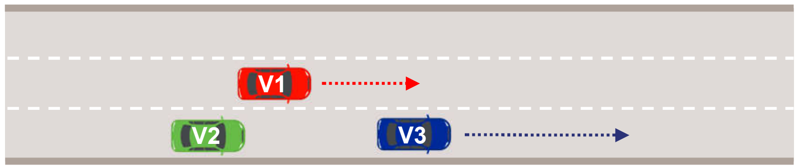
Using modifiers to control scenario dynamics

- Modifiers are like constraints but more general. Examples:
- Control movements via *movement* modifiers
  - v1 drives 10..20 kph faster than v2:

    v1.drive() with: speed([10..20]kph, faster_than: v2)
- Control the scenario's location via *path* modifiers
  - Path (road) p should have at least 2 lanes:

    path_min_lanes(p, 2)
- Control *synchronization* between events
  - Sync these two events to within -1..1 second of each other:

    synchronize(phase_a.end, phase_b.start, [-1..1]s)
- You can use *any number* of modifiers in the same invocation
  - E.g. to express the complex situation on the right



```
v3.drive(p) with:
    lane(right_of: v1)
    speed([7..15]kph, faster_than: v1)
    position([20..70]m, ahead_of: v1)
    position([10..30]m, ahead_of: v2)
    lane(same_as: v2)
    lateral([10..25]cm, left_of: v2)
```

Scenario invocation syntax

- Scenario name
  - scenario operators

    serial: … parallel: … first_of: … one_of: … mix: … repeat: …
  - atomic scenarios (actions)

    drive() … walk() … wait …
  - user-defined scenarios

    overtake() … cut_in() …
- Scenario invocation

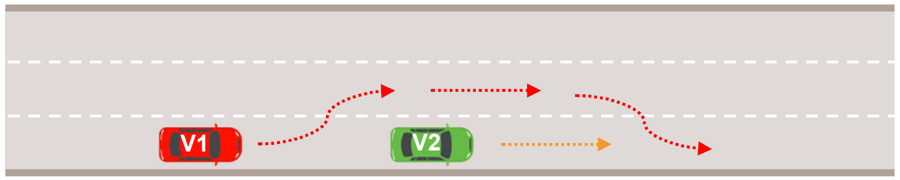
  [label:] [path.]name(parameter, …) [with: *modifier …]*
  - label is optional

    d: drive(…) … or drive(…) …
  - *path* is optional

    dut.car.drive(…) … or drive(…) …
  - *parameter* can be by name or by position

    drive(path) or drive(path)
  - *modifier* is similar to scenario invocation

    speed(5 kmh, faster_than: car1)



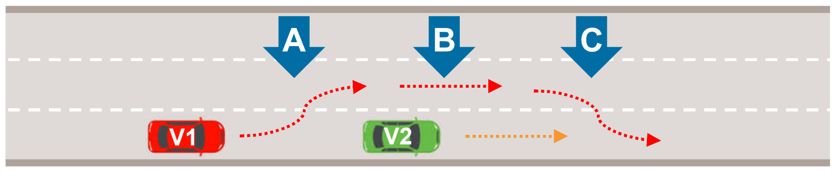
```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    p: path
    keep(v1.color != green)

    do parallel(duration: [3..20]s):
        v2.drive(p)
        serial:
            A: v1.drive(p) with:
                lane(same_as: v2, at: start)
                lane(left_of: v2, at: end)
                position([10..20]m, behind: v2, at: start)
            B: v1.drive(p)
            C: v1.drive(p) with:
                lane(same_as: v2, at: end)
                position([5..10]m, ahead_of: v2, at: end)
```

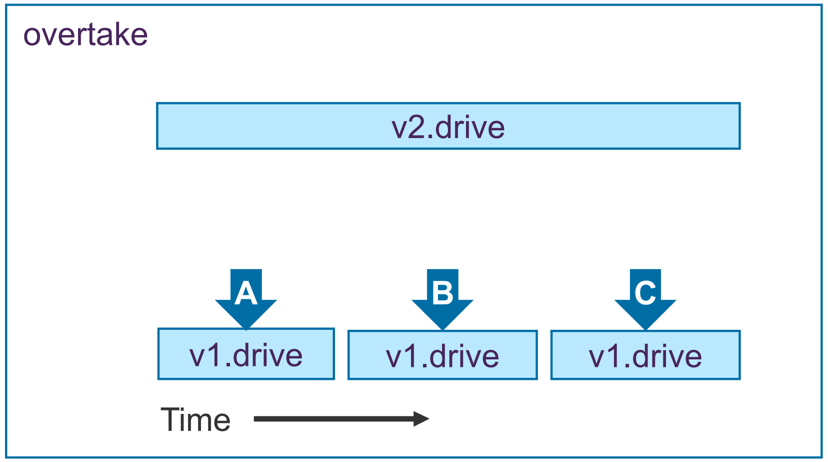
```
import sumo_config.sdl # Execution platform
import lane_change_scenarios.sdl # Library

extend top.main:  # Extend the predefined main
    set_map("some_map.xodr") # Map to use in test
    do overtake(v2: dut.car)
```

Composition: Writing a full scenario

- Here is the full *overtake* scenario

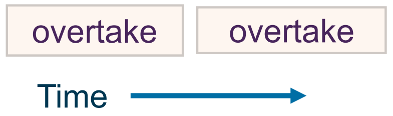
```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    p: path

    do parallel(duration: [3..20]s):
        v2.drive(p)
        serial:
            A: v1.drive(p) with:
                position([10..20]m, behind: v2, at: start)
                lane(same_as: v2, at: start)
                lane(left_of: v2, at: end)
            B: v1.drive(p) with:
                position([1..10]m, ahead_of: v2, at: end)
            C: v1.drive(p) with:
                lane(same_as: v2, at: end)
                position([5..10]m, ahead_of: v2, at: end)
```





- You can then compose this scenario using e.g. *serial*

```
scenario overtake_serial
    car_a: car
    car_b: car
    do serial:
        overtake(v1: car_a, v2: dut.car)
        overtake(v1: car_b, v2: dut.car)
```



Example: Writing a concrete scenario

- So far, we wrote an abstract scenario, then constrained it “from above”

```
scenario traffic.overtake:
    v1: car
    …
    do parallel(duration: [3..20]s):
        …   position([10..20]m, behind: v2, at: start)
```

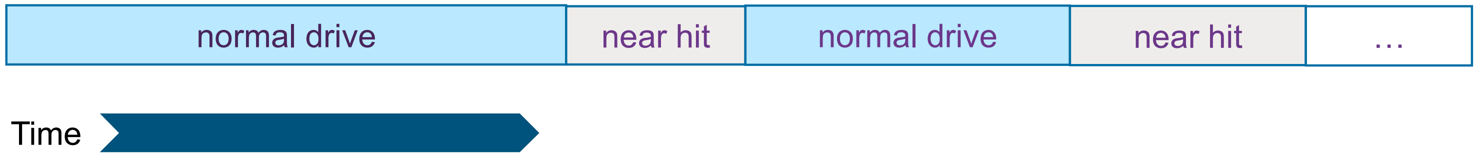
Some lines from the original abstract scenario: Note the ranges

- We can write a concrete scenario “from scratch”

```
scenario traffic.concrete_overtake:
    v1: car:
    keep(v1.color == green)
    keep(v1.category == truck)
    …
    do parallel(duration: 7second):
       … position(10.5m, behind: v2, at: start)
       … speed(18.7kph)  # Note that speed was not
mentioned
```

Same lines if we want to write a concrete scenario from the start

Example: Driver-in-the-loop



```
scenario dut.near_hit:
    do one_of():
        turn_right_plus(v2: dut)
        overtake(v2: dut)
        car_ignoring_red_light()
        …
```

This scenario will cause a random near hit situation

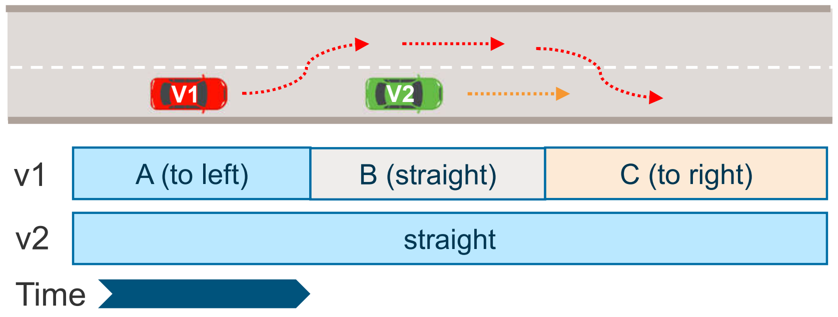
```
scenario dut.DIL_multi_near_hit:
    how_long: time # How long to run it

    do run_time(duration: how_long):
        dut.car.drive(duration: how_long) # Drive the dut
        repeat():
            wait_time([5..20]s) # Let him relax a bit
            near_hit() # Plan the next near-hit
```

This scenario will repeatedly wait some seconds and then plan and execute another random near-hit

Concrete to abstract

```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    p: path

    do parallel(duration: [3..20]s):
        v2.drive(p)
        serial:
            A: v1.drive(p) with:
                lane(same_as: v2, at: start)
                lane(left_of: v2, at: end)
                position([10..20]m, behind: v2, at: start)
            B: v1.drive(p)
            C: v1.drive(p) with:
                lane(same_as: v2, at: end)
                position([5..10]m, ahead_of: v2, at: end)
```

This is a more abstract version of overtake



This is a very concrete version of overtake

```
scenario traffic.overtake_concrete:
    v1: car with(category: sedan, color: black)
    p: path
    path_explicit(p,[point("15",30m), point("95",1.5m), …])

    do parallel(duration: 10s):
        dut.car.drive(p) with:
            lane(2)
            speed(50kph)
        serial:
            A: v1.drive(p, duration: 3s) with:
                speed(70kph)
                lane(2, at: start)
                lane(1, at: end)
                position(15m, behind: dut.car, at: start)
                position(1m, ahead_of: dut.car, at: end)
            B: v1.drive(p, duration: 4s) with:
                position(5m, ahead_of: dut.car, at: end)
            C: v1.drive(p, duration: 3s) with:
                speed(80kph)
                lane(2, at: end)
                position(10m, ahead_of: dut.car, at: end)
```

```
scenario traffic.overtake_dut
    do overtake(v2: dut.car) with:
        keep(it.A.duration == 3s)
```

This scenario invokes overtake with some parameters

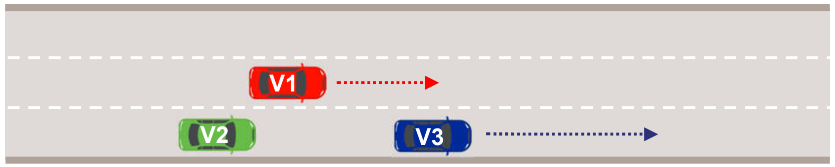
```
scenario traffic.overtake_serial
    car_a: car
    car_b: car
    do serial:
       overtake(v1: car_a, v2: dut.car)
       overtake(v1: car_b, v2: dut.car)
```

This scenario does two overtakes serially

```
scenario traffic.overtake_repeat
    do repeat(count: 10):
       wait_time([10..20]s)
       overtake_serial
```

This scenario repeats overake_serial 10 times

Multiple, independent movement constraints



This is phase A
Note the relations (speed, position, lateral offset etc.) between v3 and the other cars

Phase A	Phase B

```
scenario traffic.multi_car:
    v1: car # The first car
    v2: car # The second car
    v3: car # The third car
    p: path

    do serial:
        A: parallel(duration: [3..20]s):
            v1.drive(p) with: …
            v2.drive(p) with: …
            v3.drive(p) with:
                lane(right_of: v1)
                speed([7..15]kph, faster_than: v1)
                position([20..70]m, ahead_of: v1)
                position([10..30]m, ahead_of: v2)
                lane(same_as: v2)
                lateral([10..25]cm, left_of: v2, measured_by: center_to_center)
        B: parallel(duration: [3..20]s):
            v1.drive(p) with: …
            v2.drive(p) with: …
            v3.drive(p) with: …
```

Here is how you say that. Note that we need here six movement constraints (modifiers), each with its own set of parameters.

Using event-based synchronization

```
scenario traffic.multi_car:
    v1: car # The first car
    v2: car # The second car
    p: path

    event e1 is map.reach_position(v2, point1)
    event e2 is map.reach_speed(v2, 40kph)
    …

    do parallel:
        v1_part: serial:
            wait @e1
            v1.drive(p) with:
                speed(…)
                position(…)
                on @e2: end()
            v1.drive(p) with:
                speed(…)
                position(…)
                on @e3: end()
         v2_part: serial:
            v2.drive(p) with:
                speed(…)
                position(…)
                on @e4: end()
            v2.drive(p) with:
                speed(…)
                position(…)
                on (v1.distance_to(point2) < 7): end()
            v2.drive(p) with:
                speed(…)
                position(…)
            …
```

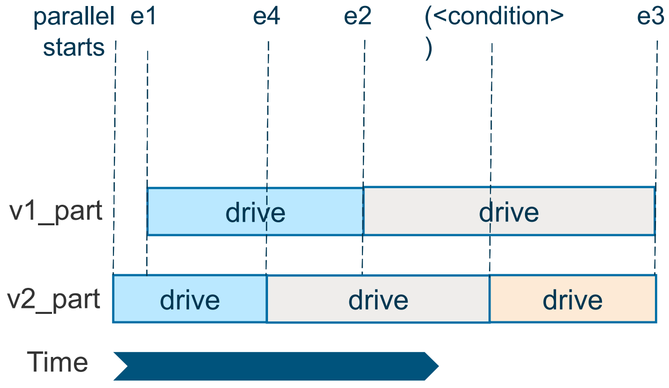
Events represent moments in time. Can be defined using any condition(or other events)

Define the whole scenario as "parallel of serials"

End each step of the serial upon some event

Can also specify the condition inline

parallel of serials



Note that this whole parallel-of-serials can still be a lego brick in something bigger

```
scenario traffic.multi_car_plus
    do serial:
       multi_car(v2: dut.car)
       if (dut.car.distance_to(point3) < 10):
           repeat(count: 3):
              overtake_serial
```

Using coverage

- Coverage is used to analyze “what we did so far”
  - Which part of the “scenario space” have we exercised our AV in?

- Use cover() to specify what items to sample, when to sample them, how to split them into buckets etc.
  - E.g. *cover(side)* means “sample the ‘side’ parameter at the end of the *overake* scenario

```
extend traffic.overtake:


    cover(side)
    cover(v1.color, name: other_car_color)
    cover(min_ttc(v1, v2), name: min_time_to_collision,
          unit: millisecond, range: [0..3000], every: 100)
```

These coverage items will be sampled whenever this scenario happens (for offline, aggregate analysis)

- Note: Coverage can be specified and collected over many scenario objects and properties
  - In-line or concurrently, possibly using procedural modeling code.

Thank you for your attention!

Gil Amid
Foretellix Ltd

Phone: +972-58-4347475
Email:gil.amid@Foretellix.com