foretellix

verifying autonomous systems

M-SDL Overview

Yoav Hollander - Founder and CTO, Foretellix

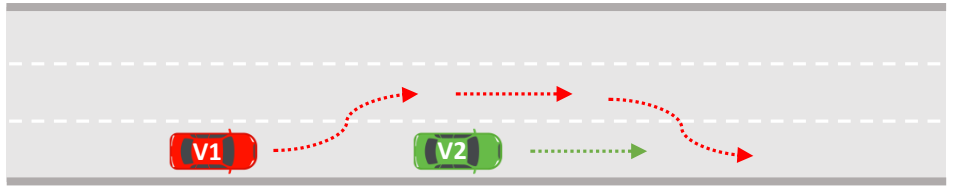
blog.foretellix.com

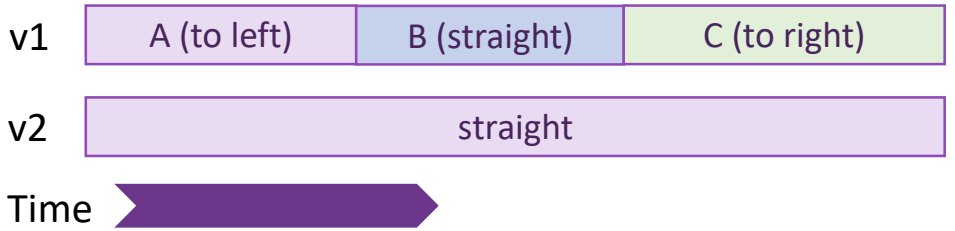
M-SDL (Measurable Scenario Description Language) – Key Characteristics

- Open, Non-Proprietary
- A good combination of the following trade-offs
  - Power: Ability to write currently-unimagined scenarios
  - Readability: For both simple *and* complex scenarios
  - Composability: Critical enabler to maintain readability
- Portable across different execution platforms (Simulation, X in a loop, test tracks and street driving) from a concrete scenario on the test track to a fully random scenario on virtual simulators
- Extensible
- Can be visualized and textualized
- Dual – make the scenario happen but also monitor for its occurrence

Why use a declarative DSL

- Easier to read / write
  - Syntax adapted to the task
  - “Write only what’s needed”
- Easier to reuse
  - Because it can be fully analyzed
- Easier to understand “what’s the full language”
  - Important for defining a standard
- But to be powerful enough, it needs to be extensible
  - Add agents (actors) and scenarios (maneuvers, actions)
  - Extend existing agents and scenarios
  - Call methods written in any language
  - Connect to models written in any language



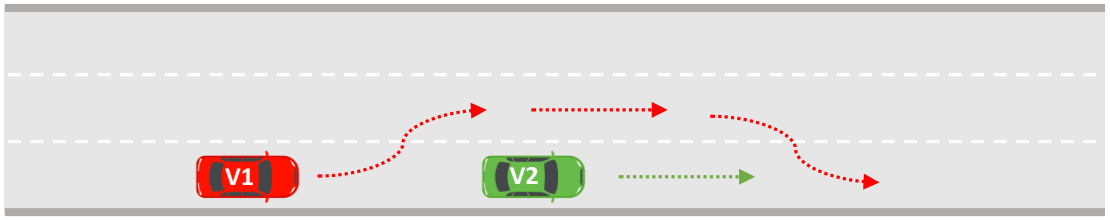



```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    p: path
    keep(v1.color != green)

    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)
```


foretellix

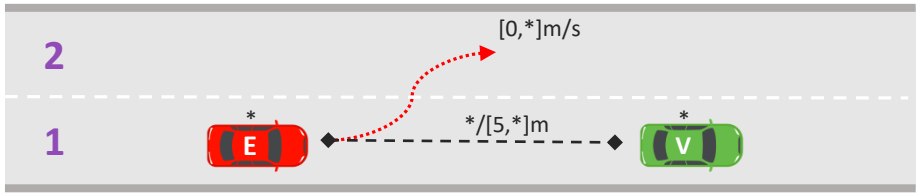
A declarative DSL can be visualized

- Can also auto-generate text explanations, UML diagrams, C++ header files etc.



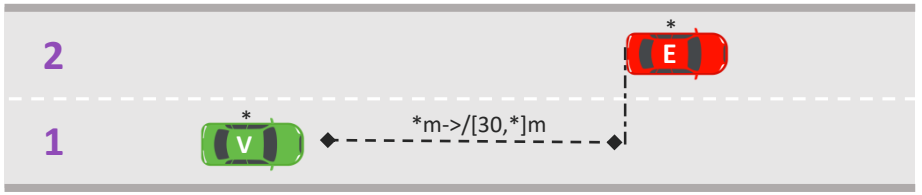

```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    p: path

    do parallel:
        v2.drive(p)
        serial:
            A: v1.drive(p) with:
                position([5..]m, behind: v2, at: start)
                lane(same_as: v2, at: start)
                lane(left_of: v2, at: end)
            B: v1.drive(p)
            C: v1.drive(p) with:
                position([30..]m, ahead_of: v2, at: start)
                lane(same_as: v2, at: end)
```

Phase A: Change to left lane



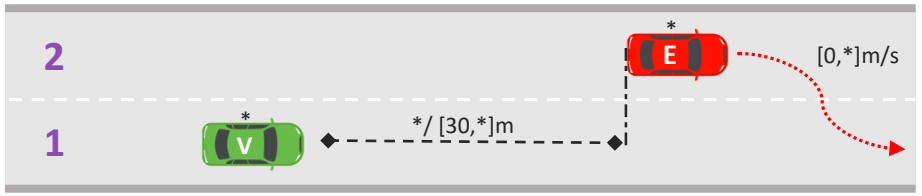
Duration: [0.0, Infinity]s

Phase B: Passing vehicle



Duration: [0.0, Infinity]s

Phase C: Change back to right lane

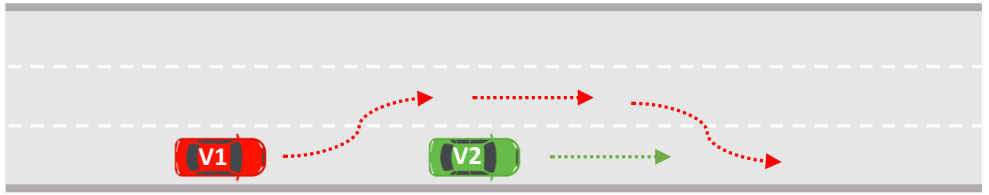


Duration: [0.0, Infinity]s

foretellix

A declarative DSL can be active and passive

- 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 recorded scenarios
  - E.g. from drone camera or from the AV's sensors
  - And 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
    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)
```


foretellix

The sentence-style variant

- We can also support the sentence-style variant, which reads more like English
  - No problem supporting sentence-style and function- style together

function-style variant

```
lane(same_as: v2, at: start)
lane(left_of: v2, at: end)
position([10..20]m, behind: v2, at: start)
```

sentence-style variant

```
lane same_as (v2) at (start)
lane left_of (v2) at (end)
position ([10..20]m) behind (v2) at (start)
```

M-SDL syntax: A quick tour though the M-SDL "schema"

- Top level statements
  - import
    import my_file.sdl
  - type definition / type extension
    agent car_group: … struct junction: … scenario dut.cut_in: …
    extend car_group: …
- Instruct / agent / scenario definition
  - field
    s: speed
  - constraint
    keep(s > 30 kph)
  - event
    event too_close is (distance_between(car1, car2) < 10m)
  - method definition
    def distance_between (car1: car, car2: car) is external "python cars.dist"
- In scenario definition, also
  - scenario modifiers
    set_map("my_map.xodr")
  - behavior definition
    do <scenario-invocation> …

```
agent car: # Already in basic library
    color: car_color
    category: car_category
```

```
extend car:
    weight: real
    keep(weight in [500..4000]kg)
```

```
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):
        …
```

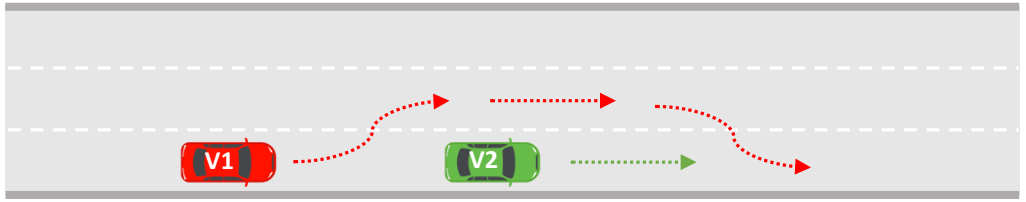
```
import sumo_config.sdl # Execution platform
import lane_change_scenarios.sdl # Library

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

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)
```


foretellix

Scenario modifiers (for flexibility and composability)

- Movement modifiers

```
v1.drive(p) with:
    position(7m, behind: v2)
    position(10m, ahead_of: v3)
    speed([10..20]kph, faster_than: v4)
    lane(left_of: v5, at: start)
```

Can add any number of modifiers

- Topology modifiers

```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    p: path
    path_min_lanes(p, 3) # Path (road) must have at least three lanes
    path_has_sign(p, speed_limit) # Path must have a speed_limit sign
    …
```

Can specify topology constraints – tool can find any matching location in the map

```
extend traffic.overtake
    set_map("some_map.xodr") # Specify which map to use
    path_explicit(p,  # Specify an explicit, point-by-point path
                  [point("15",30m), point("95",1.5m), …])
```

Can specify explicit path on the map (using segment IDs and distances)

foretellix

Scenario modifiers (continued)

- Synchronization modifiers (see next slide)

```
extend top.main:
    do mix():
        s1: …
        s2: …
    synchronize(s1.A, s2.t1.enter, [-1..1]s)
```

These two points in the scenarios should happen within [-1..1]s of each other

- Coverage modifiers

```
scenario traffic.overtake:
    v1: car # The first car
    v2: car # The second car
    side: [left, right] # From which side to overtake
    …
```

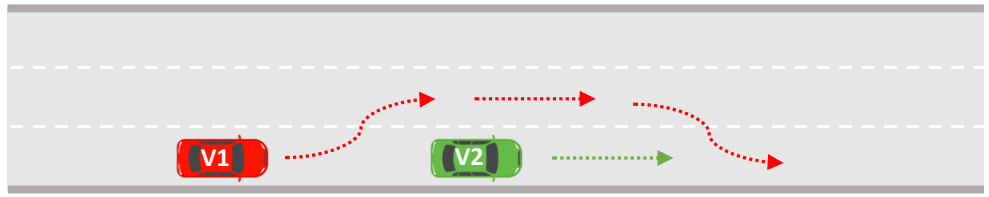
```
extend traffic.overtake:
    cover(side)
    cover(v1.color, name: other_car_color)
    cover(min_ttc(v1, v2), name: min_time_to_collision,
          unit: millisecond, range: [0..3000], every: 100)
```

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

Example: mixing and synchronization

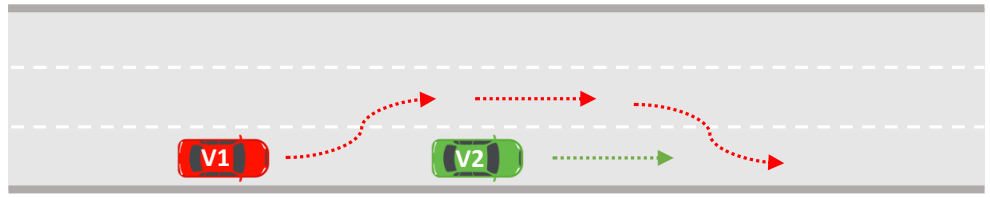
```
extend top.main:
    do overtake(v2: dut.car)
```

Run overtake
(with v2 as the dut car)



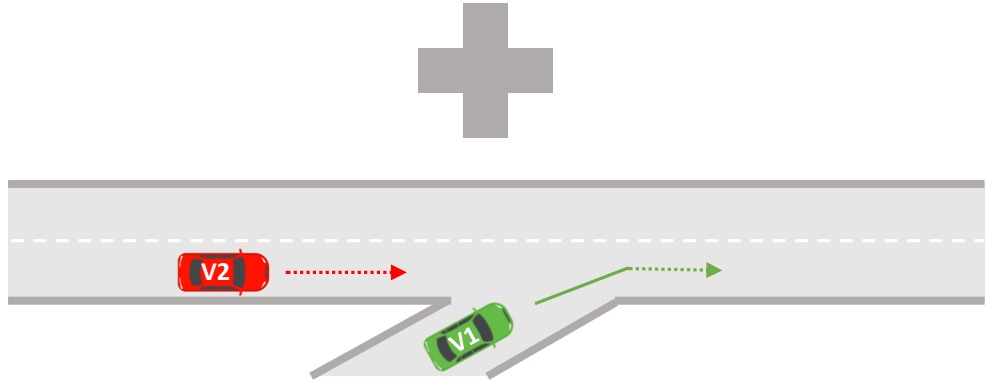

```
extend top.main:
    do mix():
       s1: overtake(v2: dut.car)
       s2: turn_right(v2: dut.car)
    synchronize(s1.A, s2.t1.enter, [-1..1]s)
```

mix overtake
(with v2 as the dut car)
and turn_right
(with v2 the dut car)



Synchronize the end of phase A of overtake) and the end of the enter phase (of turn_right) to within [-1..1]s of each other

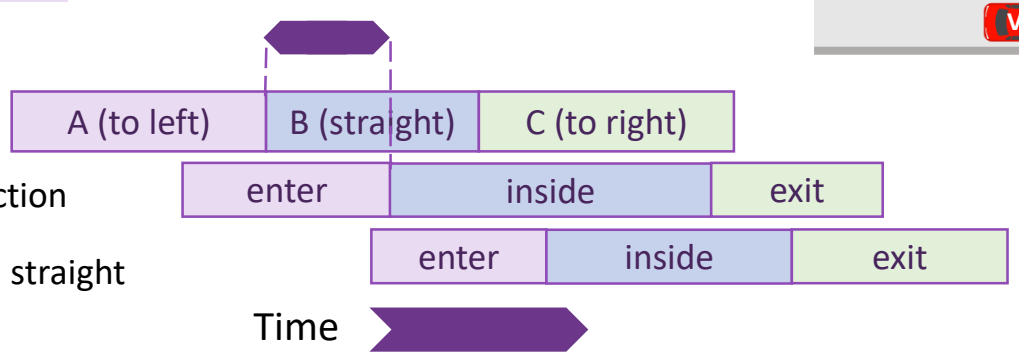
offset



v1 (of *overtake)* does an overtake

v1 (of *turn_right)* turns right in junction

dut (of both) traverses the junction straight



foretellix

Example: Adding new simulator-specific attributes

- Suppose that a specific simulator X supports a new weight attribute for cars. We can add:

```
# Write this in X_config.sdl
extend car:
    weight: real
    keep(weight in [500..4000]kg)
```

Importing this file will add a weight attribute to all cars.
By default will be random in all scenarios.

```
# Write this in X_config_overtake.sdl
extend traffic.overtake:
    keep(v1.weight > 2000kg)
```

Importing this file will constrain weight, but only in overtake

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

foretellix

Example: Driver-in-the-loop

Time

```
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

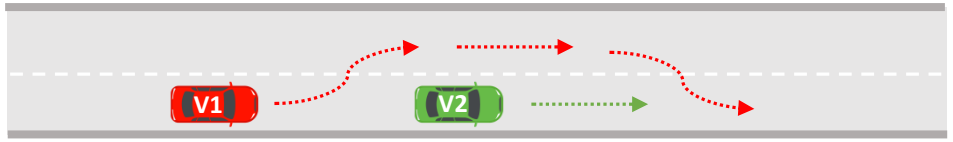
```
scenario dut.DIL_multi_near_hit:
    how_long: time # How long to run it

    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



v1	A (to left)	B (straight)	C (to right)
v2	straight		

Time

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)
```

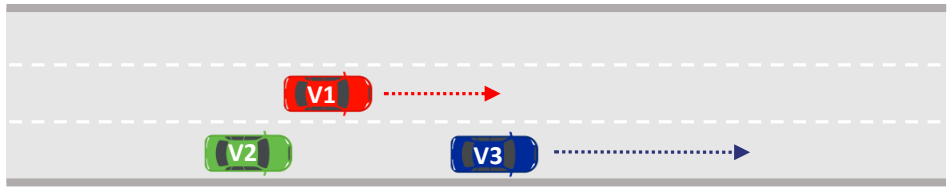
This scenario does two overtakes serially

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

This scenario repeats overake_serial 10 times

foretellix

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.

foretellix

Using event-based synchronization

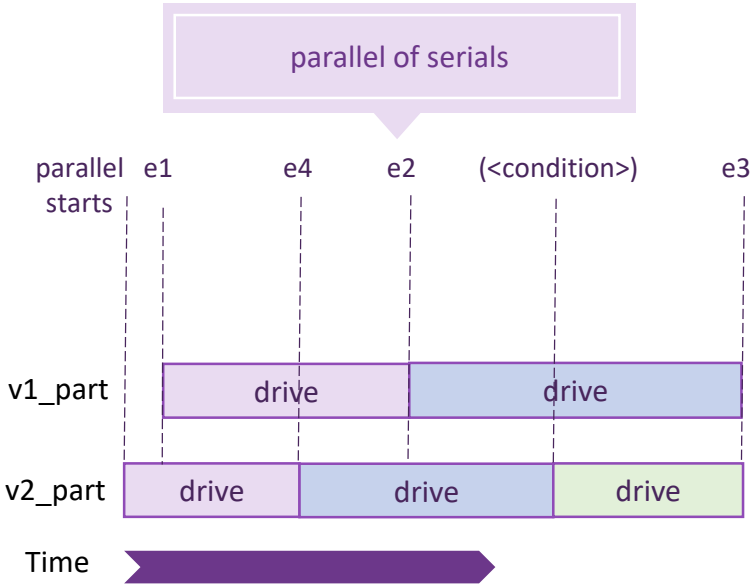
```
scenario traffic.multi_car:
    v1: car # The first car
    v2: car # The second car
    p: path

    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



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
```


foretellix

Thank you

foretellix
verifying autonomous systems